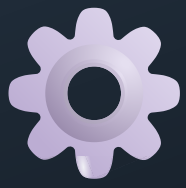


Assembly Fundamentals

A Complete 10-Chapter Course on the LC-3 Teaching Architecture

Topics covered:

The fetch-decode-execute cycle & CPU anatomy · Memory & the four addressing modes

Registers, condition codes & load/store architecture · Instruction encoding, bit by bit

Branching, loops & subroutines · The stack, built entirely by hand

TRAP-based I/O & the OS/hardware boundary · The two-pass assembler & linking

Capstone: a complete, real, working LC-3 program

Exercises: 30 hands-on exercises with worked solutions

Format: A4 · Dark-theme code examples · the prerequisite layer for cpu8bit1 (6502/Z80) and the future x86-64 course

Table of Contents

CHAPTER 1 OF 10

What Assembly Actually Is

Assembly Fundamentals

Chapter 1 · What Assembly Actually Is

Every program you've ever run — a browser, a Python script, this very course's own site — eventually becomes a stream of raw binary instructions a CPU executes one at a time. Assembly language is the thinnest possible layer of human readability wrapped around that binary reality. This chapter explains exactly what that layer is, walks through how a text file of assembly actually becomes a running program, and explains why this course teaches all of it using an invented-for-teaching architecture — **LC-3** — instead of jumping straight into the real, quirk-laden chips this site already covers or will eventually cover.

The Ladder of Abstraction

Every programming language sits somewhere on a ladder between "what the CPU actually does" and "what's convenient for a human to write." Assembly sits one rung above the very bottom.

LEVEL	WHAT ONE LINE LOOKS LIKE	RELATIONSHIP TO MACHINE CODE
Machine code	`0001 001 000 0 00 010`	Is the machine code — raw bits the CPU fetches and decodes directly
Assembly	`ADD R1, R0, R2`	One assembly instruction → almost always exactly one machine instruction
High-level (Python, C, JS)	`x = a + b`	One line → anywhere from a handful to hundreds of machine instructions

That near-1:1 correspondence is the entire point of assembly. A high-level language hides the machine so you can think about the problem; assembly deliberately doesn't — it exists specifically so you can see, instruction by instruction, exactly what the CPU is doing. `py1-1`'s own `print("hello")` compiles down through several layers before it ever reaches a CPU; an LC-3 `ADD` instruction in this course *is*, for all practical purposes, the thing the CPU executes.

The Assemble-Link-Run Pipeline

A text file full of mnemonics like `ADD` and `LD` doesn't run by itself — a CPU has never heard of the word "ADD." Getting from a `.asm` source file to a running program passes through several distinct stages:

```
; source.asm — a human writes this
START   LD   R1, VALUE
        ADD  R1, R1, R1
        HALT
VALUE   .FILL #5
```

1. **Assembler** — reads the source file and translates each mnemonic into its binary machine-code equivalent, resolving labels like `VALUE` into actual memory addresses. Produces an **object file**.
2. **Linker** — combines one or more object files together, resolving any references that point at a *different* file, and produces a single, complete executable image.
3. **Loader** — copies that executable image into memory at the address it expects to run from.
4. **CPU execution** — the processor's own fetch-decode-execute cycle (the full subject of `assembly1-2`) takes over from there, one instruction at a time.

This chapter only previews that pipeline — `assembly1-9` comes back to it in full, explaining exactly how an assembler turns a mnemonic into bits, and exactly what a linker does when a symbol lives in a different file than the one that uses it.

Why Not Just Learn a Real Chip First?

This site already has (or will have) courses on real, historically important processors — the 6502 and Z80 in `cpu8bit1`, and eventually a modern x86-64 course. So why not just start there?

Because every real chip carries decisions that were forced by its own era, not by any universal principle of computing:

- The 6502's famously tiny register set exists largely because MOS Technology built it to be sold for **$25 in 1975** — fewer transistors meant a cheaper, more competitive chip, not a deliberate teaching choice.
- The Z80 was designed as a superset of the Intel 8080, meaning a good portion of its instruction set exists purely to stay **backward-compatible** with a chip that came before it.
- x86-64 carries over **four decades** of backward-compatible accretion — instructions, addressing quirks, and legacy modes that exist only because something built in the 1970s and 80s still has to keep working today.

None of that is a flaw in those chips — it's just historical reality, and `cpu8bit1` and the future x86-64 course both cover it honestly. But it means that learning a real chip *first* means learning "what is a register" and "why does this instruction exist for a 1975 manufacturing reason" at the same time, tangled together. This course exists to untangle them — teaching the universal concepts on their own first, so that by the time you reach a real chip, every quirk you encounter reads as a genuine, nameable design trade-off instead of unexplained noise.

Meet the Teaching Architecture: LC-3

This course uses **LC-3** ("Little Computer 3"), a 16-bit instruction set architecture designed specifically for teaching computer organization — not an invented toy made up for this course, but a real, widely used academic architecture from Patt & Patel's textbook *Introduction to Computing Systems*, taught at universities including UT Austin. LC-3 was deliberately designed to be small enough to fully understand in a single course, yet complete enough to write real, working programs on: 16-bit words, 8 general-purpose registers, a clean and compact instruction set, and thorough public documentation.

Every chapter from here forward writes real LC-3 assembly and traces it against LC-3's actual, documented behavior — nothing in this course is simplified beyond what LC-3 itself already simplifies for you.

CONCEPT PREVIEWED HERE	WHAT IT ROUGHLY MEANS	FULLY EXPLAINED IN
Fetch-decode-execute cycle	The repeating loop every instruction passes through to actually run	assembly1-2
Memory & addressing modes	How an instruction says *where* its data lives	assembly1-3
Registers & the register file	LC-3's 8 fast, on-chip storage slots (R0–R7)	assembly1-4
Instructions & opcodes	The actual operations LC-3 can perform, and their binary encoding	assembly1-5
Condition codes & branching	How a program makes decisions (if/else, loops)	assembly1-6
Subroutines & the stack	Calling and returning from reusable blocks of code	assembly1-7
I/O & system calls	How a program talks to a keyboard, screen, or operating system	assembly1-8
Assemblers & linkers	How this chapter's own pipeline actually works, in full	assembly1-9

THIS CHAPTER IS A MAP, NOT A TEST

Nothing above needs to be memorized yet. Every concept previewed in this chapter gets its own full, hands-on chapter later in the course — this one exists purely to show where the course is headed and why it starts here.

LC-3 IS REAL, NOT INVENTED

It can be tempting to assume a "teaching architecture" means something simplified to the point of being fictional. LC-3 is a genuine, documented ISA used in real university courses — the registers, instructions, and encodings you'll work with here are the actual LC-3 specification, not a loosened approximation of it. What's simplified is the *era* it was designed in, not the rigor of the material.

Hands-On Exercises

EXERCISE 1

Given the three lines below, identify which abstraction level each one belongs to (machine code, assembly, or a high-level language), and explain in your own words why the assembly line has a much closer, more predictable relationship to the machine-code line than the high-level line does: `x = a + b`, `ADD R1, R0, R2`, `0001 001 000 0 00 010`.

View solution

EXERCISE 2

A program is split across two source files, `main.asm` and `helper.asm`. `main.asm` calls a label defined only inside `helper.asm`. Using this chapter's four-stage assemble-link-run pipeline, name the specific stage where that cross-file label reference actually gets resolved, and explain what would go wrong if that stage were skipped.

View solution

EXERCISE 3

Using this chapter's own reasoning in "Why Not Just Learn a Real Chip First?", name two specific historical quirks (one from the 6502, one from the Z80) that this course's use of LC-3 lets you avoid dealing with until `cpu8bit1` — and explain why postponing them helps rather than just delays the learning.

View solution

CHAPTER 1 QUICK REFERENCE

- Machine code → assembly → high-level languages is a ladder of abstraction; assembly sits one rung above raw binary, nearly 1:1 with it
- The pipeline: **assembler** (source → object file) → **linker** (object files → one executable) → **loader** (executable → memory) → CPU execution
- Real chips (6502, Z80, x86-64) carry era-specific historical baggage — cost constraints, backward compatibility, decades of accretion
- **LC-3** — a real, documented 16-bit teaching ISA (Patt & Patel), used to strip that baggage away so the universal concepts come first
- This course's own roadmap: fetch-decode-execute → memory/addressing → registers → instructions → branching → subroutines/stack → I/O → assemblers/linkers
- The teaching architecture is simplified in era, not in rigor — everything taught here is LC-3's real, documented behavior

CHAPTER 2 OF 10

Anatomy of a CPU

Assembly Fundamentals

Chapter 2 · Anatomy of a CPU

`assembly1-1` ended its own assemble-link-run pipeline with a shrug: "CPU execution takes over from there." This chapter opens that shrug up completely. Before writing a single real LC-3 instruction, you need to know what's actually sitting inside the box that's about to run it — the handful of components every CPU is built from, and the repeating loop, the **fetch-decode-execute cycle**, that those components run together to make anything happen at all.

The Building Blocks of a CPU

Strip away the marketing and a CPU is a small number of cooperating parts, each with one job:

COMPONENT	ITS JOB	LC-3 EXAMPLE
Register file	A small set of fast, on-chip storage slots — the CPU's own "scratch paper"	R0–R7 (full detail in assembly1-4)
ALU (Arithmetic Logic Unit)	Does the actual math and logic — addition, AND, NOT — on whatever values it's handed	Executes `ADD` and `AND` instructions
Control unit	Reads the current instruction and decides what everything else should do this cycle	Interprets the opcode bits of every instruction
Buses	The shared "wires" that actually move bits between components	Address bus, data bus, control bus — see below
PC (Program Counter)	A special register holding the memory address of the *next* instruction to fetch	Incremented every cycle unless a branch changes it
IR (Instruction Register)	Holds the instruction currently being decoded and executed	Loaded fresh at the start of every fetch

None of these components do anything meaningful on their own. What makes a CPU a CPU is the specific, repeating sequence in which they hand work off to each other — which is exactly what the rest of this chapter walks through.

Buses — How Parts Talk to Each Other

A bus is nothing more exotic than a shared set of wires that one component can put a value onto, and another component can read a value off of. LC-3's design (and most simple CPUs) uses three logically distinct buses:

- **Address bus** — carries a memory address: "I want to read or write *this* location."
- **Data bus** — carries the actual value being read from or written to that address.
- **Control bus** — carries signals like "this is a read" or "this is a write," coordinating *when* the other two buses' contents are valid.

Two registers act as the CPU's staging area for talking to memory over these buses: the **MAR** (Memory Address Register) holds the address about to go out on the address bus, and the **MDR** (Memory Data Register) holds the value coming back on the data bus. You'll see both of these by name in the very next section.

The Fetch-Decode-Execute Cycle, Step by Step

Every single instruction a CPU ever runs — no matter how simple or complex — passes through the same repeating cycle. LC-3 breaks it into six phases, though not every instruction needs all six:

1. **Fetch** — `MAR ← PC`; the value at that address is read from memory into `MDR`; `MDR`'s contents are copied into the `IR`; and `PC` is incremented so it's already pointing at the *next* instruction.
2. **Decode** — the control unit examines the opcode bits sitting in the `IR` and determines exactly which operation this instruction represents, and which other registers or memory locations it will need.
3. **Evaluate Address** *(only for instructions that reference memory)* — computes the actual memory address the instruction needs, using one of the addressing modes `assembly1-3` covers in full.
4. **Fetch Operands** *(only if needed)* — reads whatever values the instruction needs from the register file or from memory, using the address computed in the previous phase.
5. **Execute** — the ALU actually performs the operation (addition, AND, NOT, and so on).
6. **Store Result** — the result gets written back to wherever it belongs: a register, or a memory location.

Tracing a Real Instruction: ADD R1, R0, R2

Take the exact instruction from `assembly1-1`'s own pipeline example. This is a register-only instruction — it never touches memory for its operands — so phases 3 and 4 are skipped entirely:

```
; Fetch
MAR <- PC              ; address of this instruction
MDR <- mem[MAR]         ; the instruction's bits
IR  <- MDR
PC  <- PC + 1           ; PC now points past this instruction

; Decode
control unit reads IR's opcode bits -> "this is ADD, register mode"
```

```
; Evaluate Address  -- SKIPPED, ADD (register mode) never touches memory
; Fetch Operands
read R0 and R2 from the register file

; Execute
ALU computes R0 + R2

; Store Result
write the ALU's result into R1
```

Now contrast that with a memory-referencing instruction like `LD R1, VALUE` (also from `assembly1-1`'s own example). Fetch, Decode, Execute, and Store Result all still happen — but this time **Evaluate Address** and **Fetch Operands** aren't skipped: Evaluate Address computes the actual memory address `VALUE` refers to, and Fetch Operands reads the value sitting at that address into `MDR` before it's written into R1. `assembly1-3` picks this exact distinction back up when it covers addressing modes in full.

THE CYCLE NEVER STOPS ON ITS OWN

A CPU has no built-in concept of "the program is done" or "wait for input." It simply keeps running fetch-decode-execute, forever, one instruction after another — PC pointing at whatever comes next. A program only stops because it executes a specific instruction that tells the CPU to halt (LC-3's own `HALT`, previewed in `assembly1-1`'s pipeline example, covered in full in `assembly1-8`). "The program finished" isn't something the CPU perceives — it's just one more instruction being fetched and executed like any other.

THIS CHAPTER IS CONCEPTUAL, NOT CYCLE-ACCURATE

Real hardware often overlaps or pipelines these phases for speed (a topic real CPUs like x86-64 lean on heavily). LC-3, and this course, deliberately treat the cycle as six clean, sequential phases — because the goal here is understanding *what* has to happen and *in what order*, not squeezing out performance. That's exactly the kind of historical/real-hardware complexity `assembly1-1` named as a reason to start with a teaching ISA.

Hands-On Exercises

EXERCISE 1

List the six phases of the fetch-decode-execute cycle in order, and explain in your own words what MAR and MDR each hold at the moment the Fetch phase completes.

View solution

EXERCISE 2

Using this chapter's own `ADD R1, R0, R2` trace as a model, explain why the Evaluate Address and Fetch Operands phases are skipped for this instruction but are NOT skipped for `LD R1, VALUE`. What's the general rule that decides

whether those two phases run?

View solution

EXERCISE 3

A classmate says "the CPU knows when a program is done and stops running." Using this chapter's own tip-box reasoning, correct that statement — what does a CPU actually do at every single cycle, and how does a program actually come to a stop?

View solution

CHAPTER 2 QUICK REFERENCE

- **Register file, ALU, control unit, buses, PC, IR** — the core components every CPU is built from
- **Buses**: address bus (where), data bus (what value), control bus (read vs. write, timing)
- **MAR** holds an address about to go to memory; **MDR** holds the value coming back
- The six-phase cycle: **Fetch** → **Decode** → **Evaluate Address** → **Fetch Operands** → **Execute** → **Store Result**
- Evaluate Address / Fetch Operands only run for instructions that actually reference memory
- The CPU has no concept of "done" — it fetches forever until an instruction like HALT explicitly stops it
- This chapter's six clean phases are conceptual — real chips like x86-64 pipeline/overlap them for speed

CHAPTER 3 OF 10

Memory & Addressing

Assembly Fundamentals

Chapter 3 · Memory & Addressing

`assembly1-2` named an "Evaluate Address" phase that only runs for instructions touching memory, without ever saying exactly how that address gets computed. This chapter closes that gap. You'll see what memory actually looks like from the CPU's point of view, why a 16-bit instruction physically cannot hold a full 16-bit address, and the four distinct addressing modes LC-3 uses to work around that limit.

Memory as an Addressable Array

Picture LC-3's memory as one enormous array — 65,536 numbered slots (2^{16}, since LC-3 addresses are 16 bits wide), each one holding exactly one 16-bit **word**. The address is the array index; the word is the value stored there. Reading memory means "give me the value at index *N*"; writing memory means "store this value at index *N*."

LC-3 IS WORD-ADDRESSABLE, NOT BYTE-ADDRESSABLE

Each memory location holds a full 16-bit word, and addresses increment by 1 to move to the *next word* — not the next byte. This is a genuine, real LC-3 design choice, and it's already a difference worth noticing: most modern architectures (including the future x86-64 course) are **byte-addressable**, where each address points at a single 8-bit byte and a 16-bit value spans two consecutive addresses. Neither approach is "more correct" — it's a real trade-off between addressing simplicity and address-space efficiency, and it's the first of several such trade-offs this course will name honestly rather than gloss over.

Why Instructions Can't Just Hold a Full Address

Every LC-3 instruction is exactly 16 bits wide — the same width as a single memory word. A handful of those bits are always spent on the opcode (4 bits) and, often, a destination register (3 bits) and other bookkeeping. That leaves nowhere near enough room to also embed a full, standalone 16-bit address inside the instruction itself.

This single hardware constraint is the entire reason addressing *modes* exist at all: instead of an instruction saying "the address is exactly this," it says "compute the address this specific way," using whatever few bits it has left. That's what the rest of this chapter walks through.

The Four Addressing Modes

1. Immediate — the value is right there in the instruction

No address computation at all — the operand is embedded directly in the instruction's own bits.

```
AND R1, R1, #0   ; the value 0 is baked directly into this instruction
```

Fast and simple, but limited: LC-3 only reserves 5 bits for an immediate value in `ADD` / `AND`, meaning it can represent numbers from -16 to 15 — nowhere near a full 16-bit range.

2. PC-relative — the address is computed from where you already are

Used by `LD`, `ST`, `LEA`, and `BR`. The instruction holds a small, signed 9-bit offset; the CPU adds that offset to the (already-incremented) `PC` to get the target address:

```
effective address = PC + SEXT(offset9)
```

```
START   LD   R1, VALUE   ; address computed as PC + a small offset
        ...
VALUE   .FILL #5
```

This is the exact `LD R1, VALUE` from `assembly1-1`'s pipeline example and `assembly1-2`'s own Evaluate Address trace — you now have the actual formula behind that step. The assembler computes the offset for you at assemble time (per the pipeline from `assembly1-1`) so you can just write a label.

3. Indirect — the address of the address

Used by `LDI` and `STI`. The 9-bit offset works exactly like PC-relative — but instead of pointing at the data itself, it points at *another address*, which in turn holds the real target address:

```
pointer address    = PC + SEXT(offset9)
effective address  = mem[pointer address]
```

```
START   LDI  R1, PTR    ; PTR itself holds an address, not the value
        ...
PTR     .FILL VALUE  ; a pointer
VALUE   .FILL #42    ; the real data
```

If you've ever worked with a pointer-to-a-pointer in C, this is the exact same idea: one extra hop through memory before you reach the real value.

4. Base+offset (indexed) — an address relative to a register

Used by `LDR` and `STR`. Instead of offsetting from the PC, the address is computed relative to whatever value currently sits in a chosen base register:

```
effective address = BaseR + SEXT(offset6)
```

```
LDR R1, R2, #3   ; address = (value in R2) + 3
```

This is the mode that makes arrays and data structures practical — R2 could hold the starting address of a list, and the offset selects which element to read, exactly the way indexed access works in a high-level language.

MODE	EXAMPLE	EFFECTIVE ADDRESS	WHEN YOU'D REACH FOR IT
Immediate	`AND R1, R1, #0`	N/A — value is in the instruction	Small constants (clearing a register, small increments)
PC-relative	`LD R1, VALUE`	PC + offset9	A single, known variable defined nearby in your program
Indirect	`LDI R1, PTR`	mem[PC + offset9]	A pointer whose target can change, or lives far away
Base+offset	`LDR R1, R2, #3`	R2 + offset6	Array elements, struct fields, anything computed at runtime

THESE FORMULAS ARE ASSEMBLY1-2'S MISSING PIECE

Whichever addressing mode an instruction uses, its formula above is exactly what LC-3's Evaluate Address phase computes before Fetch Operands can run. `assembly1-2` deliberately left this step abstract — now you know precisely what's happening inside it for every kind of memory-referencing instruction.

PC-RELATIVE ADDRESSING HAS A LIMITED REACH

Because the offset in PC-relative and indirect addressing is only 9 bits, it can only reach roughly ±256 words from the current instruction — not the entire 65,536-word memory. This is a real, nameable LC-3 design constraint born from the same 16-bit instruction-width limit named earlier in this chapter, not a bug. It previews exactly the kind of concrete, "here's the actual trade-off and why it exists" reasoning `cpu8bit1` and the future x86-64 course apply to their own real chips' addressing quirks.

Hands-On Exercises

EXERCISE 1

Explain, using this chapter's own "Why Instructions Can't Just Hold a Full Address" section, why LC-3 needs multiple addressing modes at all instead of just one instruction format that always stores a full 16-bit address.

View solution

EXERCISE 2

Given `LDI R1, PTR` where `PTR` is stored at address `x3050` and contains the value `x4000`, and memory address `x4000` contains the value `#99`, trace the two-step effective-address computation and state what ends up in R1.

View solution

EXERCISE 3

You need to read the 4th element of an array whose starting address is already sitting in R3 at runtime (the exact index isn't known until the program runs). Which of the four addressing modes from this chapter fits that job, and why would the other three modes fail to work for this specific case?

View solution

CHAPTER 3 QUICK REFERENCE

- LC-3 memory: 65,536 **word**-addressable locations (16 bits each) — not byte-addressable like most modern chips
- A 16-bit instruction has no room to hold a full 16-bit address — this is why addressing modes exist
- **Immediate** — the value is embedded in the instruction itself (small range only)
- **PC-relative** — address = PC + a small signed offset (LD/ST/LEA/BR)
- **Indirect** — address = mem[PC + offset] — a pointer to the real address (LDI/STI)
- **Base+offset (indexed)** — address = a register's value + a small offset (LDR/STR) — the mode behind array/struct access
- PC-relative/indirect offsets are only 9 bits wide, limiting their reach to roughly ±256 words

CHAPTER 4 OF 10

Registers & the Register File

Assembly Fundamentals

Chapter 4 · Registers & the Register File

R0 through R7 have already shown up quietly in every code example since `assembly1-1` — as operands in `ADD`, as a base in `LDR`, as a destination for `LD`. This chapter finally makes them the main subject: what the register file actually is, the two special-purpose pieces of CPU state that sit alongside it (the PC and the condition codes), and — the central question this chapter exists to answer — *why LC-3 has exactly eight of them*.

The Register File — R0 Through R7

LC-3 has eight **general-purpose registers**, R0 through R7, each holding exactly one 16-bit word — the same width as a memory location (per `assembly1-3`). "General-purpose" means exactly what it says: any register can hold a number, a memory address, or a character code, with nothing about the hardware restricting what a given register is "for." Unlike some historical designs with one fixed accumulator register that every arithmetic operation is forced to use, any of R0–R7 can serve as a source or destination for any instruction that takes register operands.

The Program Counter — Not Part of the Register File

`assembly1-2` already introduced the **PC** as the register holding the address of the next instruction to fetch, incremented automatically during every Fetch phase. It's worth being explicit here: the PC is *not* one of R0–R7 — it's a separate, special-purpose register that ordinary instructions like `ADD` or `AND` can never read or write directly. The only way to influence it is through the branch and subroutine-call instructions `assembly1-6` and `assembly1-7` cover.

Condition Codes — N, Z, P

Alongside the register file sits one more small piece of state: three single-bit flags called **N** (negative), **Z** (zero), and **P** (positive). Exactly one of the three is set to 1 at any given time — never zero of them, never more than one.

Any instruction that writes a result into a register — `ADD`, `AND`, `NOT`, and the register-loading instructions from `assembly1-3` (`LD`, `LDR`, `LDI`, `LEA`) — automatically sets N/Z/P based on that result: N if the result was negative, Z if it was exactly zero, P if it was positive. Instructions that don't write to a register — `ST`-family instructions, for example — never touch the condition codes at all.

```
AND R1, R1, #0   ; R1 becomes 0 -> Z is set (N and P cleared)
ADD R1, R1, #5   ; R1 becomes 5 -> P is set (N and Z cleared)
ADD R1, R1, #-9  ; R1 becomes -4 -> N is set (Z and P cleared)
```

Condition codes exist for exactly one reason: so a later instruction can make a decision based on the result of an earlier one. `assembly1-6`'s branch instructions read N/Z/P directly to implement everything from `if` statements to loops — this chapter is what makes that possible.

> **CONDITION CODES REFLECT ONLY THE LAST REGISTER WRITE**
>
> N/Z/P get overwritten by *every single instruction* that writes to a register — including ones that might feel like bookkeeping rather than "real" computation, like an `LEA` just to compute an address. If you need to branch on a result, the branch has to come immediately after the instruction that produced it — anything else that touches a register in between will silently overwrite the flags you meant to check.

Load/Store Architecture — Why So Few Registers?

`assembly1-1` named the 6502's tiny register set as a direct consequence of 1975 manufacturing costs — fewer transistors, cheaper chip. LC-3's own small register file has a completely different origin: it's a deliberate consequence of a design choice called **load/store architecture**.

In a load/store design, ALU operations like `ADD` and `AND` are only ever allowed to work on values already sitting in registers — never directly on a value in memory. If you want to add something to a value stored in memory, you have to explicitly `LD` it into a register first, operate on it there, and `ST` it back out if you need the result saved. There's no shortcut instruction that reaches into memory and modifies it in one step.

That constraint is exactly what keeps the register file small and still practical: because every actual computation happens in registers, and memory is only ever touched deliberately through `LD`/`ST`-family instructions, the CPU never needs dozens of registers to avoid constantly re-loading values — a handful is enough to hold whatever a given calculation is actively working with. Eight also isn't an arbitrary round number: per `assembly1-3`'s own bit-budget reasoning, a register operand only needs **3 bits** to select one of 8 registers ($2^3 = 8$) — comfortably fitting several register fields into a single 16-bit instruction alongside an opcode.

QUESTION	LC-3 (LOAD/STORE, THIS COURSE)	6502 (ASSEMBLY1-1'S OWN EXAMPLE)
Why so few registers?	Deliberate design principle — ALU ops never touch memory directly, so a small register file is genuinely	1975 manufacturing cost — fewer transistors meant a cheaper, more

QUESTION	LC-3 (LOAD/STORE, THIS COURSE)	6502 (ASSEMBLY1-1'S OWN EXAMPLE)
	sufficient	competitive chip
Can an instruction operate on memory directly?	Never — must LD into a register first	Covered in full when `cpu8bit1` reaches the 6502's own addressing modes

THE SAME QUESTION, TWO DIFFERENT ANSWERS

When `cpu8bit1-3` asks "why does the 6502 have so few registers?", you'll already know the shape of that question from this chapter — but the 6502's answer is about 1975 economics, not clean architectural principle. Comparing the two answers directly is exactly the kind of "real, nameable trade-off" reasoning `assembly1-1` promised this course would build toward.

Hands-On Exercises

EXERCISE 1

Using this chapter's own bit-budget explanation, explain why LC-3 has exactly 8 general-purpose registers rather than, say, 16 or 4. Tie your answer to the number of bits needed to select a register.

View solution

EXERCISE 2

Trace the condition codes through this sequence, stating which flag (N, Z, or P) is set after each line: `AND R2, R2, #0`, then `ADD R2, R2, #7`, then `LEA R3, LABEL` (assume LABEL's computed address, as a signed value, is negative), then `ADD R2, R2, #-7`. Which instruction's result do the final condition codes actually reflect?

View solution

EXERCISE 3

Explain, using this chapter's own load/store architecture section, why `ADD R1, VALUE, #1` (attempting to add 1 directly to a value in memory) is not a valid LC-3 instruction, and what the correct two-instruction sequence would look like instead.

View solution

CHAPTER 4 QUICK REFERENCE

- **R0–R7** — 8 general-purpose 16-bit registers, no register has a fixed, hardwired purpose
- The **PC** is a separate special-purpose register, not part of R0–R7, only modifiable via branch/subroutine-call instructions
- **N/Z/P condition codes** — exactly one set at a time, automatically updated by any instruction that writes a register result
- Condition codes reflect only the most recent register-writing instruction — anything in between silently overwrites them
- **Load/store architecture** — ALU instructions only ever operate on registers; memory is touched only via explicit LD/ST
- LC-3's small register file is a deliberate design choice — contrast with the 6502's own manufacturing-cost-driven minimalism (assembly1-1)
- 8 registers = exactly 3 bits to select one, fitting cleanly into a 16-bit instruction (per assembly1-3's bit-budget reasoning)

CHAPTER 5 OF 10

Data Movement & Arithmetic Instructions

Assembly Fundamentals

Chapter 5 · Data Movement & Arithmetic Instructions

`ADD`, `AND`, `LD`, `LDR`, and their relatives have already appeared in nearly every example since `assembly1-1`, always used a little informally. This chapter formalizes all of them properly — and, for the first time, opens up exactly how each one is encoded as actual bits, closing the loop `assembly1-1` left open when it showed `ADD R1, R0, R2` next to its raw binary form without explaining how one became the other.

LC-3's Instruction Set at a Glance

This chapter covers two families: the arithmetic/logic instructions, and the data-movement instructions whose addressing modes `assembly1-3` already explained.

FAMILY	INSTRUCTIONS	WHAT THEY DO
Arithmetic / logic	`ADD`, `AND`, `NOT`	Operate on register values, write a result back to a register
Load (memory → register)	`LD`, `LDR`, `LDI`, `LEA`	Bring a value (or an address) from memory into a register
Store (register → memory)	`ST`, `STR`, `STI`	Write a register's value out to memory

Every load/store instruction pairs directly with one of `assembly1-3`'s four addressing modes: `LD`/`ST` use PC-relative, `LDI`/`STI` use indirect, `LDR`/`STR` use base+offset, and `LEA` uses PC-relative too — but with a twist explained below.

ADD and AND — Two Operand Modes, One Mode Bit

`ADD` and `AND` are unusual among LC-3 instructions in that each one has **two distinct forms**, selected by a single bit in the instruction:

- **Register mode** — both operands come from the register file: `ADD R1, R0, R2` computes R0 + R2.
- **Immediate mode** — one operand comes from a register, the other is a small constant embedded directly in the instruction (per `assembly1-3`'s Immediate addressing mode): `ADD R1, R0, #5` computes R0 + 5.

```
ADD R1, R0, R2   ; register mode — both operands are registers
ADD R1, R0, #5   ; immediate mode — second operand is a literal constant
```

`NOT`, by contrast, only has one form — it's a **unary** operation (one input, logically inverting every bit), so there's no second operand to choose a mode for at all: `NOT R1, R0`.

Opcode Format Anatomy

Every LC-3 instruction is 16 bits, and every one of them starts the same way: the top 4 bits (bits 15–12) are the **opcode**, identifying which of LC-3's operations this instruction is. 4 bits selects one of 16 possible opcodes ($2^4 = 16$) — the same bit-budget logic `assembly1-3` and `assembly1-4` already applied to addresses and registers, now applied to the operation itself.

ADD, Register Mode

0001	DR	SR1	0	00	SR2
opcode	3 bits	3 bits	mode=0	unused	3 bits

Bit 5 is the mode bit — 0 means register mode, and bits [2:0] name a second source register (SR2). `DR` is the destination register that receives the result.

ADD, Immediate Mode

0001	DR	SR1	1	imm5
opcode	3 bits	3 bits	mode=1	5 bits, sign-extended

When bit 5 is 1, the remaining 5 bits are read as a signed immediate value instead of a second register — exactly this 5-bit width is why `assembly1-3`'s immediate mode can only represent numbers from -16 to 15. `AND` is encoded identically, just with a different opcode (0101) in place of ADD's 0001.

NOT — A Fixed Pattern Instead of a Second Operand

1001	DR	SR	111111
opcode	3 bits	3 bits	always all 1s

Because `NOT` only ever needs one source register, it has no mode bit at all — the final 6 bits are simply always fixed to `111111` by the LC-3 specification, occupying the space a second operand would otherwise use.

LD — PC-Relative Load

0010	DR	PCoffset9
opcode	3 bits	9 bits, sign-extended

This is the exact bit layout behind the `PC + SEXT(offset9)` formula `assembly1-3` introduced — the 9-bit field here *is* that offset. `ST`, `LDI`, and `STI` all follow this same DR/SR-plus-PCoffset9 shape (with different opcodes); `LDR`/`STR` instead use a 3-bit BaseR field plus a shorter 6-bit offset, matching `assembly1-3`'s base+offset formula.

LEA NEVER TOUCHES MEMORY AT ALL

`LEA` ("Load Effective Address") uses the exact same PC-relative bit layout as `LD` — but instead of reading the value stored at the computed address, it loads the *address itself* into the destination register. Combined with `LDR`'s base+offset addressing from `assembly1-3`, this is exactly how you'd get an array's starting address into a register before indexing into it.

A 5-BIT IMMEDIATE IS SMALLER THAN IT LOOKS

`ADD R1, R1, #100` looks like ordinary arithmetic, but LC-3's immediate field is only 5 bits — signed, it can hold exactly -16 through 15. Anything larger simply can't be expressed as an immediate operand at all; the assembler will reject it, and the correct fix is to store the larger constant in memory with `.FILL` and `LD` it into a register instead.

Hands-On Exercises

EXERCISE 1

Given the 16-bit pattern `0001 010 011 1 00111`, use this chapter's ADD bitfield diagrams to determine: which opcode this is, which mode (register or immediate), the destination register, the source register, and the operand's decimal value.

View solution

EXERCISE 2

Explain, using this chapter's own bitfield diagrams, why ADD and AND need a mode bit but NOT doesn't — and what NOT's instruction has in the bit positions a mode bit and second operand would otherwise occupy.

View solution

EXERCISE 3

Write LC-3 assembly that loads a value from memory location `COUNT`, adds 200 to it, and stores the result back to `COUNT`. Explain, using this chapter's warn-box, why a single `ADD` instruction with an immediate operand of 200 cannot be used here.

View solution

CHAPTER 5 QUICK REFERENCE

- **ADD / AND** — two forms each, register mode (bit 5 = 0) or immediate mode (bit 5 = 1, 5-bit signed operand, -16 to 15)
- **NOT** — unary, no mode bit; its final 6 bits are always fixed to `111111`
- Every instruction's top 4 bits (15–12) are the opcode — 4 bits selects 1 of 16 possible operations
- **LD/ST/LDI/STI** — DR/SR + a 9-bit PC-relative offset, the literal encoding behind assembly1-3's PC-relative formula
- **LDR/STR** — DR/SR + a 3-bit base register + a 6-bit offset (base+offset addressing)
- **LEA** — same layout as LD, but loads the computed address itself rather than the value stored there
- A 5-bit immediate can only hold -16 to 15 — larger constants must live in memory and be loaded with LD

CHAPTER 6 OF 10

Control Flow — Branching & Conditionals

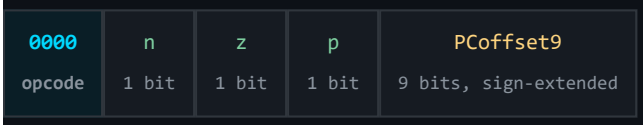
Assembly Fundamentals

Chapter 6 · Control Flow — Branching & Conditionals

`assembly1-4` introduced the N/Z/P condition codes with a promise: "a later instruction can make a decision based on the result of an earlier one." This chapter delivers on that promise. The `BR` instruction family reads N/Z/P directly, and it's the only tool LC-3 gives you for everything a high-level language expresses with `if`, `else`, and loops.

The BR Instruction — Reading Condition Codes

`BR` branches — changes the PC instead of just letting it advance normally — but only if the current condition codes match a set of bits baked into the instruction itself. Its encoding reuses the exact PC-relative shape `assembly1-5` already showed for `LD`:



The three bits right after the opcode aren't a destination register the way `LD`'s were — they're three independent flags. At runtime, the CPU compares those three bits against the current N/Z/P condition codes; if *any* of the bits set to 1 in the instruction match a condition code that's currently set to 1, the branch is taken and the PC jumps to `PC + SEXT(offset9)`, exactly like `LD`'s own address computation. If none match, execution just falls through to the next instruction as normal.

MNEMONIC	N / Z / P BITS	BRANCHES WHEN…
BRn	1 / 0 / 0	the last result was negative
BRz	0 / 1 / 0	the last result was exactly zero
BRp	0 / 0 / 1	the last result was positive
BRnz	1 / 1 / 0	the last result was ≤ 0
BRzp	0 / 1 / 1	the last result was ≥ 0
BRnp	1 / 0 / 1	the last result was nonzero
BR / BRnzp	1 / 1 / 1	always — since exactly one of N/Z/P is always set (assembly1-4)
(no mnemonic)	0 / 0 / 0	never — a technically valid but useless instruction, effectively a no-op

BRNZP IS YOUR UNCONDITIONAL JUMP

Because exactly one of N, Z, or P is always set (per `assembly1-4`), a `BR` with all three bits set is guaranteed to branch every single time, regardless of what the condition codes actually are. LC-3 has no separate "unconditional jump to a label" instruction — `BRnzp` *is* that instruction, just expressed through the same mechanism as every conditional branch.

Implementing if/else

Translate `if (R1 == 0) { R2 = 100; } else { R2 = -100; }` into LC-3. Note that 100 and -100 don't fit in `assembly1-5`'s own 5-bit immediate range, so both constants have to live in memory and be loaded, exactly as that chapter's warn-box described:

```
        ADD R1, R1, #0      ; adds zero — doesn't change R1, but resets N/Z/P to reflect it
        BRz  ISZERO
        LD   R2, NEG100
        BRnzp DONE
ISZERO  LD   R2, POS100
DONE    ; execution continues here either way
        ...
NEG100  .FILL #-100
POS100  .FILL #100
```

That leading `ADD R1, R1, #0` is deliberate, not filler — it's exactly the "re-establish the condition codes right before branching" move `assembly1-4`'s own warn-box called for. If R1's value came from several instructions earlier, and any register-writing instruction ran since, the condition codes might no longer reflect R1 at all without this step.

Implementing Loops

Loops are just conditional branches that point *backward* instead of forward. Here's a loop that sums the numbers from 1 up to whatever's stored at `N`:

```
        AND R2, R2, #0      ; R2 = sum = 0
        LD  R1, N            ; R1 = counter = N
LOOP    ADD R2, R2, R1      ; sum += counter
        ADD R1, R1, #-1     ; counter-- — this ALSO sets condition codes for the new R1
        BRp LOOP            ; loop while counter is still > 0
        ST  R2, SUM
        ...
N       .FILL #5
SUM     .FILL #0
```

Notice there's no separate "test the counter" instruction here at all, unlike the if/else example above — the decrement `ADD R1, R1, #-1` already writes R1 and sets condition codes as a side effect, and `BRp` immediately follows it with nothing in between. That's the condition-code discipline from `assembly1-4` applied correctly: reuse a result's condition codes immediately, or re-establish them deliberately if anything comes between the computation and the branch.

BR CAN'T REACH EVERYWHERE

`assembly1-3` already flagged that a 9-bit PC-relative offset can only reach roughly ±256 words. That limit applies to `BR` exactly as much as it applies to `LD` — a branch target more than ~256 words away from the `BR` instruction itself simply cannot be expressed. LC-3 programs that need to jump further rely on a different instruction entirely, `JMP`, which jumps to whatever address is sitting in a register rather than computing one from an offset — the same mechanism `assembly1-7` uses to implement `RET`.

Hands-On Exercises

EXERCISE 1

Using this chapter's own BR bitfield diagram and mnemonic table, explain why a BR instruction with all three n/z/p bits cleared to 0 is technically valid but never actually useful in a real program.

View solution

EXERCISE 2

In this chapter's if/else example, explain specifically what could go wrong if the leading `ADD R1, R1, #0` were removed, tying your answer back to assembly1-4's own warn-box about condition codes reflecting only the last register write.

View solution

EXERCISE 3

Modify this chapter's own summing loop so it instead counts how many iterations run before the counter reaches zero (i.e. computes the original value of N, but by counting rather than reading it back). Write the full modified loop.

View solution

CHAPTER 6 QUICK REFERENCE

- **BR** — opcode 0000, three condition bits (n/z/p) + a 9-bit PC-relative offset, reusing LD's own PC-relative encoding shape
- Branches if ANY bit set to 1 in the instruction matches a currently-set condition code
- **BRnzp** — always branches, since exactly one of N/Z/P is always set; LC-3's stand-in for an unconditional jump
- **if/else** — set/re-establish condition codes, then branch around the block you want to skip
- **Loops** — a backward-pointing conditional branch; reuse condition codes set by the loop's own decrement when possible
- Always branch immediately after the instruction whose result you're checking, or deliberately re-set flags first (assembly1-4)
- BR shares LD's ±256-word PC-relative reach limit; longer jumps need JMP (previewed for assembly1-7)

CHAPTER 7 OF 10

Subroutines & the Stack

Assembly Fundamentals

Chapter 7 · Subroutines & the Stack

`assembly1-6` ended with a forward pointer: longer jumps rely on `JMP`, "the same mechanism `assembly1-7` uses to implement `RET`." This chapter makes good on that. You'll see how LC-3 calls and returns from reusable blocks of code, why a single register isn't enough once calls start nesting, and how to build a stack entirely out of ordinary instructions — because unlike some real chips this course will eventually meet, LC-3 doesn't give you one for free.

JSR / JSRR — Calling a Subroutine

`JSR` ("Jump to Subroutine") does two things in one instruction: it saves the address of the instruction right after itself into **R7** — the return address — and then jumps to the subroutine, using the same PC-relative idea as `BR`, just with a wider offset:

0100	1	PCoffset11
opcode	PC-relative	11 bits, sign-extended

Notice the offset here is **11 bits**, not the 9 bits `BR` and `LD` use — LC-3 gives subroutine calls roughly four times the reach of a regular branch, since a subroutine is often placed much farther away in a program than a nearby loop target.

`JSRR` does the identical "save R7, then jump" job, but computes its target from a base register instead of a PC-relative offset — useful when the subroutine's address isn't known until runtime:

0100	0	00	BaseR	000000
opcode	register mode	unused	3 bits	unused

```
JSR  MYSUB     ; R7 <- return address; PC <- PC-relative target
JSRR R3        ; R7 <- return address; PC <- value in R3
```

RET — Returning from a Subroutine

`RET` is, at the bit level, nothing more than `JMP R7` — LC-3's specification doesn't even give it a separate opcode. `JMP` jumps unconditionally to whatever address sits in a chosen base register:

1100	000	BaseR	000000
opcode	unused	3 bits	unused

Set `BaseR` to `111` (R7) and you get exactly `RET` — jump to whatever address `JSR`/`JSRR` most recently saved there. The two instructions are a matched pair by *convention*, not by any special hardware connection between them: `JSR` happens to write R7, and `RET` happens to be the specific case of `JMP` that reads it back.

Why Nested Calls Need a Stack

R7 holds exactly one return address at a time. That's fine for a single, non-nested call — but consider subroutine **A** calling subroutine **B**, where B itself calls a third subroutine:

```
A:  ...
    JSR B        ; R7 <- "return to A", PC <- B
    ...

B:  ...
    JSR C        ; R7 <- "return to B" — OVERWRITES A's return address!
    ...
    RET          ; jumps using R7 — but R7 no longer points back to A
```

The instant B's own `JSR C` runs, it overwrites R7 with B's return address, permanently destroying the return address A was counting on. By the time B's `RET` eventually runs, R7 correctly returns to A's caller-side location — but only because C's own return already restored it along the way, which only works by luck of ordering, not by design. A single shared register simply cannot hold more than one pending return address, and nested calls need more than one.

LC-3 Has No Dedicated Stack Pointer — You Build the Stack Yourself

This is exactly the kind of problem a hardware **stack** — a last-in-first-out storage structure with a dedicated stack-pointer register the CPU manages automatically — is built to solve. LC-3 deliberately doesn't provide one. There's no register reserved by the hardware for this purpose, and no dedicated `PUSH`/`POP` instructions. By convention, LC-3 programs simply dedicate one general-purpose register — commonly **R6** — to act as a software-maintained stack pointer, and build push/pop behavior out of instructions you already know:

```
; PUSH R0 onto the stack (stack conventionally grows downward)
ADD R6, R6, #-1   ; move the stack pointer down one slot
STR R0, R6, #0    ; store R0 at the new top-of-stack address

; POP into R0
```

```
LDR R0, R6, #0    ; read the value currently at top-of-stack
ADD R6, R6, #1    ; move the stack pointer back up
```

Both sequences lean directly on `assembly1-3`'s base+offset addressing mode (`STR` / `LDR` with R6 as the base) and `assembly1-4`'s load/store discipline — there's nothing new here mechanically, just a convention for how to use what you already have.

A Full Calling Convention — Saving R7 Around a Nested Call

With a working stack, A can safely call B, and B can safely call C, by pushing R7 before making its own nested call and popping it back right after:

```
A:      ...
        JSR B
        ...

B:      ; save R7 before B makes its own nested call
        ADD R6, R6, #-1
        STR R7, R6, #0

        JSR C          ; safe now — B's own return address is preserved on the stack

        ; restore R7 before returning to A
        LDR R7, R6, #0
        ADD R6, R6, #1
        RET              ; now correctly returns to A
```

This push-before-call, pop-before-return pattern is the actual calling convention that makes nesting safe to any depth — every subroutine that itself calls another subroutine follows the same rule.

FORGETTING TO SAVE R7 IS A CLASSIC, SILENT BUG

Skip the push/pop around a nested `JSR` and the program won't crash immediately — it'll run `RET` successfully, just to the *wrong place*, because R7 no longer holds the return address you actually needed. The symptom shows up far from the actual mistake, which is exactly what makes it a notoriously easy bug to introduce and a hard one to trace back to its source.

LC-3'S SOFTWARE STACK IS A REAL, DELIBERATE DESIGN CHOICE — NOT A MISSING FEATURE

When `cpu8bit1` covers the 6502's fixed, hardware-managed page-1 stack (with its own dedicated SP register) and the Z80's more flexible, general-purpose SP, both will read as two different, more automated answers to the exact same nested-return-address problem this chapter just solved entirely in software. LC-3 doesn't lack a stack because it's incomplete — it simply keeps stack management as one more thing built from primitives you already understand, rather than hiding it behind dedicated hardware.

ARCHITECTURE	STACK POINTER	PUSH/POP
LC-3 (this chapter)	A general-purpose register (R6, by convention only)	Built manually from ADD/STR/LDR
6502 (preview, cpu8bit1)	Dedicated hardware SP, fixed to page 1 of memory	Native instructions, but constrained to that fixed page
Z80 (preview, cpu8bit1)	Dedicated, flexible 16-bit SP register	Native PUSH/POP, anywhere in memory

Hands-On Exercises

EXERCISE 1

Explain, in your own words and using this chapter's own bitfield diagrams, exactly why RET requires no dedicated opcode of its own — what specific values does JMP's encoding need in order to behave as RET?

View solution

EXERCISE 2

Using this chapter's own A-calls-B-calls-C example, explain precisely which register gets overwritten, when, and why the program's return path breaks specifically because a single register can't hold more than one pending return address at once.

View solution

EXERCISE 3

Write a PUSH/POP pair (using this chapter's own R6-based convention) that saves and restores BOTH R0 and R7 around a nested JSR call, in the correct order — and explain why the restore order must be the exact reverse of the save order.

View solution

CHAPTER 7 QUICK REFERENCE

- **JSR** — saves the return address in R7, jumps via an 11-bit PC-relative offset (wider reach than BR's 9 bits)
- **JSRR** — same save-and-jump behavior, but jumps to an address held in a base register
- **RET** — literally JMP R7; no separate opcode, just a specific case of JMP
- A single register (R7) can only hold one pending return address — nested calls overwrite it

- LC-3 has no dedicated stack pointer register — by convention, a general-purpose register (commonly R6) plays that role
- **PUSH** = decrement R6, then STR the value; **POP** = LDR the value, then increment R6
- Push R7 before a nested JSR, pop it back before RET — the calling convention that makes any depth of nesting safe
- Forgetting to save R7 doesn't crash the program — it returns to the wrong place, far from the actual mistake

CHAPTER 8 OF 10

Input/Output & System Calls

Assembly Fundamentals

Chapter 8 · Input/Output & System Calls

`HALT` has quietly ended nearly every code example since `assembly1-1`, and `assembly1-2`'s own tip-box promised it would eventually be explained in full. This chapter finally does that — and, along the way, introduces the single instruction behind every bit of keyboard and screen interaction an LC-3 program can perform: `TRAP`.

The TRAP Instruction — Calling Into the Operating System

`TRAP` behaves a lot like `assembly1-7`'s own `JSR` — it saves a return address into R7 and transfers control elsewhere — but instead of jumping to a label you wrote yourself, it jumps to a pre-defined **operating system service routine**, looked up through a small table of addresses called the **trap vector table**.

The 8-bit `trapvect8` field isn't an offset the way `BR`'s or `LD`'s fields were — it's an index into the trap vector table, a small region of memory holding the *actual* starting address of each OS routine. `TRAP x25` doesn't jump to address `x25` directly; it looks up whatever address is stored at vector table entry `x25`, and jumps there.

The OS/Hardware Boundary

This indirection exists for a real reason: your program never needs to know exactly how the keyboard or display hardware actually works, what memory-mapped registers they use, or how their timing quirks are handled. It only needs to know the trap number for "read a character" or "print a string." The operating system's own service routines — sitting behind that vector table — are the only code that actually deals with the hardware directly. `TRAP` is the boundary between "your program's logic" and "the OS/hardware layer that talks to the real world," and it's a boundary this course has been relying on silently since its very first example.

The Standard TRAP Routines

MNEMONIC	VECTOR	WHAT IT DOES
GETC	x20	Reads one character from the keyboard into R0 — no echo to the screen
OUT	x21	Writes the character currently in R0 to the display
PUTS	x22	Writes a null-terminated string, starting at the address in R0, to the display
IN	x23	Prints a prompt, reads one character (with echo this time), stores it in R0
HALT	x25	Stops the fetch-decode-execute cycle entirely

GETC NEVER ECHOES WHAT WAS TYPED

A very common early mistake: calling `TRAP x20` (GETC) to read a character and expecting to see it appear on screen automatically. It doesn't — GETC deliberately reads silently. If you want the typed character to actually show up, you have to explicitly follow it with `TRAP x21` (OUT), or use `TRAP x23` (IN) instead, which reads *and* echoes in one call.

HALT, In Full

Recall `assembly1-2`'s own warning: "the CPU has no concept of 'done' — it fetches forever until an instruction like HALT explicitly stops it." Now the mechanism behind that is fully visible: `HALT` is simply `TRAP x25`. Its OS service routine doesn't return control back to your program the way `GETC` or `OUT` do — instead, it halts the fetch-decode-execute cycle at the hardware level, the one and only way execution actually stops.

A Small Working Example

```
        TRAP x20        ; GETC — read one character into R0 (no echo)
        TRAP x21        ; OUT — echo it back manually
        LEA  R0, MSG    ; R0 <- address of the message (assembly1-3/assembly1-5's LEA)
        TRAP x22        ; PUTS — print the null-terminated string at that address
        TRAP x25        ; HALT
MSG     .STRINGZ "Thanks!"
```

This reads one character, echoes it manually (per this chapter's own warn-box), prints a short message using `LEA` to get the string's address into R0 exactly the way `assembly1-5` introduced, and halts.

TRAP INHERITS JSR'S OWN R7 RULES

Because TRAP saves a return address into R7 exactly like JSR, everything `assembly1-7` established about nested calls applies here too: if a subroutine you wrote uses a TRAP call and R7 hasn't been saved on the stack first, that TRAP will silently overwrite the subroutine's own pending return address, the same bug from `assembly1-7`'s own warn-box — just triggered by TRAP instead of a nested JSR.

Polling vs. Interrupts — A First Preview

Under the hood, the OS's `GETC` routine has to somehow know when a key has actually been pressed. The simple technique — and the one this course's examples rely on without you ever needing to think about it — is **polling**: repeatedly checking a status register in a tight loop until it indicates a key is ready. It works, but it wastes CPU cycles doing nothing but asking "is it ready yet?" over and over.

A more advanced alternative is **interrupts**: instead of the CPU repeatedly asking, the keyboard hardware itself signals the CPU the instant a key is pressed, letting the CPU do other useful work in between. This course doesn't go further than naming the distinction — interrupt-driven I/O is real, genuinely more efficient LC-3 territory, but it adds real complexity this Fundamentals course deliberately leaves for later, more advanced material, the same way `assembly1-1` deferred real-hardware quirks to `cpu8bit1` and the future x86-64 course.

Hands-On Exercises

EXERCISE 1

Explain, using this chapter's own "OS/Hardware Boundary" section, why TRAP looks up an address through a vector table instead of just jumping directly to a fixed address the way BR or JSR does with a PC-relative offset.

View solution

EXERCISE 2

A learner writes `TRAP x20` followed immediately by `TRAP x25`, expecting to see the character they typed appear on screen before the program halts. Using this chapter's own warn-box, explain what they'll actually observe, and fix their code.

View solution

EXERCISE 3

A subroutine called via JSR itself calls TRAP x22 (PUTS) to print a status message before returning. Using this chapter's own tip-box and assembly1-7's calling convention, explain what safeguard this subroutine needs around its TRAP call, and why.

View solution

CHAPTER 8 QUICK REFERENCE

- **TRAP** — opcode 1111 + an 8-bit trapvect8 index into the trap vector table; saves a return address into R7 like JSR
- TRAP is the boundary between your program's logic and the OS/hardware layer that actually talks to the keyboard/display
- **GETC (x20)** — read a character, no echo · **OUT (x21)** — print R0's character · **PUTS (x22)** — print a null-terminated string · **IN (x23)** — read with prompt+echo · **HALT (x25)** — stop the fetch-decode-execute cycle for good
- GETC never echoes automatically — pair it with OUT, or use IN instead
- HALT is just TRAP x25 — this is the actual mechanism behind assembly1-2's "a program stops via an explicit HALT instruction"
- TRAP inherits JSR's R7-overwrite risk — save R7 first if a subroutine both calls another routine AND uses TRAP
- **Polling** (repeatedly checking a status register) vs. **interrupts** (hardware signals the CPU) — interrupts are real, more advanced LC-3 material deliberately left for later

CHAPTER 9 OF 10

From Assembly to Machine Code

Assembly Fundamentals

Chapter 9 · From Assembly to Machine Code

Every chapter since `assembly1-1` has quietly leaned on labels — `LOOP`, `DONE`, `N` — without ever explaining how the assembler figures out the actual numeric address each one refers to. This chapter closes that loop for good: the real algorithm an assembler runs, how it builds and uses a symbol table, what those `.FILL` / `.STRINGZ` lines scattered through every example actually are, and — finally, in full mechanical detail — how `assembly1-1`'s own linker stage resolves a label that lives in a completely different file.

The Problem: Forward References

Look back at `assembly1-6`'s own summing loop: the very first real instruction, `LD R1, N`, refers to the label `N` — but `N`'s own line doesn't appear until several lines *later* in the source file. To compute `LD`'s PC-relative offset (per `assembly1-3`'s formula), the assembler needs to know `N`'s actual memory address — but if it only ever reads the file once, top to bottom, generating machine code as it goes, it would hit `LD R1, N` long before it has any idea where `N` will end up. This is a **forward reference**, and it's the exact problem the rest of this chapter exists to solve.

The Two-Pass Algorithm

Real assemblers — including LC-3's — solve this by reading the source file *twice*:

- **Pass 1** — scan the file top to bottom without generating any machine code yet. Track a running **location counter**, starting at the program's origin address. Every time a label appears, record it — along with the current value of the location counter — in a **symbol table**. Advance the location counter by one for every instruction or data word, whether or not it has a label.
- **Pass 2** — scan the file again, this time actually generating the 16-bit encoding for each line (per `assembly1-5`'s bitfield formats). Any time an operand is a label, look its address up in the symbol table built during Pass 1, and compute whatever offset the instruction's own addressing mode needs.

By the time Pass 2 needs `N`'s address, Pass 1 has already scanned the *entire* file and recorded it — forward references stop being a problem because the symbol table is fully built before any actual encoding happens.

Directives — Instructions to the Assembler, Not the CPU

A few lines in every example so far aren't real LC-3 instructions at all — they're **assembler directives**: instructions for the assembler itself, about how to lay out memory, that never get fetched or executed by the CPU.

DIRECTIVE	WHAT IT TELLS THE ASSEMBLER
.ORIG	The starting address (location counter value) the program should be loaded at
.END	Marks the end of the source file — nothing after this line is assembled
.FILL	Reserve one word, initialized to a given value (used since assembly1-1's own pipeline example)
.STRINGZ	Reserve a null-terminated string, one character per word (used in assembly1-8)
.BLKW	Reserve a block of N words, left uninitialized

A Worked Trace — Building the Symbol Table

Take `assembly1-6`'s own summing loop, now with `.ORIG` / `.END` added, assembled starting at address `x3000`:

```
        .ORIG x3000
        AND R2, R2, #0      ; x3000
        LD  R1, N           ; x3001 — forward reference to N
LOOP    ADD R2, R2, R1      ; x3002
        ADD R1, R1, #-1     ; x3003
        BRp LOOP            ; x3004 — backward reference to LOOP
        ST  R2, SUM         ; x3005
        TRAP x25            ; x3006
N       .FILL #5            ; x3007
SUM     .FILL #0            ; x3008
        .END
```

Pass 1 walks top to bottom, building this symbol table:

LABEL	ADDRESS
LOOP	x3002
N	x3007
SUM	x3008

Pass 2 now computes real offsets, using `assembly1-3`'s `PC + SEXT(offset9)` formula (remembering the PC used is always the address *after* the current instruction):

- `LD R1, N` at x3001: offset = N − (x3001+1) = x3007 − x3002 = **+5** (a forward reference, resolved only because Pass 1 already knew N's address)
- `BRp LOOP` at x3004: offset = LOOP − (x3004+1) = x3002 − x3005 = **−3** (a backward reference — negative, since it points earlier in the program)
- `ST R2, SUM` at x3005: offset = SUM − (x3005+1) = x3008 − x3006 = **+2**

"VALUE OUT OF RANGE" IS A REAL PASS 2 ERROR

`assembly1-3` and `assembly1-6` both warned that PC-relative offsets are limited to roughly ±256 words (9 bits) or ±1024 words for JSR's 11-bit offset. During Pass 2, if a label's actual computed offset exceeds what the instruction's offset field can hold, the assembler can't just quietly truncate it — it reports a genuine assembly-time error. Those earlier warnings weren't abstract concerns; this is the exact moment they'd stop your program from assembling at all.

Object Files & the Linker — Multi-File Programs Resolved For Real

`assembly1-1`'s own Exercise 2 asked what happens when `main.asm` references a label defined only in `helper.asm`. Here's the full mechanism: each file goes through its own independent two-pass assembly, producing its own **object file** and its own local symbol table. When `main.asm`'s Pass 2 hits a reference to a label that never appeared anywhere in `main.asm`'s own Pass 1 symbol table, the assembler can't compute a real offset — instead, it records that reference as **unresolved**, tagged as external, inside `main.asm`'s object file.

The **linker** is what closes this gap: it reads every object file being combined, collects each one's exported symbols (labels meant to be visible to other files) and each one's unresolved external references, matches them up, and patches the real, final addresses into place — producing one complete, fully-resolved executable image with no unresolved references left anywhere.

A SYMBOL TABLE IS JUST A LOOKUP — NAME TO ADDRESS

Strip away the assembler terminology and a symbol table is nothing more exotic than a dictionary: a label's text name maps to a numeric memory address, looked up whenever that label is used as an operand. Pass 1 builds the dictionary; Pass 2 (and, for cross-file references, the linker) is what actually looks values up in it.

Hands-On Exercises

EXERCISE 1

Using this chapter's own two-pass algorithm, explain specifically why a single top-to-bottom pass (generating machine code immediately, line by line) cannot correctly assemble `LD R1, N` when N is defined later in the same file — and why Pass 1 solves it.

View solution

EXERCISE 2

Using this chapter's own worked trace, compute the Pass 2 offset for a new instruction `ST R1, N` if it were inserted immediately after the existing `LD R1, N` line at address x3001 (so the new instruction sits at x3002, shifting LOOP and everything after it down by one address). Show your work.

View solution

EXERCISE 3

Revisit assembly1-1's own Exercise 2 scenario (main.asm calling a label defined only in helper.asm) using this chapter's real mechanics. Name specifically what main.asm's object file contains for that reference after its own Pass 2, and what the linker does to finally resolve it.

View solution

CHAPTER 9 QUICK REFERENCE

- **Forward reference** — a label used before its own definition is reached in a single top-to-bottom scan
- **Pass 1** — builds the symbol table (label → address) without generating any machine code
- **Pass 2** — generates real machine code, looking up label addresses in the symbol table Pass 1 built
- **Directives** (.ORIG, .END, .FILL, .STRINGZ, .BLKW) — instructions to the assembler about memory layout, never executed by the CPU
- Offsets are computed as target address − (this instruction's address + 1) — forward references are positive, backward references are negative
- A label's offset exceeding its field's range (±256 for 9 bits, ±1024 for 11 bits) is a real Pass 2 assembly error, not just a theoretical limit
- **Object file** — one file's own assembled output, with any cross-file references left marked unresolved
- **Linker** — combines multiple object files, matching unresolved external references against other files' exported symbols and patching in real addresses

CHAPTER 10 OF 10

Capstone: Writing a Complete LC-3 Program

Assembly Fundamentals

Chapter 10 · Capstone — Writing a Complete LC-3 Program

Nine chapters have each built one working piece of LC-3 — the fetch-decode-execute cycle, addressing modes, registers and condition codes, the instruction encodings, branching, subroutines and the stack, TRAP-based I/O, and finally the assembler mechanics that turn all of it into bits. This capstone combines every one of those pieces into a single, real, working program: a **vowel counter** that reads a short line of typed lowercase letters, counts how many are vowels, and prints the result.

The Program

Reads up to 8 characters from the keyboard (stopping early if Enter is pressed), echoing each one as it's typed. Once input stops, it walks back through what was typed, uses a subroutine to test each character, and prints the final vowel count as a single digit.

```
        .ORIG x3000

; === Vowel Counter ===
        AND R3, R3, #0       ; R3 = number of characters read so far = 0
        LEA R4, BUFFER       ; R4 = pointer to the next free buffer slot

READLOOP
        TRAP x20             ; GETC — read one character into R0 (no echo)
        ADD R1, R0, #-13      ; is it Enter (ASCII 13)?
        BRz READDONE         ; if so, stop reading

        TRAP x21             ; OUT — echo the character back to the screen
        STR R0, R4, #0       ; store the character into the buffer
        ADD R4, R4, #1       ; advance the buffer pointer
        ADD R3, R3, #1       ; count++

        ADD R1, R3, #-8      ; is the buffer full (8 characters)?
        BRz READDONE         ; if so, stop reading
        BRnzp READLOOP

READDONE
        AND R2, R2, #0       ; R2 = vowel count = 0
        LEA R4, BUFFER       ; R4 = pointer, reset to the start of the buffer
        ADD R3, R3, #0       ; refresh condition codes on R3 (assembly1-4 discipline)
        BRz PRINTRESULT      ; nothing was typed at all — skip straight to printing 0
```

```
COUNTLOOP
        LDR R0, R4, #0      ; load the next character from the buffer
        JSR ISVOWEL          ; R0 in, R0 out: 1 if vowel, else 0
        ADD R2, R2, R0       ; vowel count += result
        ADD R4, R4, #1      ; advance the pointer
        ADD R3, R3, #-1     ; remaining--
        BRp COUNTLOOP       ; loop while characters remain

PRINTRESULT
        LEA R0, MSG
        TRAP x22            ; PUTS — "Vowels found: "
        LD  R1, ASCII0       ; '0' won't fit as a 5-bit immediate (assembly1-5) — load it from memory
        ADD R0, R2, R1       ; R0 = vowel count + '0' = the correct ASCII digit
        TRAP x21            ; OUT — print the digit
        AND R0, R0, #0
        ADD R0, R0, #10      ; x0A newline — 10 DOES fit as a 5-bit immediate, no memory constant needed
        TRAP x21            ; OUT — print the newline
        TRAP x25            ; HALT

; --- Subroutine: ISVOWEL ---
; Input: R0 = a character. Output: R0 = 1 if a lowercase vowel, else 0. Clobbers R1.
; Uses assembly1-5's own NOT-then-add-1 idiom to negate a value, since LC-3
; has no direct "compare register to register" instruction — only ADD.
ISVOWEL
        LD  R1, VOWEL_A
        NOT R1, R1
        ADD R1, R1, #1       ; R1 = -'a' (two's complement negation)
        ADD R1, R0, R1       ; R1 = R0 - 'a'
        BRz ISVOWEL_YES

        LD  R1, VOWEL_E
        NOT R1, R1
        ADD R1, R1, #1
        ADD R1, R0, R1
        BRz ISVOWEL_YES

        LD  R1, VOWEL_I
        NOT R1, R1
        ADD R1, R1, #1
        ADD R1, R0, R1
        BRz ISVOWEL_YES

        LD  R1, VOWEL_O
        NOT R1, R1
        ADD R1, R1, #1
        ADD R1, R0, R1
        BRz ISVOWEL_YES

        LD  R1, VOWEL_U
        NOT R1, R1
        ADD R1, R1, #1
        ADD R1, R0, R1
        BRz ISVOWEL_YES

        AND R0, R0, #0       ; not a vowel
        RET
```

```
ISVOWEL_YES
        AND R0, R0, #0
        ADD R0, R0, #1
        RET

BUFFER    .BLKW #8
MSG       .STRINGZ "Vowels found: "
ASCII0    .FILL x0030
VOWEL_A  .FILL x0061
VOWEL_E  .FILL x0065
VOWEL_I  .FILL x0069
VOWEL_O  .FILL x006F
VOWEL_U  .FILL x0075

        .END
```

ISVOWEL DOESN'T NEED TO SAVE R7 — AND THAT'S WORTH NOTICING

`assembly1-7` established push-before-call, pop-before-return as the rule for safe nesting. `ISVOWEL` never itself executes a `JSR` or `TRAP`, so nothing inside it can overwrite the return address the calling `JSR ISVOWEL` placed in R7 — its own `RET` is safe to use directly. The save/restore dance from `assembly1-7` is only necessary when a subroutine's own body might overwrite R7 before it returns; recognizing when it's *not* needed is just as important as applying it correctly when it is.

Chapter Attribution

PROGRAM PIECE	CONCEPT	FROM
The whole program's execution	Fetch-decode-execute cycle running every instruction	assembly1-2
STR/LDR into BUFFER, LEA R4	Base+offset addressing, PC-relative LEA for the buffer's address	assembly1-3
R0–R4 usage, condition-code refresh on R3	The register file, and reusing/re-establishing N/Z/P deliberately	assembly1-4
ADD/AND/NOT, immediate vs. register operands, ASCII0/VOWEL_* constants	Arithmetic instructions and the 5-bit immediate range limit	assembly1-5
BRz, BRp, BRnzp throughout	Condition-code-driven branching for both loops and the vowel test	assembly1-6
JSR ISVOWEL / RET	Subroutine call and return via R7	assembly1-7
TRAP x20/x21/x22/x25	Keyboard input, echoing, string output, and halting	assembly1-8

PROGRAM PIECE	CONCEPT	FROM
.ORIG/.END/.BLKW/.STRINGZ/.FILL, labels throughout	Assembler directives and the two-pass symbol resolution behind every label used here	assembly1-9

HONEST SCOPE NOTE

This program deliberately stays within what this course actually taught, which means real limitations: it only recognizes lowercase vowels (no case-insensitivity logic); a typed line longer than 8 characters is silently cut off rather than erroring or growing the buffer; the vowel count is printed as a single digit because converting a general multi-digit number to decimal text is a genuinely separate technique this course never covered; and keyboard input still relies on the polling `assembly1-8` only previewed, not the interrupt-driven alternative. None of these are bugs — they're honest boundaries of a Fundamentals course, and several of them (multi-digit output, real hardware I/O timing) are exactly the kind of ground `cpu8bit1` and the future x86-64 course pick up from here.

Hands-On Exercises

EXERCISE 1

Trace this program's own COUNTLOOP by hand for the input "bee" (3 characters, all read before Enter). Show the value of R2 (vowel count) after each pass through the loop, and the final digit character printed.

View solution

EXERCISE 2

Explain, using this chapter's own tip-box and assembly1-7's calling convention, exactly what WOULD go wrong if ISVOWEL itself called a nested subroutine (say, to log each character) without first saving R7 — trace the effect on the main program's own COUNTLOOP.

View solution

EXERCISE 3

This chapter's own warn-box names "no case-insensitivity" as an honest limitation. Using assembly1-5's ISVOWEL comparison technique as a model, describe (in words or pseudocode, not full assembly) the minimum change needed to also recognize uppercase vowels A/E/I/O/U — without writing ten separate comparisons.

View solution

CHAPTER 10 QUICK REFERENCE — COURSE RECAP

- **assembly1-1** — machine code vs. assembly, the assemble-link-run pipeline, why LC-3
- **assembly1-2** — CPU components, the six-phase fetch-decode-execute cycle
- **assembly1-3** — memory as a word-addressable array, the four addressing modes
- **assembly1-4** — R0–R7, the PC, N/Z/P condition codes, load/store architecture
- **assembly1-5** — ADD/AND/NOT, register vs. immediate modes, opcode bit encoding
- **assembly1-6** — BR and condition-code-driven if/else and loops
- **assembly1-7** — JSR/JSRR/RET, why nesting needs a stack, manual PUSH/POP
- **assembly1-8** — TRAP, the OS/hardware boundary, GETC/OUT/PUTS/HALT
- **assembly1-9** — the two-pass assembler, symbol tables, directives, linking
- **assembly1-10** — all of the above, combined into one real, working program
- Next stop: `cpu8bit1` (6502/Z80) applies these exact concepts to real, historically quirky hardware