8-Bit CPUs — 6502/6510 & Z80

A Complete 12-Chapter Comparative Course on Two Real, Historic Chips

Topics covered:

The 1975-76 microprocessor boom · Cost-driven minimalism vs. compatibility-driven richness

Full register, stack & addressing-mode coverage for both chips

A real side-by-side routine, cycle-cost honesty, and a real subroutine call on both

Interrupts (NMI/IRQ/BRK vs. IM 0/1/2) · Real machines: C64, NES, ZX Spectrum, Game Boy, CP/M

RISC vs. CISC formally defined, with the real Acorn/ARM/6502 lineage

Exercises: 36 hands-on exercises with worked solutions

Format: A4 · Dark-theme code examples · the direct historical-hardware sequel to Assembly Fundamentals (LC-3)

Table of Contents

CHAPTER 1 OF 12

Two Chips, One Era

8-Bit CPUs — 6502/6510 & Z80

Chapter 1 · Two Chips, One Era

The `assembly1` course taught every universal concept a CPU needs — registers, memory, the fetch-decode-execute cycle, the stack — using LC-3, an architecture built specifically so none of history's messiness would get in the way. This course exists to reintroduce that messiness, on purpose. Within a single year, 1975 and 1976, two real chips shipped that shaped an entire generation of home computers — and they solved the exact same problems LC-3 solved cleanly, under real financial and competitive pressure, in two almost opposite ways.

1975–76 — The Microprocessor Boom

In the early 1970s, Intel's 8080 was the microprocessor to build around — powerful for its time, and priced at **$179**, out of reach for most hobbyists and small companies. Two chips arrived within a year of each other specifically to disrupt that:

- **MOS Technology 6502** (1975) — designed by Chuck Peddle and a small team who had left Motorola specifically to build something radically cheaper. It launched at **$25** — a price cut so aggressive it was demonstrated by literally handing out chips at a trade show. That price point wasn't a side effect of the design; it *was* the design goal, and `cpu8bit1-2` traces exactly how it shaped every architectural decision that followed.
- **Zilog Z80** (1976) — designed by Federico Faggin, who had previously helped design the very Intel 8080 the Z80 was now competing against. Rather than starting from a clean slate, Faggin built the Z80 as a **superset of the 8080** — every existing 8080 program would still run on it, while adding real new capability on top. `cpu8bit1-5` covers exactly what that compatibility commitment cost, and what it bought.

Two chips, two completely different founding constraints — one built around a price target, one built around compatibility — and both shipped into the same booming, cash-strapped hobbyist and early home-computer market at almost the same moment.

A Quick Bridge — Baseline Vocabulary from assembly1

This course assumes the concepts `assembly1` already built, rather than re-teaching them from zero. If any of the terms below feel unfamiliar, that course is the place to go back to first.

CONCEPT	FULLY TAUGHT IN	HOW IT'LL DIFFER HERE
Fetch-decode-execute cycle	assembly1-2	Same conceptual cycle — but now real cycle counts and timing genuinely matter
Registers	assembly1-4	Small, historically-constrained register sets, organized very differently from LC-3's uniform R0–R7
Addressing modes	assembly1-3	Real addressing modes shaped by cost and compatibility pressure, not clean pedagogical design
The stack	assembly1-7	Real, hardware-managed stacks — the manual PUSH/POP LC-3 required is built into the chip itself

WHY COMPARE TWO CHIPS INSTEAD OF JUST PICKING ONE

`assembly1-1` and `assembly1-4` already used this same trick — contrasting LC-3's clean design against the 6502's cost-driven one made both easier to actually understand. Studying the 6502 and Z80 side by side does the same thing at a larger scale: it becomes possible to tell which quirks are genuinely necessary trade-offs, and which are just one company's particular 1975 circumstances, precisely because you're watching two different teams solve the same problems under different pressures.

A First Preview: RISC vs. CISC

The 6502's minimalism — few registers, a small instruction set, most instructions executing in a small, predictable number of cycles — previews a design philosophy that would later be named **RISC** (Reduced Instruction Set Computing). The Z80's richness — many specialized instructions, more addressing modes, variable-length encodings that pack more capability into fewer lines of assembly — previews what would later be called **CISC** (Complex Instruction Set Computing).

THESE TERMS DIDN'T EXIST YET

Neither MOS Technology nor Zilog set out to build a "RISC chip" or a "CISC chip" — those terms weren't coined until the Berkeley RISC research project around 1980, years after both chips shipped. Calling the 6502 and Z80 early instances of RISC and CISC is a genuinely useful lens for understanding them today, but it's a modern lens applied in hindsight, not a philosophy either design team was consciously building toward in 1975.

This course names that connection here, deliberately early, and comes back to formalize it properly in `cpu8bit1-11` — once both chips have actually been studied in enough depth for the comparison to mean something concrete, rather than just a label.

How This Course Is Structured

To keep the comparison genuinely even-handed, the 6502 and Z80 each get their own uninterrupted three-chapter block, covering identical ground in identical order:

- **Chapters 2–4** — the 6502: design philosophy, registers & the stack, addressing modes
- **Chapters 5–7** — the Z80: design philosophy, registers & the stack, addressing modes
- **Chapter 8** — the same small routine, written and compared in both
- **Chapter 9** — interrupts, revisiting the minimalism-vs-richness contrast one more time
- **Chapter 10** — where these chips actually lived: real machines, real culture
- **Chapter 11** — formalizing the RISC-vs-CISC throughline this chapter only previewed
- **Chapter 12** — a capstone comparing a larger routine across both chips

Hands-On Exercises

EXERCISE 1

Using this chapter's own framing, explain why the 6502's $25 price point (against the 8080's $179) is described as "the founding fact of the whole chip" rather than just an interesting historical detail — what does that suggest about how price constraints might have shaped the chip's actual architecture?

View solution

EXERCISE 2

Federico Faggin helped design the Intel 8080 before designing the Z80 as its compatible superset. Explain how this specific history creates a fundamentally different set of design constraints for the Z80 than the 6502 faced — constraints rooted in compatibility rather than price.

View solution

EXERCISE 3

Using this chapter's own bridge table, pick one row (fetch-decode-execute, registers, addressing modes, or the stack) and explain specifically what stays conceptually identical from assembly1 versus what will genuinely differ once this course starts covering the 6502 and Z80 in depth.

View solution

CHAPTER 1 QUICK REFERENCE

- **6502** (MOS Technology, 1975) — built around an aggressive $25 price target vs. the 8080's $179
- **Z80** (Zilog, 1976, Federico Faggin) — built as a compatible superset of the Intel 8080
- This course builds directly on assembly1's own baseline vocabulary — fetch-decode-execute, registers, addressing modes, the stack — rather than re-teaching it
- The 6502's minimalism previews RISC; the Z80's richness previews CISC — but neither term existed until ~1980, applied here as a useful modern lens, not a period-accurate one
- Chapters 2–4 cover the 6502 in full; Chapters 5–7 cover the Z80 using the identical structure, keeping the comparison even-handed
- The RISC-vs-CISC throughline is only previewed here — it's formalized in cpu8bit1-11

CHAPTER 2 OF 12

The 6502 — Design Philosophy: Minimalism

8-Bit CPUs — 6502/6510 & Z80

Chapter 2 · The 6502 — Design Philosophy: Minimalism

`cpu8bit1-1` named the 6502's $25 price target as "the founding fact of the whole chip." This chapter makes good on that claim — tracing exactly how a cost target turns into an actual, physical engineering constraint, and what MOS Technology chose to build (and not build) as a result.

The Team Behind the Chip

Chuck Peddle and several colleagues had been working at Motorola on the 6800, a capable but expensive processor. When Motorola declined to pursue a radically cheaper version of the idea, Peddle's team left and joined MOS Technology instead, determined to prove a genuinely low-cost, high-volume microprocessor was possible. The 6502, released in 1975, was that proof.

The Real Engineering Cost of Cheap — Die Size and Transistor Count

In 1975, a chip's manufacturing cost was driven heavily by two linked factors: **die size** (how much physical silicon each chip consumes) and **yield** (what fraction of chips cut from a wafer come out defect-free). A smaller die means more chips fit on a single wafer *and* a better yield, since a random manufacturing defect is less likely to land inside a smaller area. Both effects push the same direction: **fewer transistors means a smaller, cheaper, higher-yield chip** — not as an abstract design preference, but as literal manufacturing economics.

Once $25 was the target, transistor count stopped being a minor implementation detail and became the single number the whole design had to be justified against, line by line.

CHIP	YEAR	LAUNCH PRICE	TRANSISTOR COUNT (APPROX.)	GENERAL-PURPOSE REGISTERS
6502	1975	$25	~3,500	A, X, Y (3)
Intel 8080	1974	$179	~6,000	A, B, C, D, E, H, L (7)

Roughly half the transistor budget, roughly half the general-purpose registers — this isn't a coincidence. Fewer registers means less on-chip storage circuitry, directly translating a smaller register file into real, measurable die-area savings.

"Do More With Less" — Concrete Design Choices

Every one of the following, covered in full over the next two chapters, traces back to the same transistor-budget pressure:

- **A small register file** (`cpu8bit1-3`) — one accumulator (A) plus two index registers (X, Y), instead of a larger general-purpose bank.
- **Zero-page addressing** (`cpu8bit1-4`) — a clever addressing trick that gets much of the speed benefit of extra registers without spending transistors on them.
- **A fixed, hardware-managed stack** confined to a single page of memory, rather than a more flexible (and more transistor-hungry) general mechanism.

What "Minimalism" Doesn't Mean

It's tempting to read "fewer transistors" as "less capable" or "a compromised chip" — that's not what happened here. The 6502 still includes everything a real program genuinely needs: arithmetic, memory access, branching, subroutine calls, a stack. Nothing essential was cut. What was cut was redundancy and luxury: extra general-purpose registers a program could often work around, more addressing modes than were strictly necessary, more elaborate (and more expensive) internal machinery than the actual workload demanded.

"Minimalism" here means every transistor included had to earn its place in a tight budget — not that the chip does less than it needs to.

THIS SETS UP CPU8BIT1-4'S OWN PAYOFF

Zero-page addressing, covered in the very next chapter's continuation, is the clearest example of this philosophy in action: a technique that gives 6502 programs something close to the speed and convenience of extra registers, without spending a single additional transistor on a bigger register file. It's the concrete answer to "how do you do more with less" rather than just a slogan.

CHEAP DIDN'T MEAN "LOST THE MARKET"

It's worth being honest about how this story actually played out: the 6502's low cost is a direct reason it ended up inside the Apple II, the Commodore 64 (via its close relative, the 6510), the Atari 2600, and the original Nintendo Entertainment System. A design built around a tight transistor budget didn't just survive in the market — in the home-computer and games-console era specifically, it thrived precisely because that budget let manufacturers build complete, affordable machines around it.

Hands-On Exercises

EXERCISE 1

In your own words, explain the relationship between die size, transistor count, and manufacturing yield described in this chapter, and explain specifically why that relationship — not just a general preference for simplicity — is what made transistor count the 6502 team's central design constraint.

View solution

EXERCISE 2

Using this chapter's own comparison table, contrast the 6502's 3-register file (A, X, Y) against the 8080's 7-register file. Then, using this chapter's "What Minimalism Doesn't Mean" section, explain why having fewer registers doesn't automatically mean the 6502 is less capable of running real programs.

View solution

EXERCISE 3

Using this chapter's own clarification section, explain the difference between "minimalism as a limitation" and "minimalism as an earned design constraint" as it applies specifically to the 6502 — and connect your answer back to cpu8bit1-1's own exercise about the $25 price point being "the founding fact of the whole chip."

View solution

CHAPTER 2 QUICK REFERENCE

- Chuck Peddle's team left Motorola specifically to build a radically cheaper processor — the 6502 was the result
- Smaller die size → cheaper per-chip cost AND better manufacturing yield — two effects pushing the same direction
- 6502: ~3,500 transistors, $25, 3 general-purpose registers (A, X, Y) — roughly half the 8080's budget and register count
- Small register file → zero-page addressing (cpu8bit1-4) → a fixed, single-page hardware stack — all downstream of the same transistor budget
- Minimalism here means every transistor earned its place — not that anything essential (arithmetic, branching, subroutines, a stack) was left out
- The 6502's low cost is a direct reason it powered the Apple II, Commodore 64/6510, Atari 2600, and NES

CHAPTER 3 OF 12

6502 Registers and the Stack

8-Bit CPUs — 6502/6510 & Z80

Chapter 3 · 6502 Registers and the Stack

`cpu8bit1-2` named A, X, and Y as the 6502's general-purpose registers, and left the rest for here. This chapter completes the picture — the special-purpose registers that make the chip actually work (PC, SP, P), a real hardware stack that's genuinely better than `assembly1-7`'s manual LC-3 approach in one way and more limited in another, and the concrete reason the small register file forces the addressing-mode cleverness `cpu8bit1-4` covers next.

The Full 6502 Register Set

REGISTER	WIDTH	PURPOSE	LC-3 EQUIVALENT
A	8-bit	The accumulator — most arithmetic and logic instructions route through it	No direct equivalent — any of LC-3's R0–R7 could serve this role
X, Y	8-bit each	Index registers — indexed addressing (cpu8bit1-4) and loop counters	Loosely comparable to a base register in assembly1-3's base+offset mode
PC	16-bit	Address of the next instruction	The same role as LC-3's own PC (assembly1-2)
SP	8-bit	Stack pointer — real hardware, but locked to one 256-byte page (below)	LC-3 had no dedicated SP at all (assembly1-7) — the 6502 is a genuine improvement here
P	8-bit	Processor status flags — richer than LC-3's N/Z/P (below)	LC-3's N/Z/P (assembly1-4) — the 6502's version adds real capability LC-3 never needed

The Status Register — Richer Than N/Z/P

`assembly1-4`'s N/Z/P condition codes were LC-3's own deliberately simplified teaching version of a real status register. The 6502's actual **P** register packs eight individual flag bits, several of which LC-3 simply never needed:

N	V	-	B	D	I	Z	C

- **N** (Negative) and **Z** (Zero) — direct counterparts to two of LC-3's own three condition codes.
- **C** (Carry) — LC-3 never had one, because LC-3's registers were a full 16 bits wide, matching its own word size, so ordinary addition never needed to signal "this overflowed the register." The 6502's 8-bit registers

make Carry essential — see below.

- **V** (Overflow) — a separate flag specifically for *signed* overflow, distinct from Carry's *unsigned* overflow signal — a distinction LC-3's simplified model never had to draw.
- **D** (Decimal mode) — switches addition/subtraction into binary-coded decimal, useful for real-world calculations (early point-of-sale and calculator-style applications) where exact decimal results matter more than raw binary speed.
- **I** (Interrupt disable) — a real hardware feature LC-3 never modeled at all.
- **B** — only meaningful in the copy of P pushed to the stack during an interrupt, distinguishing a software-triggered `BRK` from a real hardware interrupt.

Why the Carry Flag Matters So Much Here

The 6502 is an 8-bit CPU — its registers and internal data paths handle one byte (0–255) at a time — but its **address bus is 16 bits**, giving it a full 64KB address space. That split matters: any arithmetic on a number bigger than 255 has to be built by hand, one byte at a time, chaining the result of one 8-bit addition into the next using **Carry**:

```
CLC              ; clear Carry before starting a multi-byte add
LDA NUM1_LOW
ADC NUM2_LOW     ; add with carry — low bytes
STA RESULT_LOW
LDA NUM1_HIGH
ADC NUM2_HIGH    ; carry from the low-byte add feeds into this one
STA RESULT_HIGH
```

LC-3's own 16-bit-wide registers meant a value never needed to be split across two registers just to add it — this entire pattern, and the Carry flag it depends on, simply had no reason to exist in `assembly1`. It's a direct, concrete consequence of the 6502 being genuinely 8-bit in a way LC-3, built purely for teaching, never was.

The Stack — Real Hardware, Genuinely Constrained

`assembly1-7` had to build PUSH and POP entirely out of ordinary `ADD` / `STR` / `LDR` instructions, because LC-3 provides no dedicated stack-pointer register at all. The 6502 is a real improvement here: **SP is a genuine, dedicated hardware register**, and the chip provides native push/pull instructions:

```
PHA   ; push the accumulator onto the stack
PLA   ; pull the top of the stack back into the accumulator
PHP   ; push the status register
PLP   ; pull the status register back
```

Compare that single `PHA` against `assembly1-7`'s own multi-instruction manual push sequence — this is real, dedicated hardware doing in one instruction what LC-3 needed three for.

The constraint: SP is only **8 bits** wide, not 16. It doesn't hold a full address — it holds just the low byte of one, and the high byte is permanently fixed to `$01` by the hardware. This locks the entire stack to a single 256-byte region, **page 1** (addresses `$0100`–`$01FF`), no matter what. The stack grows downward — SP decrements on every push, increments on every pull — the same downward-growing convention `assembly1-7`'s own example already used.

> **NOTHING STOPS A STACK OVERFLOW**
>
> Because SP is just an ordinary 8-bit register being decremented and incremented, there's no hardware check preventing it from wrapping around. Push too many values without popping them and SP will silently wrap from `$00` back to `$FF`, overwriting whatever was already sitting at the top of page 1 — with no error, no warning, just corrupted data. This is the same "the CPU doesn't stop you" theme `assembly1` raised more than once, just showing up here as a real, documented 6502 hazard instead of a hypothetical one.

Why So Few Registers Forced Clever Addressing

Three general-purpose registers — A, X, Y — is genuinely little working space. Adding more wasn't an option without breaking `cpu8bit1-2`'s own transistor budget. Instead, the 6502's designers found a different way to get fast, flexible access to values without paying for more physical registers: a special, quick-access region of memory that instructions can reach with shorter encodings than ordinary memory access requires. `cpu8bit1-4` covers this technique — **zero-page addressing** — in full, as the chip's real answer to "we can't afford more registers."

> **TXS/TSX — SP IS SET INDIRECTLY, THROUGH X**
>
> You can't load an immediate value straight into SP the way you can into A, X, or Y. Setting up the stack pointer goes through X first: load the desired value into X, then use `TXS` (Transfer X to SP) to move it into place. Reading SP back out works the same way in reverse, via `TSX`. It's a small, real quirk — one more example of a register that isn't quite as general-purpose as it might first appear.

Hands-On Exercises

EXERCISE 1

Using this chapter's own reasoning, explain why LC-3 never needed a Carry flag, but the 6502 genuinely does. Tie your answer to the specific difference in register width between the two architectures.

View solution

EXERCISE 2

Compare the 6502's stack to LC-3's own stack from assembly1-7: name one specific way the 6502's is a genuine improvement, and one specific way it's more constrained. Use this chapter's own PHA/PLA example and the page-1 limitation in your answer.

View solution

EXERCISE 3

A program pushes 260 values onto the 6502 stack without ever popping any of them, starting from SP's default reset value of $FF. Using this chapter's own warn-box, explain what happens to SP and to the data sitting in page 1 as a result — and why the hardware never stops it.

View solution

CHAPTER 3 QUICK REFERENCE

- **A** — the accumulator · **X, Y** — index registers · **PC** — 16-bit instruction pointer · **SP** — 8-bit stack pointer · **P** — 8-bit status flags (N V - B D I Z C)
- Carry (C) exists because the 6502 is genuinely 8-bit — multi-byte math is chained by hand across ADC instructions, something LC-3's full 16-bit registers never required
- Overflow (V) is a separate signed-overflow flag, distinct from Carry's unsigned signal
- Decimal mode (D) and Interrupt disable (I) have no LC-3 equivalent at all
- PHA/PLA/PHP/PLP — real, dedicated hardware push/pull, a genuine improvement over assembly1-7's manual LC-3 approach
- SP is only 8 bits — the stack is permanently locked to page 1 ($0100–$01FF), with no hardware protection against overflow/wraparound
- SP can't be loaded directly — set it via X and TXS
- Only 3 general-purpose registers is exactly why zero-page addressing (cpu8bit1-4) exists

CHAPTER 4 OF 12

6502 Addressing Modes — Zero Page as "Extra Registers"

8-Bit CPUs — 6502/6510 & Z80

Chapter 4 · 6502 Addressing Modes — Zero Page as "Extra Registers"

`cpu8bit1-3` closed with a promise: with only three general-purpose registers, the 6502 needed another way to get fast, flexible access to values. This chapter delivers on it — **zero page**, the chip's real, concrete answer to a transistor budget that couldn't afford more registers, plus the rest of the addressing modes that make real 6502 programs practical.

Zero Page — The Chip's Real Answer

Addresses `$0000`–`$00FF` — the first 256 bytes of memory — get special treatment on the 6502. Because every address in that range fits in a single byte, an instruction accessing zero page only needs to encode **one address byte** instead of two. That shorter encoding means less to fetch *and* fewer internal cycles to compute the effective address — zero page is measurably faster than ordinary memory access, not just shorter to type.

The practical effect: with 256 available slots, zero page behaves like a bank of pseudo-registers, vastly larger than the 3 real ones (A, X, Y) the chip actually has — at a real cost (an extra memory access instead of a truly free register read), but a far smaller cost than physical registers would have added to the transistor budget `cpu8bit1-2` already established as the chip's central constraint.

Absolute Addressing — Reaching the Full 64KB

Absolute addressing embeds a full, two-byte address directly in the instruction, reaching anywhere across the 6502's entire 64KB space. This is worth contrasting directly with `assembly1-3`'s own PC-relative `LD`: LC-3 had to compute an address from an offset specifically because its instructions were a fixed 16 bits wide, with no room left over for a full standalone address. The 6502 doesn't have that problem — its instructions are **variable length** (1 to 3 bytes), so a full 2-byte address simply costs one extra byte in the instruction stream, not a fight for bits within a single fixed-width word.

```
LDA $0050    ; zero page — 2 bytes, address fits in one byte
LDA $1050    ; absolute — 3 bytes, a full two-byte address
```

Indexed Addressing — Zero Page,X and Absolute,X/Y

Adding X or Y to a base address turns either mode into an **indexed** one — the mechanism behind iterating over an array, with the index register playing a role similar to `assembly1-3`'s own base+offset `LDR`, except the "offset" here is a whole register that changes every loop iteration rather than a fixed 6-bit constant baked into the instruction.

```
LDA $0050,X   ; zero page,X — base $0050 plus X
LDA $1050,X   ; absolute,X — base $1050 plus X, reaches the full 64KB
```

ZERO PAGE,X WRAPS — IT NEVER SPILLS INTO PAGE 1

`LDA $FF,X` with X = `$02` does **not** read from `$0101`. The addition wraps *within* zero page itself, landing at `$01` — the high byte of the address stays permanently zero no matter how far the addition would otherwise carry. Absolute,X has no such limit; only the zero-page indexed modes wrap this way. It's a real, well-documented 6502 quirk, and a genuine source of bugs for programmers who assume it behaves like absolute,X.

(Zero Page),Y — Indirect-Indexed: A Real Pointer Dereference

This is the 6502's most powerful addressing mode, and the direct answer to a problem `assembly1-3`'s own indirect mode (`LDI`/`STI`) never fully solved: accessing an array whose starting address isn't known until the program actually runs.

A zero-page location holds a full 16-bit **pointer** — the low byte at the zero-page address itself, the high byte at the next one over. `(zp),Y` reads that pointer, then adds Y to the pointer's *value* to get the final effective address:

```
; zero page $10/$11 holds a pointer to $3000
LDA #$00
STA $10        ; low byte of the pointer
LDA #$30
STA $11        ; high byte of the pointer — together, $10/$11 hold $3000

LDY #$00
LDA ($10),Y    ; reads from $3000 + Y — a real runtime pointer dereference
```

This is genuinely more capable than LC-3's own `LDI`, which could follow a pointer but never combine that with an index in the same instruction. `(zp),Y` does both at once — exactly what's needed to walk through an array whose address is only known at runtime.

A related, less commonly needed mode, **(zero page,X)** — indexed-indirect — works the other way around: X is added to the zero-page address *first* (wrapping within zero page, the same way `zp,X` does), and whatever pointer sits at the resulting location is used directly, with no further offset. It shows up most often selecting between several fixed pointers — a jump table, for instance — rather than walking through an array.

Real Cycle-Cost Comparison

MODE	BYTES	CYCLES (LDA)	WHEN TO REACH FOR IT
Immediate	2	2	A known constant
Zero page	2	3	A frequently-used variable — the "extra register" case
Zero page,X	2	4	An array indexed by X, with its base in zero page
Absolute	3	4	A variable located anywhere in the 64KB space
Absolute,X / ,Y	3	4 (+1 on page crossing)	An array indexed by X/Y, with its base anywhere in memory
(Zero page),Y	2	5 (+1 on page crossing)	Dereferencing a runtime pointer, then indexing into it
(Zero page,X)	2	6	Selecting between several fixed pointers — jump tables

Zero page's own 3-cycle cost against absolute's 4 is the concrete, measurable version of "extra registers" this chapter opened with — a real, repeatable speed advantage for any value accessed often enough to be worth keeping in the first 256 bytes.

ASSEMBLERS PICK ZERO PAGE FOR YOU AUTOMATICALLY

A real 6502 assembler doesn't require choosing between `LDA $0050` and a longer absolute form manually — if the target address fits in a single byte, it assembles as zero page by default, since there's essentially never a reason not to take the faster, shorter encoding when it's available. The two forms in this chapter's own absolute-addressing example look almost identical in source code, but assemble to genuinely different instructions underneath.

Hands-On Exercises

EXERCISE 1

Using this chapter's own tip-box, explain why `LDA $0050` assembles as a 2-byte, 3-cycle zero-page instruction while `LDA $1050` assembles as a 3-byte, 4-cycle absolute instruction, even though both look like a similar "load from an address" instruction in source code.

View solution

EXERCISE 2

Zero page address $10 holds the byte $00, and $11 holds the byte $30. Y currently holds $05. Trace this chapter's own (zp),Y formula to determine the effective address of `LDA ($10),Y`, and state what that address actually is.

View solution

EXERCISE 3

X holds $02. Using this chapter's own warn-box, determine exactly where `LDA $FF,X` actually reads from, and explain why a programmer expecting it to behave like absolute,X would be wrong.

View solution

CHAPTER 4 QUICK REFERENCE

- **Zero page** ($0000–$00FF) — shorter encoding, faster access, the chip's real substitute for more physical registers
- **Absolute** — a full 2-byte address embedded directly, reaching the whole 64KB space, possible because 6502 instructions are variable-length
- **Zero page,X / absolute,X,Y** — indexed addressing for arrays; zero page,X wraps within page zero, absolute,X does not
- **(Zero page),Y** — indirect-indexed: dereference a runtime pointer stored in zero page, then add Y — genuinely more capable than LC-3's own LDI
- **(Zero page,X)** — indexed-indirect: index into zero page first, then follow the pointer found there — used for jump-table-style selection
- Zero page (3 cycles) beats absolute (4 cycles) for the same LDA — a real, measurable "extra register" speed advantage
- Real assemblers automatically choose zero-page encoding whenever the target address fits in one byte

CHAPTER 5 OF 12

The Z80 — Design Philosophy: Richness and Compatibility

8-Bit CPUs — 6502/6510 & Z80

Chapter 5 · The Z80 — Design Philosophy: Richness and Compatibility

Chapters 2–4 covered the 6502 end to end. This chapter opens an identical three-chapter arc for the **Z80** — design philosophy first, exactly as `cpu8bit1-2` did — so the two chips stay genuinely comparable rather than lopsided. And the contrast starts immediately: where the 6502 was built by a team that left one company to chase a price target from a clean slate, the Z80 was built by a designer extending his *own* earlier work, under a completely different kind of constraint.

The Team Behind the Chip — Federico Faggin's Break From Intel

Federico Faggin had already helped design the Intel 4004, 8008, and — directly relevant here — the **8080** itself, before leaving Intel in 1974 to found Zilog alongside Ralph Ungermann. The Z80, released in 1976, wasn't Faggin encountering the 8080 as an outside competitor the way MOS Technology approached the market — it was its own original co-designer building a deliberate successor to it.

Built to Be a Superset — What "Compatible" Actually Meant

`cpu8bit1-1`'s own Exercise 2 already touched this at a high level; here's the full picture. The Z80 was designed so that **any existing 8080 machine code would run on it completely unchanged** — while adding a substantial layer of new registers, instructions, and addressing modes on top. This produces a fundamentally different starting position than the 6502's own:

- The 6502 team could freely decide "we don't need this" — nothing existing constrained them, since they were designing from a genuine blank slate.
- The Z80 team could only **add**. Every instruction and register the 8080 already had needed to keep working exactly as before, whether or not it was still the cleanest way to do something — new capability had to be layered on top of that existing foundation, never used to replace or simplify it.

The CISC-Leaning Richness — Concrete Consequences

That compatibility mandate shows up as real, measurable richness once the 8080's own instruction set becomes the Z80's *starting point* rather than its ceiling:

- **More instructions** — the 8080 offered roughly 78 instruction types; the Z80, built as its superset, is commonly cited as offering around 158 — roughly double, all while remaining fully compatible with the original 78.
- **More registers** — a genuine shadow register set and dedicated index registers, both entirely new, covered in full in `cpu8bit1-6`.
- **More addressing modes** — covered in `cpu8bit1-7`.

Fitting roughly double the instructions into a design that still had to decode the 8080's original 256-opcode space required a real engineering trick: the Z80 reserves four special **prefix bytes** — `CB`, `DD`, `ED`, and `FD` — each one signaling "the next byte selects from an entirely separate table of additional instructions." It's a genuinely clever escape hatch, extending the instruction space far beyond what a single unprefixed byte could ever address — engineering cleverness aimed squarely at capability, the mirror image of the 6502's own cleverness aimed at cost.

Contrasted Directly With the 6502's Minimalism

	6502 (CPU8BIT1-2)	Z80 (THIS CHAPTER)
Year	1975	1976
Designer	Chuck Peddle & team (left Motorola)	Federico Faggin & team (left Intel)
Founding constraint	Price — a $25 target from a clean slate	Compatibility — a mandatory 8080-compatible superset
Transistor count (approx.)	~3,500	~8,500
Instruction count (approx.)	56 mnemonics	~158 mnemonics
Governing philosophy	Cut everything non-essential	Keep everything, add more on top

Notice the transistor count moves in exactly the direction each chip's own founding constraint predicts: the 6502's cost target pushed it down, the Z80's added capability pushed it up — roughly double the 6502's own budget. Both chips are still answering the same underlying question `cpu8bit1-2` raised — what does the transistor budget actually buy you — just pointed in opposite directions by two genuinely different starting pressures.

> **RICHNESS ISN'T "BETTER" ANY MORE THAN MINIMALISM WAS**
>
> It would be easy to read "twice as many instructions" as a straightforward win. `cpu8bit1-10` covers where each chip actually ended up commercially, and the real story doesn't favor either philosophy outright — the Z80 powered the ZX Spectrum, MSX machines, the original Game Boy, and a huge share of CP/M-based business computers,

succeeding on very different terms than the 6502's own Apple II/Commodore/NES/Atari lineage. Two philosophies, two real, separate commercial success stories.

"CISC" IS A HINDSIGHT LABEL HERE TOO

Exactly as `cpu8bit1-1` flagged for "RISC," nobody at Zilog set out in 1976 to build "a CISC chip" — that term, like its counterpart, wasn't coined until around 1980. Calling the Z80's richness an early instance of CISC is a useful modern lens for understanding it, not a philosophy Faggin's team was consciously naming at the time. It's also worth being honest that "richer" doesn't automatically mean "every instruction is equally well-designed or fast" — some of that nuance is exactly what `cpu8bit1-8` 's own side-by-side routine comparison will surface concretely.

Hands-On Exercises

EXERCISE 1

Federico Faggin co-designed the Intel 8080 before founding Zilog and building the Z80 as its compatible superset. Using this chapter's own framing, explain how this specific history is different from simply "a competitor built a rival chip" — what did Faggin's prior role on the 8080 itself mean for how the Z80 was designed?

View solution

EXERCISE 2

Using this chapter's own comparison table, explain why the Z80's transistor count (~8,500) is roughly double the 6502's (~3,500), tying your answer to each chip's own founding constraint rather than just stating that the Z80 "has more stuff."

View solution

EXERCISE 3

Explain, using this chapter's own description of the CB/DD/ED/FD prefix bytes, how the Z80 manages to offer roughly double the 8080's instruction count despite starting from the same 256-value single-byte opcode space the 8080 itself used.

View solution

CHAPTER 5 QUICK REFERENCE

- Federico Faggin co-designed the Intel 8080, then left to found Zilog and build the Z80 as its deliberate, compatible successor (1976)

- The Z80 runs all existing 8080 machine code unchanged, while adding new registers/instructions/addressing modes on top
- Compatibility only allows ADDING capability, never removing or simplifying existing behavior — the opposite constraint from the 6502's clean-slate cost target
- ~158 Z80 instruction mnemonics vs. the 8080's ~78 and the 6502's 56 — roughly double the 8080's own count
- CB/DD/ED/FD prefix bytes are the real mechanism letting the Z80 exceed the 8080's original 256-opcode ceiling
- ~8,500 transistors vs. the 6502's ~3,500 — richness costs real silicon, the mirror image of the 6502's own cost-driven minimalism
- "CISC" is a retroactive label, same as "RISC" from cpu8bit1-1 — neither term existed when either chip actually shipped

CHAPTER 6 OF 12

Z80 Registers — Main, Shadow, and Index

8-Bit CPUs — 6502/6510 & Z80

Chapter 6 · Z80 Registers — Main, Shadow, and Index

`cpu8bit1-5` described the Z80's richness in the abstract — more instructions, more registers, more addressing modes. This chapter makes the register half of that claim concrete, following the exact structure `cpu8bit1-3` used for the 6502, so the comparison stays direct and even-handed.

The Main Register Set — A, F, and the BC/DE/HL Pairs

Like the 6502, the Z80 has an accumulator (**A**) and a flags register (**F**) — but from there the two chips diverge sharply. Instead of two more single-purpose 8-bit registers (the 6502's X and Y), the Z80 provides **six** general-purpose 8-bit registers — B, C, D, E, H, L — and, crucially, they can be used either individually or combined into three genuine **16-bit register pairs**: BC, DE, and HL.

This pairing is something the 6502 simply has no equivalent for at all — X and Y stay strictly 8-bit, always. The Z80's pairs make native 16-bit arithmetic possible directly:

```
LD  HL, $3000
LD  BC, $0005
ADD HL, BC     ; a single instruction adds two full 16-bit values
```

Compare that against `cpu8bit1-3`'s own multi-byte addition example, which needed two separate 8-bit `ADC` instructions chained together through Carry because the 6502 has no native 16-bit arithmetic at all.

HL as a Pointer — Simpler Than Anything the 6502 Offers

HL in particular is routinely used as a direct memory pointer:

```
LD  HL, $3000
LD  A, (HL)     ; reads the byte AT the address held in HL — one register, one instruction
```

This is genuinely simpler than `cpu8bit1-4`'s own `(zp),Y` — which required a pointer stored specifically in zero page, plus a separate index register, plus a two-step lookup. On the Z80, the pointer just lives directly in a 16-bit register; no zero-page indirection is needed at all.

The Shadow Register Set — A Genuinely Unique Feature

Here's where the Z80 does something neither the 6502 nor LC-3 offers any equivalent for: a complete **second copy** of A, F, and the BC/DE/HL pairs — the **shadow** (or alternate) register set, written A′, F′, B′C′D′E′H′L′. Only one set is active at any moment, and two special instructions swap between them:

- `EXX` — swaps the general-purpose BC/DE/HL with their shadow counterparts, all at once, in a single instruction.
- `EX AF,AF'` — swaps A and F with their own shadow versions, separately from EXX.

The historical, documented purpose: fast interrupt handling. An interrupt routine could execute `EXX` to instantly gain an entirely fresh set of working registers, do whatever it needed to do, then `EXX` back — without ever needing to push the main registers onto the stack to protect them, and without ever needing to pop them back afterward. In practice, plenty of real Z80 programs (including software on the ZX Spectrum) simply used the shadow set as extra general-purpose storage rather than reserving it strictly for interrupts, since nothing in the hardware enforces how it's used — but the interrupt-speed use case is what motivated building it in the first place.

Index Registers — IX and IY

Two additional 16-bit registers, **IX** and **IY**, exist specifically to support indexed addressing — full treatment in `cpu8bit1-7`. A quick preview: `LD A,(IX+5)` reads the byte at whatever address IX holds, plus 5. These are exactly the registers behind two of `cpu8bit1-5`'s own prefix bytes — IX-indexed instructions are signaled by the `DD` prefix, IY-indexed ones by `FD`. What was an abstract encoding mechanism in that chapter now has a concrete, working purpose.

Contrasted Against the 6502's Minimal Register File

	6502 (CPU8BIT1-3)	Z80 (THIS CHAPTER)
8-bit general-purpose registers	3 (A, X, Y)	7 in the main set (A, B, C, D, E, H, L) — doubled to 14 counting the full shadow set
Native 16-bit register pairs	None	BC, DE, HL — plus IX, IY, plus their own shadow-set counterparts
Alternate/shadow register set	None at all	A complete second copy of A/F/BC/DE/HL, swapped via EXX / EX AF,AF′
Direct-register memory pointer	None — memory pointers always route through zero page (cpu8bit1-4)	HL holds a pointer directly, no zero-page step required

TWO THREADS TO PICK BACK UP LATER

`cpu8bit1-7` covers IX/IY-based addressing in full, building directly on this chapter's own preview. And `cpu8bit1-9` revisits the shadow register set's real interrupt-handling payoff in depth — directly extending `assembly1-8`'s own polling-vs-interrupts preview into a concrete hardware mechanism LC-3 never had access to at all.

ALL THIS REGISTER CAPABILITY ISN'T FREE — AND IT'S ALL-OR-NOTHING PER GROUP

Every register in this chapter is part of the reason the Z80 needed roughly 8,500 transistors against the 6502's own ~3,500, per `cpu8bit1-5`'s own comparison table — richness in registers is a direct, real silicon cost, the same way minimalism was a direct silicon saving for the 6502. It's also worth noting a real hardware constraint: `EXX` and `EX AF,AF'` swap their *entire* register group at once — there's no way to selectively swap just one register out of BC/DE/HL while leaving the others in place.

Hands-On Exercises

EXERCISE 1

Using this chapter's own HL example and cpu8bit1-4's own (zp),Y explanation, explain specifically why dereferencing a pointer through HL is a simpler operation than the 6502's own indirect-indexed addressing — name what step the 6502 needs that the Z80 doesn't.

View solution

EXERCISE 2

Explain the real, historical purpose of the Z80's shadow register set, and explain specifically why neither the 6502 nor LC-3 (assembly1-8's own TRAP-based I/O) has anything comparable to offer for the same problem.

View solution

EXERCISE 3

Using this chapter's own compare table and cpu8bit1-5's transistor-count numbers, explain why the Z80 having roughly 14 total 8-bit-equivalent registers (counting both the main and shadow sets) against the 6502's 3 is consistent with — not a coincidence alongside — the two chips' own transistor budgets.

View solution

CHAPTER 6 QUICK REFERENCE

- **Main set** — A, F, and six general-purpose registers (B, C, D, E, H, L), individually 8-bit or paired into 16-bit BC/DE/HL

- BC/DE/HL enable native 16-bit arithmetic (e.g. `ADD HL,BC`) — something the 6502 can only do by hand-chaining 8-bit ADC through Carry
- **HL as a pointer** — `LD A,(HL)` dereferences directly, no zero-page indirection needed, unlike the 6502's own (zp),Y
- **Shadow set** — a full second copy of A/F/BC/DE/HL, swapped via `EXX` / `EX AF,AF'`, built for fast interrupt handling — no 6502 or LC-3 equivalent at all
- **IX, IY** — dedicated 16-bit index registers behind the DD/FD prefix bytes from cpu8bit1-5, full addressing coverage in cpu8bit1-7
- EXX/EX AF,AF′ swap their entire register group at once — no selective single-register swap is possible
- This register-level richness is a direct, real contributor to the Z80's own ~8,500-transistor budget from cpu8bit1-5

CHAPTER 7 OF 12

Z80 Addressing Modes and the Stack

8-Bit CPUs — 6502/6510 & Z80

Chapter 7 · Z80 Addressing Modes and the Stack

This closes out the Z80 block. `cpu8bit1-6` previewed IX/IY and HL-as-pointer; this chapter delivers the full addressing-mode picture, including the real cost the Z80's flexibility carries — and finishes with the stack, where the Z80 turns out to be a genuine synthesis of what LC-3 and the 6502 each did separately.

Register-Indirect Addressing — (HL), (BC), (DE)

`cpu8bit1-6` already introduced `(HL)` as a direct pointer dereference. `(BC)` and `(DE)` work identically in principle — reading or writing whatever address the register pair currently holds — though in practice they're supported by a narrower set of instructions (mainly simple loads and stores), while HL remains the chip's general-purpose pointer workhorse, usable with far more of the instruction set. All three are simpler than `assembly1-3`'s own LC-3 indirect mode (`LDI`), which required a two-step "read a pointer from a PC-relative address, then follow it" — here, the pointer already sits directly in a register, one step only.

Indexed Addressing With Displacement — IX+d and IY+d

`IX` and `IY` support a genuinely powerful addressing mode: a base register plus a signed 8-bit displacement (`d`, roughly -128 to +127), computed at execution time:

```
LD IX, $4000
LD A, (IX+5)   ; reads the byte at $4000 + 5 = $4005
```

This is a natural fit for accessing individual fields of a data structure at fixed offsets from a base pointer — IX holding the structure's address, `d` selecting which field — and unlike the 6502's own indexed modes from `cpu8bit1-4`, it isn't tied to zero page at all. IX and IY can point *anywhere* in the full 64KB space.

That flexibility has a real, measurable cost. Recall `cpu8bit1-4`'s own cycle table for the 6502's addressing modes:

CHIP	INSTRUCTION	CYCLES	NOTES
Z80	`LD A,(HL)`	7	Direct register-pointer dereference

CHIP	INSTRUCTION	CYCLES	NOTES
Z80	`LD A,(IX+d)`	19	Prefix byte + opcode + displacement byte, plus real address-calculation time
6502	`LDA` zero page	3	cpu8bit1-4
6502	`LDA (zp),Y`	5	cpu8bit1-4 — the closest 6502 equivalent to a flexible pointer-plus-offset read

`LD A,(IX+d)` costs nearly four times what `(HL)` costs, and almost four times what the 6502's own `(zp),Y` costs for a broadly comparable "pointer plus offset" access. The Z80's version can reach anywhere in memory rather than needing a zero-page base — but that extra reach is paid for in real, per-instruction cycles, not just extra transistors.

> **RICHNESS HAS A PRICE, AND IT ISN'T ONLY TRANSISTORS**
>
> `cpu8bit1-5` and `cpu8bit1-6` both measured the Z80's richness in transistor count. This is the same story showing up at *runtime* instead of at manufacturing time — IX/IY addressing genuinely does more, but every use of it costs real, measurable extra cycles compared to the chip's own simpler modes. A capability existing isn't the same as it being free to use.

The Stack — Real Hardware, Genuinely Flexible

`cpu8bit1-3` covered the 6502's real but page-1-locked hardware stack, itself a genuine improvement over `assembly1-7`'s entirely software-built LC-3 stack. The Z80's stack is a third point on that same spectrum: SP is a full **16-bit** register, free to point *anywhere* in the 64KB address space — no fixed page, no artificial ceiling.

Push and pop are native instructions, and — reflecting `cpu8bit1-6`'s own register pairing — they operate on a full 16-bit register pair at once, not a single byte the way the 6502's `PHA` does:

```
PUSH BC   ; pushes both B and C together, 16 bits in one instruction
POP  BC   ; pulls both back
```

The stack still grows downward — SP decrements on every push — the same convention `assembly1-7`'s own LC-3 example and `cpu8bit1-3`'s 6502 stack both already used.

ARCHITECTURE	WHERE THE STACK CAN LIVE	HARDWARE SUPPORT	PUSH/POP GRANULARITY
LC-3 (assembly1-7)	Anywhere in memory — the programmer's own choice	None — manually built from ADD/STR/LDR	One value at a time, via generic instructions
6502 (cpu8bit1-3)	Locked to page 1 only ($0100–$01FF)	Real — dedicated SP register, PHA/PLA/PHP/PLP	One byte at a time
Z80 (this chapter)	Anywhere in the full 64KB space	Real — dedicated 16-bit SP register, PUSH/POP	One 16-bit register pair at a time

Put side by side, the Z80's stack genuinely combines the best of both predecessors covered so far: LC-3's freedom to live anywhere, with real dedicated hardware doing the actual work — the flexibility of a software stack with the speed of a hardware one.

NEXT: PUTTING ALL OF THIS TO WORK

Every addressing mode and stack mechanism from both this chapter and Chapters 2–4 gets used for real in `cpu8bit1-8`, which implements one identical routine in both 6502 and Z80 assembly, side by side — the course's first direct, concrete payoff of studying the two chips in parallel rather than in isolation.

Hands-On Exercises

EXERCISE 1

Given IX = $4000, compute the effective address of `LD A,(IX+5)` using this chapter's own formula, and show your work.

View solution

EXERCISE 2

Using this chapter's own cycle table, explain why `LD A,(IX+d)` (19 cycles) costs almost four times as much as the 6502's own `LDA (zp),Y` (5 cycles), even though both are broadly "flexible pointer plus offset" addressing modes.

View solution

EXERCISE 3

Using this chapter's own three-way stack comparison table, explain what each of LC-3, the 6502, and the Z80 trades away in exchange for its own particular combination of flexibility and hardware support.

View solution

CHAPTER 7 QUICK REFERENCE

- **(HL)/(BC)/(DE)** — direct register-pointer dereference, simpler than LC-3's own two-step LDI
- **IX+d / IY+d** — a base register plus a signed 8-bit displacement, reaching anywhere in the full 64KB space
- `LD A,(IX+d)` costs 19 cycles vs. `(HL)`'s 7 and the 6502's own (zp),Y at 5 — flexibility has a real runtime cost, not just a transistor one
- The Z80's SP is a full 16-bit register — the stack can live anywhere in memory, with no page restriction

- **PUSH/POP** move a full 16-bit register pair in one instruction, vs. the 6502's single-byte PHA/PLA
- Three-way stack comparison: LC-3 (flexible, software-only) → 6502 (hardware, page-locked) → Z80 (hardware AND flexible)
- This closes the Z80 block — cpu8bit1-8 puts every addressing mode and stack mechanic from both chips to work side by side

CHAPTER 8 OF 12

Writing the Same Program in Both

8-Bit CPUs — 6502/6510 & Z80

Chapter 8 · Writing the Same Program in Both

Chapters 2–7 studied each chip in isolation. This chapter puts them side by side for the first time — one identical task, written twice, so every difference on the page is a real difference in the chips themselves, not an accident of two separate examples. The task: sum five bytes stored in an array into a single running total.

The 6502 Version

`ARRAY` and `SUM` are both placed in zero page (`cpu8bit1-4`), so `ARRAY,X` below is genuinely zero-page,X indexed addressing, not the more expensive absolute,X form:

```
        LDX #0          ; X = array index = 0
        LDA #0          ; A = 0
        STA SUM         ; SUM = 0
LOOP    LDA SUM
        CLC             ; NEW — clear Carry before ADC (see warn-box below)
        ADC ARRAY,X     ; SUM += ARRAY[X] — zero-page,X (cpu8bit1-4)
        STA SUM
        INX             ; NEW — increment X (X++)
        CPX #5          ; NEW — compare X against 5, setting flags
        BNE LOOP        ; NEW — branch if the comparison found "not equal"
DONE    JMP DONE        ; the 6502 has no HALT — loop on itself forever

ARRAY   .BYTE 10,20,30,40,50    ; zero page, e.g. $10
SUM     .BYTE 0                 ; zero page, e.g. $20
```

Three genuinely new instructions here: **INX** (increment X), **CPX** (compare X against a value, setting the same flags `cpu8bit1-3` already introduced), and **BNE** (branch if the Z flag is clear — "not equal"). Notice the loop needs *three separate instructions* — `INX`, `CPX #5`, `BNE LOOP` — just to advance and test the loop counter.

FORGETTING CLC BEFORE ADC IS A CLASSIC 6502 BUG

`ADC` always folds the current Carry flag into its addition, per `cpu8bit1-3`'s own Carry explanation. If Carry happens to be set from something earlier in the program, an unguarded `ADC` silently adds one extra — a famous, well-documented source of off-by-one bugs in real 6502 code. `CLC` immediately before `ADC` (when you want ordinary, uncontaminated addition) is close to a reflex among 6502 programmers.

The Z80 Version

```
        LD HL, ARRAY  ; HL = pointer to the array's start (cpu8bit1-6)
        LD B, 5       ; B = remaining count = 5
        LD A, 0       ; A = 0
LOOP    ADD A, (HL)   ; A += the byte HL points to — register-indirect (cpu8bit1-7)
        INC HL        ; advance the pointer to the next byte
        DJNZ LOOP     ; NEW — decrement B, loop if B != 0, all in ONE instruction
        LD (SUM), A   ; store the final sum
DONE    JR DONE       ; the Z80 has no HALT either — loop on itself

ARRAY   DEFB 10,20,30,40,50
SUM     DEFB 0
```

One genuinely new instruction: **DJNZ** ("Decrement and Jump if Not Zero") — decrements B and branches back to the target in a *single instruction* if the result isn't zero, falling straight through if it is. It's a famous, well-known Z80 instruction, and this routine is exactly the pattern it exists for.

The Comparison

This is the concrete payoff `cpu8bit1-5` and `cpu8bit1-6` only described in the abstract. The 6502 needs three separate instructions (`INX`, `CPX #5`, `BNE`) to manage its loop counter every single iteration; the Z80's `DJNZ` folds decrement-compare-and-branch into one. That's the "richness" from `cpu8bit1-5`, made visible in real code rather than described as a slogan.

	6502	Z80
Instructions in the loop body	7 (LDA, CLC, ADC, STA, INX, CPX, BNE)	3 (ADD, INC, DJNZ)
Cycles per iteration (approx.)	19	26

Fewer instructions on the Z80 side — but *more* total cycles per iteration, not fewer. Richness bought real code-density here, not raw speed.

RAW CYCLE COUNTS ALONE CAN MISLEAD

Cycle counts only measure work done *per clock tick* — they say nothing about how fast that clock actually ticks, and the two chips didn't run at the same speed in real machines. A Z80 running at a higher clock speed than a 6502 in a comparable home computer could easily finish this loop in less real, wall-clock time despite needing more cycles per iteration to do it. Comparing cycle counts alone, without knowing each chip's actual clock speed in the machine it's running on, is comparing two different units and reading them as if they were one.

TWO THREADS STILL AHEAD

`cpu8bit1-9` returns to this same minimalism-vs-richness contrast one more time, applied specifically to interrupt handling — including the shadow register set's own real payoff from `cpu8bit1-6`. `cpu8bit1-11` is where this chapter's own concrete evidence gets formally tied back into the RISC-vs-CISC preview `cpu8bit1-1` opened the whole course with.

Hands-On Exercises

EXERCISE 1

Trace the 6502 version's loop for X = 0 through 4. After the fifth pass through the loop body, X becomes 5. Identify exactly which single instruction is responsible for actually ending the loop at that point, and explain what condition it's checking.

View solution

EXERCISE 2

Explain, in your own words, exactly what three separate jobs DJNZ combines into one instruction, and what specific consequence that has for the "7 instructions vs. 3 instructions" comparison this chapter draws between the two loop bodies.

View solution

EXERCISE 3

Using this chapter's own cycle counts (19 for the 6502, 26 for the Z80) and the real historical clock speeds of a 6502 running at roughly 1MHz and a Z80 running at roughly 3.5MHz, calculate the real wall-clock time each chip takes to complete one loop iteration, and state which one is actually faster in practice.

View solution

CHAPTER 8 QUICK REFERENCE

- Same task (sum 5 bytes) implemented on both chips, using only addressing modes and registers already covered in Chapters 2–7
- New 6502 instructions: **INX** (increment X), **CPX** (compare X), **BNE** (branch if not equal) — three instructions to manage one loop counter
- New Z80 instruction: **DJNZ** — decrement B and branch, all in one instruction
- Loop body: 7 instructions / 19 cycles (6502) vs. 3 instructions / 26 cycles (Z80) — fewer instructions didn't mean fewer cycles here

- Cycle counts alone can't tell you which chip is actually faster in real time — clock speed matters too, and the two chips didn't run at the same speed
- Always CLC before ADC on the 6502 unless you specifically want the current Carry folded into the addition — a classic, well-known bug source
- Neither chip has a dedicated HALT — both loop on themselves (JMP DONE / JR DONE) to stop, unlike LC-3's own TRAP-based HALT (assembly1-8)

CHAPTER 9 OF 12

Interrupts — NMI/IRQ/BRK vs. IM 0/1/2

8-Bit CPUs — 6502/6510 & Z80

Chapter 9 · Interrupts — NMI/IRQ/BRK vs. IM 0/1/2

`assembly1-8` named interrupts only in passing — hardware signaling the CPU instead of the CPU repeatedly polling — and deliberately left the mechanism for later, more advanced material. This chapter is that material, on two real chips, and it's also where `cpu8bit1-6`'s own shadow register preview finally gets to pay off for real.

The 6502's Interrupt Model — NMI, IRQ, BRK

The 6502 keeps interrupts as simple as everything else about it:

- **IRQ** (Interrupt Request) — triggered by external hardware wanting attention. *Maskable*: the I flag in the status register (`cpu8bit1-3`'s own P register) can disable it entirely.
- **NMI** (Non-Maskable Interrupt) — reserved for events too critical to ever ignore. Cannot be disabled by any flag, and has its own dedicated vector.
- **BRK** — not a hardware event at all, but a real opcode a program can execute deliberately, triggering interrupt-like behavior in software. Historically used for debugging breakpoints and software-triggered system calls.

Whichever one fires, the sequence is identical: the CPU automatically pushes PC and the status register onto the stack — `cpu8bit1-3`'s own page-1 stack, doing real work here — then jumps to a fixed handler address read from a small vector table near the very top of memory ($FFFA–$FFFF holds the NMI/RESET/IRQ vectors). The handler ends with `RTI` (Return from Interrupt), which pulls the status register and PC back off the stack, resuming exactly where the program left off. This is exactly why `cpu8bit1-3`'s own status register includes the B flag — its entire purpose is letting the handler tell a genuine hardware IRQ apart from a deliberate software BRK once execution lands at that shared vector.

Only three vectors exist in total (NMI, RESET, and a shared IRQ/BRK vector) — one fixed handler address per category, no further sorting done by the hardware itself.

The Z80's Interrupt Model — NMI and Three Interrupt Modes

The Z80 also has an NMI, working much like the 6502's own (a fixed vector, unmaskable, automatic PC push). Its maskable interrupt is where the real richness shows up: the programmer selects one of **three interrupt**

modes via a dedicated instruction (`IM 0`, `IM 1`, `IM 2`):

- **IM 0** — the interrupting device itself supplies an instruction for the CPU to execute directly, usually a short call. This is inherited straight from 8080 compatibility (`cpu8bit1-5`'s own founding constraint) — flexible, but the most primitive and least commonly used of the three.
- **IM 1** — always jumps to one single fixed address ($0038), no matter which device interrupted. As simple as the 6502's own shared IRQ vector.
- **IM 2** — the genuinely powerful mode. The CPU builds a 16-bit pointer by combining the high byte from a dedicated register (**I**, the interrupt vector register) with a low byte the interrupting device itself supplies, then looks up a full 16-bit handler address at that computed location in a table the programmer sets up in advance. Because the device-supplied byte is even, up to 128 distinct table entries are addressable — meaning every peripheral can get its own dedicated handler, instead of one shared entry point that then has to work out which device actually interrupted.

IM 2'S POWER COMES WITH REAL SETUP DISCIPLINE

IM 2 doesn't work by accident — the I register has to be loaded with the correct table's high byte in advance, the interrupting hardware has to be wired to actually supply a compatible low byte, and the 128-entry table itself has to be built and populated correctly before any interrupt fires. This is the same "richness has a price" theme from `cpu8bit1-7`'s own cycle-cost warn-box, showing up here as setup complexity rather than raw runtime cost.

Where the Shadow Register Set Finally Pays Off

`cpu8bit1-6` previewed `EXX` / `EX AF,AF'` and named their real, historical purpose: fast interrupt entry. Now it's possible to see exactly why. A Z80 interrupt handler can execute a single `EXX` the moment it starts, instantly gaining a completely fresh, empty set of BC/DE/HL to work with — no need to push the main program's register values onto the stack to protect them first. A single `EXX` back at the end restores everything, and the handler returns.

A 6502 interrupt handler has no such option — anything it needs a register for, it must `PHA` / `PLA` (and similarly for X and Y, transferred through A first, since only A has native push/pull) around its own use of that register, each one a real, separate instruction costing real cycles on both entry and exit. The shadow set turns an entire category of that overhead into a single instruction.

Minimalism vs. Richness, a Third Time

	6502	Z80
Interrupt sources	NMI, IRQ, BRK (software) — 3 fixed vectors	NMI, plus a maskable INT with 3 selectable modes

	6502	Z80
Masking	The I flag in P disables IRQ	EI/DI instructions enable/disable the maskable interrupt
Handler dispatch	One shared vector per category — software must sort out the source	IM 2 gives up to 128 distinct, hardware-selected handlers
Register protection on entry	Manual PHA/PLA (and friends) around whatever's needed	A single EXX / EX AF,AF′ swaps in an entire fresh register set

This is the same contrast `cpu8bit1-2` and `cpu8bit1-5` established for the chips overall, and `cpu8bit1-8` made concrete for loops — now showing up a third time, in a third domain entirely.

ASSEMBLY1-8'S PREVIEW, FINALLY MADE CONCRETE

"Interrupts are real, more efficient LC-3 territory" is exactly how `assembly1-8` left this topic. Both chips in this course now show two genuinely different, real implementations of that idea — one minimal and uniform, one rich and selectively dispatched — closing a loop this whole course arc has been building toward since its very first LC-3 chapter.

Hands-On Exercises

EXERCISE 1

Explain the real difference between the 6502's IRQ and NMI in terms of maskability, and explain why BRK is categorized alongside them despite not being triggered by external hardware at all.

View solution

EXERCISE 2

Using this chapter's own explanation of IM 2, describe how the final 16-bit vector address is assembled from the I register and the device-supplied byte, and explain why this allows up to 128 distinct interrupt handlers rather than just one shared entry point.

View solution

EXERCISE 3

Using cpu8bit1-6's own shadow-register material and this chapter's own coverage, explain concretely why a Z80 interrupt handler using EXX enters and exits faster than an equivalent 6502 handler that must manually PHA/PLA the registers it needs.

📄 View solution

CHAPTER 9 QUICK REFERENCE

- **6502**: IRQ (maskable, hardware), NMI (unmaskable, hardware), BRK (software) — 3 fixed vectors, shared IRQ/BRK entry point
- **Z80**: NMI (unmaskable), plus a maskable INT with 3 selectable modes (IM 0/1/2)
- **IM 1** is as simple as the 6502's shared vector; **IM 2** combines the I register + a device-supplied byte into a full vectored table, up to 128 distinct handlers
- Both chips push PC (and status, on the 6502) to the stack automatically and return via RTI — the same mechanism, different vector richness
- **EXX / EX AF,AF'** finally pay off cpu8bit1-6's own preview — one instruction replaces the 6502's manual PHA/PLA register-saving on every interrupt entry/exit
- IM 2's power requires real setup discipline (I register, device wiring, table construction) — richness costing complexity, not just cycles or transistors
- This is the third domain (after registers and addressing) where the same minimalism-vs-richness contrast shows up concretely

CHAPTER 10 OF 12

Where These Chips Actually Lived

8-Bit CPUs — 6502/6510 & Z80

Chapter 10 · Where These Chips Actually Lived

Chapters 2–9 studied both chips as pure engineering — transistors, registers, cycles, interrupts. This chapter grounds all of it in real machines and real markets: where minimalism and richness actually ended up, in products people bought by the millions. It's also where this course's own title finally gets fully explained — the "6510" has been sitting there unexplained since `cpu8bit1-1`.

The 6502's Real-World Footprint

- **Apple II** (1977) — Steve Wozniak's design, one of the machines that helped launch the personal computer industry as a mass-market category.
- **Commodore PET, VIC-20, and Commodore 64** (the C64 in 1982) — one of the best-selling home computers of all time. The C64 specifically used the **6510**, not a stock 6502 — a close variant adding an 8-bit I/O port used for bank-switching between ROM and RAM, but otherwise instruction-set identical to the 6502 this entire course has been teaching. This is finally the full explanation behind the course's own "6502/6510" title.
- **Atari 2600** (1977) and later Atari 8-bit computers — the 2600 specifically used the **6507**, a cut-down 6502 in a smaller, cheaper package with fewer address lines.
- **Nintendo Entertainment System** (1983 Japan / 1985 US) — powered by the Ricoh 2A03, a 6502 derivative widely reported to have had its decimal mode (`cpu8bit1-3`'s own D flag) deliberately removed to avoid a licensing fee tied to that specific circuitry.

The Z80's Real-World Footprint

- **Sinclair ZX Spectrum** (1982) — hugely influential across the UK and European home computing scene, home to an enormous game library and demoscene.
- **The MSX standard** — a Japanese-led standardized home computer platform adopted by multiple manufacturers, all sharing Z80-based compatibility.
- **Amstrad CPC series** — another major European Z80-based home computer line.
- **The original Nintendo Game Boy** (1989) — powered by the Sharp LR35902, a genuine Z80 derivative that removes the shadow register set and IX/IY entirely, while adding a small number of its own new

instructions.

- **CP/M** — the dominant business and professional operating system of the late 1970s and early 1980s, running primarily on Z80-based (and 8080-compatible) machines — a huge share of the Z80's real commercial footprint that had nothing to do with games at all, made possible directly by `cpu8bit1-5`'s own 8080-compatibility story.

A Shared Pattern — Both Chips Spawned Cost/Licensing-Driven Derivatives

It's worth naming the parallel directly: both chips ended up inside an iconic games console via a derivative that deliberately removed something for cost or licensing reasons. The 6502 became the NES's 2A03 by losing decimal mode; the Z80 became the Game Boy's LR35902 by losing its entire shadow register set — the exact feature `cpu8bit1-9` just finished showing off as the Z80's own genuine interrupt-handling advantage. Even in real hardware built directly from these two chips, the same kind of trade-off this whole course has been tracing — what's worth keeping, what's worth cutting — kept happening all over again, one generation later.

CHIP	ICONIC MACHINES	NOTABLE DERIVATIVE
6502/6510	Apple II, Commodore 64 (6510), Atari 2600/800, NES	Ricoh 2A03 (NES) — decimal mode removed
Z80	ZX Spectrum, MSX, Amstrad CPC, CP/M business machines, Game Boy	Sharp LR35902 (Game Boy) — shadow registers and IX/IY removed

Both Are Still Actively Used Today

Neither chip is a purely historical curiosity. **WDC** (Western Design Center), founded by Bill Mensch — one of the 6502's own original co-designers named back in `cpu8bit1-2` — still designs, produces, and licenses modern 6502-family cores today. The Z80 itself remained in continuous commercial production for an extraordinarily long stretch — decades — before shifting primarily toward licensed IP cores rather than standalone chips. Both architectures also have genuinely active hobbyist communities today, building homebrew computers, new games, and demos on real (or faithfully compatible) hardware, not just studying them as history.

THE MARKET NEVER ACTUALLY SETTLED "WHICH CHIP IS BETTER"

`cpu8bit1-5`'s own tip-box already flagged this, and this chapter's real sales history confirms it: the 6502 and Z80 each achieved massive, independent commercial success on genuinely different terms — one through ultra-cheap home computers and consoles, the other through a mix of home computers and a dominant share of the era's professional computing market via CP/M. Neither philosophy "won" over the other in any simple sense.

TWO CHAPTERS LEFT

`cpu8bit1-11` takes everything this course has shown — technically in Chapters 2–9, commercially in this one — and formally names the RISC-vs-CISC throughline `cpu8bit1-1` only previewed at the very start. The capstone, `cpu8bit1-12`, closes the course with one more direct side-by-side comparison.

Hands-On Exercises

EXERCISE 1

Explain what the 6510 actually is relative to the 6502 this course has been teaching, and explain specifically why the Commodore 64 needed that particular variant rather than a stock 6502.

View solution

EXERCISE 2

Explain the real historical reason widely given for the NES's 2A03 removing the 6502's decimal mode, and connect it to the Game Boy's own LR35902 removing the Z80's shadow register set — what do the two removals have in common as a category of design decision?

View solution

EXERCISE 3

Using this chapter's own "Still Actively Used Today" section, explain why calling either the 6502 or the Z80 "obsolete" would be inaccurate — cite at least one concrete piece of evidence from the chapter for each architecture.

View solution

CHAPTER 10 QUICK REFERENCE

- **6502 family** — Apple II, Commodore 64 (via the 6510), Atari 2600 (via the 6507) and 8-bit line, NES (via the 2A03)
- **Z80** — ZX Spectrum, MSX, Amstrad CPC, CP/M business machines, Game Boy (via the LR35902)
- The 6510 = a 6502 plus an added I/O port for the C64's bank-switching — the "6510" in this course's own title, finally explained
- Both chips spawned a famous derivative that removed a real feature for cost/licensing reasons: the 2A03 (decimal mode) and the LR35902 (shadow registers)
- WDC (co-founded by 6502 co-designer Bill Mensch) and Z80-derived IP cores both remain in active production and licensing today
- Both chips have thriving modern hobbyist/homebrew communities — neither is purely a museum piece

- Neither philosophy "won" the market outright — both succeeded massively, on different terms

CHAPTER 11 OF 12

RISC vs. CISC — The Debate These Two Chips Actually Preview

8-Bit CPUs — 6502/6510 & Z80

Chapter 11 · RISC vs. CISC — The Debate These Two Chips Actually Preview

`cpu8bit1-1` named RISC and CISC in a single paragraph and immediately warned that neither term existed in 1975 or 1976. Nine chapters of real, concrete evidence later, it's time to formally define both terms properly and see exactly how much of that evidence actually supports the connection — including where it doesn't.

What RISC and CISC Actually Mean

RISC (Reduced Instruction Set Computing) favors a small number of simple, uniform instructions, each executing in a small, predictable number of cycles, with complex behavior built by combining several simple instructions in software rather than relying on one instruction that does more. It typically pairs this with a strict **load/store** discipline — exactly the principle `assembly1-4` already taught as LC-3's own core design rule: ALU instructions only ever operate on registers, and memory is touched only through dedicated load/store instructions. The philosophy has a real, documented origin: the Berkeley RISC project (David Patterson, around 1980) and Stanford's MIPS project, both built specifically as a reaction against increasingly complex designs, using real measurements of real programs to argue that simpler, more uniform instructions could pipeline better and actually run faster in practice.

CISC (Complex Instruction Set Computing) favors a larger, richer instruction set, where individual instructions can perform more work — sometimes combining memory access and computation in a single instruction — with variable-length encoding and more specialized addressing modes. The term itself emerged largely as a retrospective label, applied to already-existing designs (the DEC VAX, and eventually x86) once RISC's contrasting philosophy gave critics a name to compare them against.

The 6502 as a Genuine RISC Precedent

Every piece of this course's own 6502 coverage lines up with RISC's later hallmarks:

- **Small, uniform instruction count** — 56 mnemonics (`cpu8bit1-2`), a genuinely small set.
- **Narrow, predictable cycle costs** — `cpu8bit1-4`'s own table shows most 6502 instructions executing in a tight 2–7 cycle range, with nothing like the wide variance CISC-style complex instructions can have.

- **A real load/store discipline** — arithmetic routes through the accumulator, while `LDA` / `STA` are the dedicated instructions that actually touch memory. It's genuinely the same separation `assembly1-4` taught, arrived at independently.

The strongest evidence, though, isn't just resemblance — it's a real, acknowledged historical line. Acorn Computers, the British company behind the BBC Micro (a 6502-based machine), went on to design **ARM** — one of the most successful real RISC architectures in computing history, today running in the overwhelming majority of the world's smartphones. ARM's own original designers, including Sophie Wilson and Steve Furber, have spoken publicly about the 6502's efficient, minimal design as a direct influence on ARM's earliest philosophy. This isn't an abstract parallel drawn by hindsight — it's a chip this course has been teaching for ten chapters, sitting in the actual, documented ancestry of a real, dominant modern RISC family.

The Z80 as a Genuine CISC Precedent

The Z80's own coverage lines up just as clearly on the other side:

- **A large, rich instruction set** — roughly 158 mnemonics (`cpu8bit1-5`) against the 6502's 56.
- **Variable-length, prefix-extended encoding** — the CB/DD/ED/FD mechanism (`cpu8bit1-5`) exists specifically to pack in more complexity than a single opcode byte could hold.
- **Genuinely wide cycle-cost variance** — `cpu8bit1-7`'s own comparison showed `(HL)` at 7 cycles against `(IX+d)` at 19 — a spread the 6502 never approaches.
- **Instructions that combine multiple logical steps** — `DJNZ` (`cpu8bit1-8`) folds decrement, compare, and branch into one instruction, precisely the "do more per instruction" complexity RISC design deliberately avoids.

Here it's important to be precise rather than overreach: the Z80 is not a direct ancestor of x86 the way the 6502 is a direct, acknowledged ancestor of ARM. The Z80 descends from the 8080 via Zilog; Intel's own 8086 — the direct start of the x86 line — descends from the 8080 via Intel itself, as a rival, sibling design, not a child of the Z80. What genuinely connects them is philosophical, not genealogical: both emerged from the same instruction-rich, compatibility-driven tradition rooted in the 8080, and the same complexity-favoring impulse that produced the Z80's own richness is the same impulse that drove x86's own decades of accumulated complexity.

TRAIT	RISC HALLMARK	6502'S EVIDENCE	CISC HALLMARK	Z80'S EVIDENCE
Instruction count	Small	56 mnemonics (cpu8bit1-2)	Large	~158 mnemonics (cpu8bit1-5)
Register count	Many (reduces memory traffic)	Only 3 — see the honest divergence below	No specific requirement	~14 total, main + shadow (cpu8bit1-6)
Instruction complexity	One operation per instruction	Mostly simple (cpu8bit1-4)	Multiple operations combined	DJNZ: decrement+compare+branch (cpu8bit1-8)

TRAIT	RISC HALLMARK	6502'S EVIDENCE	CISC HALLMARK	Z80'S EVIDENCE
Memory access	Load/store only	LDA/STA separate from ALU ops	Often folded into complex instructions	`ADD A,(HL)` — memory access AND arithmetic, one instruction
Cycle-cost range	Narrow, predictable	2–7 cycles (cpu8bit1-4)	Wide variance	7–19+ cycles (cpu8bit1-7)

Why Neither Chip Is "Really" RISC or CISC

`cpu8bit1-1` and `cpu8bit1-5` both already flagged the basic anachronism — neither design team was building toward a checklist that didn't exist yet. But there's a sharper, more specific honesty worth adding here: the 6502's register count table above isn't just an anachronism, it's a genuine *divergence*. Real RISC design, once formalized, generally concluded that CPUs should have **many** registers — specifically to reduce how often a program needs to touch memory at all. The 6502 has the opposite: **only 3**, for a completely different reason (`cpu8bit1-2`'s own transistor-cost pressure, not a deliberate strategy to minimize memory traffic). `cpu8bit1-4`'s own zero-page trick is, in a sense, the 6502's real compensation for lacking the very thing later RISC design would have called for directly. The 6502 arrived at several genuinely RISC-shaped answers, from a starting motivation that had nothing to do with RISC's own actual reasoning — and diverges sharply on the one trait (register count) where the two philosophies' motivations point in opposite directions entirely.

A PRECEDENT, NOT A PROTOTYPE

"Precedent" is the right word, and it's a deliberately weaker claim than "prototype." The 6502 previews RISC's own conclusions without having been built toward them; it isn't an early, incomplete RISC chip waiting for the theory to catch up. The Z80 previews CISC's own conclusions the same way, arrived at through 8080 compatibility rather than any conscious embrace of complexity for its own sake.

Bridging to x86-64

`assembly1-1` named x86-64 as the future course carrying "four decades of backward-compatible accretion" — and that lineage traces conceptually back to exactly the same instruction-rich, compatibility-first tradition this chapter just placed the Z80 within. Where this chapter's own Z80 material shows that tension at 1976 scale — roughly 158 instructions, a handful of prefix bytes — the still-unwritten x86-64 course picks up the same underlying story after another four decades of the same forces compounding, at a scale this course has only just begun to preview.

ONE CHAPTER LEFT

`cpu8bit1-12` closes the course with one final side-by-side routine, applying everything this chapter just formalized to real code one more time — the last direct payoff of studying these two chips together rather than apart.

Hands-On Exercises

EXERCISE 1

Using this chapter's own "Why Neither Chip Is Really RISC or CISC" section, explain the one specific way the 6502 genuinely diverges from later RISC ideals, and explain why that divergence happened for a completely different reason than real RISC chips' own decision to include many registers.

View solution

EXERCISE 2

Explain the real historical connection between Acorn, ARM, and the 6502 described in this chapter, and explain why it's stronger evidence for this chapter's own thesis than simply observing that the 6502 "resembles" RISC principles in the abstract.

View solution

EXERCISE 3

Using this chapter's own compare table, explain specifically why `ADD A,(HL)` combining a memory read and an arithmetic operation into one instruction counts as a genuinely CISC-style trait — contrast it against how the 6502 would have to express the same operation using separate instructions.

View solution

CHAPTER 11 QUICK REFERENCE

- **RISC** — small, uniform, predictable-cycle instructions, strict load/store discipline; formalized ~1980 by Berkeley RISC/Stanford MIPS
- **CISC** — larger, richer instruction sets, variable-length encoding, instructions that combine multiple operations; largely a retrospective label
- The 6502 matches RISC on instruction count, cycle uniformity, and load/store discipline — and is a real, acknowledged ancestor of ARM via Acorn
- The Z80 matches CISC on instruction count, prefix-extended variable-length encoding, wide cycle variance, and multi-step instructions like DJNZ
- The Z80-to-x86 connection is philosophical (shared 8080-rooted, compatibility-first tradition), not a direct chip lineage the way 6502-to-ARM is
- The 6502's own tiny register count is a genuine DIVERGENCE from later RISC ideals — real RISC chips use many registers to cut memory traffic; the 6502 has few, for pure cost reasons
- Both chips are precedents, not prototypes — neither was built toward a theory that didn't exist yet

- This chapter's Z80 material previews, at 1976 scale, the same tension the still-unwritten x86-64 course will cover after four more decades of accretion

CHAPTER 12 OF 12

Capstone: Building and Comparing Two Small Programs

8-Bit CPUs — 6502/6510 & Z80

Chapter 12 · Capstone — Building and Comparing Two Small Programs

One task, one final time, on both chips: find the largest value in a 5-byte array. Unlike `cpu8bit1-8`'s own summing routine, this one uses a real **subroutine call** — the first time either chip's real hardware stack (`cpu8bit1-3`, `cpu8bit1-7`) actually gets exercised by a working program in this course, rather than just described.

The 6502 Version

```
        LDX #1          ; start comparing from index 1
        LDA ARRAY       ; A = ARRAY[0]
        STA MAX         ; MAX = ARRAY[0] (the starting candidate)
LOOP    CPX #5
        BEQ DONE        ; NEW — branch if equal (complements BNE from cpu8bit1-8)
        LDA ARRAY,X     ; A = the next candidate — zero-page,X (cpu8bit1-4)
        JSR UPDATE_MAX  ; NEW — call the subroutine (real stack use, cpu8bit1-3)
        INX
        JMP LOOP
DONE    JMP DONE        ; no HALT on the 6502 (cpu8bit1-8)

; --- Subroutine: UPDATE_MAX ---
; Input: A = candidate. Updates MAX in place if the candidate is bigger.
UPDATE_MAX
        CMP MAX         ; NEW — compare A against MAX (like SBC, but nothing stored)
        BCC SKIP        ; NEW — branch if Carry Clear (A < MAX)
        STA MAX         ; A >= MAX — update it
SKIP    RTS             ; NEW — return, pulling the address JSR pushed

ARRAY   .BYTE 10,50,30,80,20
MAX     .BYTE 0
```

New here: **BEQ** (branch if equal — the counterpart to `cpu8bit1-8`'s own `BNE`), **CMP** (compares A against a value the same way `ADC`/`SBC` would, without storing a result), **BCC** (branch if Carry is clear), and — for the first time in this course — **JSR**/**RTS**, a real subroutine call and return, automatically using the page-1 stack `cpu8bit1-3` only described until now.

The Z80 Version

```
        LD HL, ARRAY
        LD A, (HL)       ; A = ARRAY[0]
        LD (MAX), A      ; MAX = ARRAY[0]
        INC HL
        LD B, 4           ; 4 remaining comparisons
LOOP    LD A, (HL)       ; A = the next candidate — register-indirect (cpu8bit1-7)
        CALL UPDATE_MAX ; NEW — call the subroutine (real stack use, cpu8bit1-7)
        INC HL
        DJNZ LOOP        ; cpu8bit1-8
DONE    JR DONE

; --- Subroutine: UPDATE_MAX ---
; Input: A = candidate. Updates MAX in place if the candidate is bigger.
UPDATE_MAX
        LD C, A          ; save the candidate in C — NOT B, which DJNZ's own loop still needs
        LD A, (MAX)
        CP C             ; NEW — compare A (MAX) against C (candidate)
        JR NC, SKIP      ; NEW — jump if No Carry (MAX >= candidate)
        LD A, C
        LD (MAX), A
SKIP    RET             ; NEW — return, pulling the address CALL pushed

ARRAY   DEFB 10,50,30,80,20
MAX     DEFB 0
```

New here: **CP** (compares A against a value, like `CMP` on the 6502), **JR NC** (jump if no carry), and — for the first time on this side too — **CALL**/**RET**, the Z80's own subroutine call, using its fully flexible 16-bit stack from `cpu8bit1-7`.

> **A DELIBERATE REGISTER CHOICE, NOT AN ACCIDENT**
>
> The subroutine saves the candidate in **C**, not B — because B is the outer loop's own `DJNZ` counter, and overwriting it inside the subroutine would silently corrupt the loop the moment the subroutine returned. This is exactly the register-preservation discipline `assembly1-7` first introduced for R7, showing up here as an equally real concern for an entirely different register on an entirely different chip.

The Real Payoff: Two Opposite Flag Conventions

Both subroutines do the same comparison — yet the 6502's `CMP` / `BCC` and the Z80's `CP` / `JR NC` read the Carry flag in **opposite** directions:

CHIP	AFTER CMP/CP	CARRY MEANS	INSTRUCTION USED TO DETECT "SMALLER"
6502	`CMP MAX` (A − MAX)	SET if A ≥ MAX (no borrow needed)	`BCC` — Carry Clear means A < MAX
Z80	`CP C` (A − C)	SET if A < C (a borrow was needed)	`JR NC` — No Carry means A ≥ C

On the 6502, Carry set means "no borrow was needed" — the first value was large enough. On the Z80, Carry set means the exact opposite — "a borrow was needed," meaning the first value was *too small*. Anyone porting comparison logic between the two chips has to consciously flip this reading, or the ported code will silently do the wrong thing while looking completely correct on the page. It's a small, precise, entirely real example of exactly the kind of "same-looking operation, genuinely different behavior" this whole course has been surfacing since `cpu8bit1-1`.

Chapter Attribution

CAPSTONE PIECE	CONCEPT	FROM
Both chips' overall approach	Cost-driven minimalism vs. compatibility-driven richness	cpu8bit1-2, cpu8bit1-5
ARRAY,X / (HL) with INC	Zero-page,X and register-indirect addressing	cpu8bit1-4, cpu8bit1-7
JSR/RTS and CALL/RET	Real subroutine calls, finally exercising the hardware stack	cpu8bit1-3, cpu8bit1-7
CPX/BEQ, DJNZ	Loop management — three 6502 instructions vs. one Z80 instruction	cpu8bit1-8
CMP/BCC vs. CP/JR NC	Genuinely opposite Carry-flag conventions between the two chips	cpu8bit1-3, cpu8bit1-6 (the status/flag registers each chip introduced)
The deliberate C-not-B register choice	Register preservation across a subroutine call	assembly1-7, echoed for the Z80's own registers here

HONEST SCOPE NOTE

This capstone deliberately stays within what this course actually taught. Left out, on purpose: any real hardware or emulator setup walkthrough (this course has been entirely about reading and reasoning through code on the page); a full instruction-set reference for either chip (only the instructions actually needed for these examples were introduced); interrupt-driven demonstrations (cpu8bit1-9 covered the mechanism, but no worked interrupt-driven program was built); and sound or graphics chip programming, which would require covering hardware entirely specific to individual machines rather than the CPUs themselves. None of these are gaps in what was taught — they're deliberate boundaries of a course about the two chips themselves, not the machines built around them.

THE COURSE THIS CLOSES, AND THE ONE IT OPENS

This closes the 8-Bit CPUs course — twelve chapters tracing two real, historically significant chips from their founding constraints (cost vs. compatibility) through registers, addressing, interrupts, real markets, and finally a formal RISC-vs-CISC framing with a real, documented lineage into ARM. It also stays open: `cpu8bit1-11` already named the still-unwritten x86-64 course as the place this same tension continues, at a scale forty more years of accretion actually produces.

Hands-On Exercises

EXERCISE 1

Trace the 6502's UPDATE_MAX subroutine for a call where A (the candidate) = 30 and MAX currently holds 50. Using this chapter's own CMP/BCC semantics, determine whether MAX gets updated, and state the value of MAX afterward.

View solution

EXERCISE 2

Trace the Z80's UPDATE_MAX subroutine for a call where the candidate is 80 and MAX currently holds 50. Using this chapter's own CP/JR NC semantics (note: the OPPOSITE Carry convention from the 6502), determine whether MAX gets updated, and state the value of MAX afterward.

View solution

EXERCISE 3

Pick three rows from this chapter's own chapter-attribution table and explain, in one or two sentences each, exactly which piece of this capstone's code draws on that chapter's material and why it was needed here.

View solution

CHAPTER 12 QUICK REFERENCE — COURSE RECAP

- **cpu8bit1-1** — the 1975–76 boom, a bridge from assembly1, RISC/CISC previewed
- **cpu8bit1-2 to -4** — the 6502: cost-driven minimalism, the full register/stack picture, zero page as "extra registers"
- **cpu8bit1-5 to -7** — the Z80: compatibility-driven richness, main/shadow/index registers, addressing and a fully flexible stack
- **cpu8bit1-8** — one routine, both chips: DJNZ vs. INX/CPX/BNE, and why cycle counts alone can mislead
- **cpu8bit1-9** — interrupts: 3 fixed vectors vs. 3 selectable modes, EXX's real payoff
- **cpu8bit1-10** — real machines, real markets, both chips still active today
- **cpu8bit1-11** — RISC vs. CISC formally defined, the real 6502→ARM lineage, an honest divergence named
- **cpu8bit1-12** — a real subroutine call on both chips, surfacing a genuinely opposite flag convention
- Next stop: the still-unwritten x86-64 course, picking up the Z80's own richness story after four more decades