SAP-3/16

Processor Reference Manual

Architecture, instruction set and peripherals — every bit, every T-state, every register in one place.

Jan Müller

2026

Contents

Appendices

Preface

This book exists because of questions that are hard to answer from memory. How many T-states does `LDA (zp),Y` cost? Does `CMP` touch the overflow flag? What exactly does it mean that `DIV` by zero "does not crash"? Which bit of `TERM_STATUS` is sticky, and when is it cleared?

The answers exist — in the microcode, in the ALU, in the device handlers — but as long as they have to be hunted for in source files, they are not a reference. This book collects them in one place.

The machine

SAP-3/16 is an 8-bit processor with a 16-bit address bus: eight bits of data, a flat 0x0000–0xFFFF address space with no banking, 179 legal opcodes built from 79 mnemonics and eight addressing modes. Vectored interrupts, a stack in main memory, indexed and indirect addressing.

It is not a historical machine you could buy. It is an architecture designed so that **everything is visible**: every opcode decodes from its own bits, every transfer over a bus is a step you can single-step, and boot is a sequence of ordinary memory reads.

How this book differs from the others

There are three books in the library and each does something different:

A Computer in Your Palm tells the story of how a computer works, from a single bit to programs with sound and graphics. If this is your first processor, start there. Currently in Czech only.

Let's Build Arkanoid builds one specific game stage by stage. It teaches how a **project** is done in assembly, not how an instruction works. Czech only.

The Reference Manual this book. It tells no story; it answers questions. It is written to be opened in the middle.

Anyone who can already program and wants to know this machine can start right here. Parts I and II read in order as an account of the architecture; Part III is a dictionary you consult for one instruction; Part IV describes the peripherals.

Notation

Numbers are written the way the assembler takes them: `0x2A` hexadecimal, `0b101010` binary, `42` decimal. Where encoding is discussed, bits are numbered from the most significant, so `bit 7` is the top one.

The addressing modes keep the same abbreviations throughout:

Written	Meaning
`#n`	immediate — the value is in the instruction
`zp`	zero page — a one-byte address, 0x0000-0x00FF
`abs`	absolute — a two-byte address anywhere in memory
`abs,X` / `abs,Y`	absolute indexed by X or Y
`(zp)`	indirect — a pointer stored in the zero page
`zp,X`	zero-page indexed; the sum wraps inside the page
`(zp),Y`	indirect indexed — Y is added to the zero-page pointer

Flags are listed only where an instruction **defines** them. An empty entry means the flag keeps its previous value — not that it is cleared. That distinction is fundamental in SAP-3/16: logic operations leave `C` alone precisely so that bits can be masked between a compare and its jump.

Microcode is written in square brackets, one pair per T-state, signals separated by commas; the number in parentheses at the end is the total T-state count including the fetch:

```
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [ORO,AI,FI]  (5)
```

A question mark after a bracket marks a step that is skipped when a condition fails (the conditional jumps use it).

Where the numbers come from

Every opcode table, every T-state count, every flag set and every register address in this book is **generated from the processor's implementation**, not transcribed. The typesetting is done by hand; the numbers are not. A check that the generated data matches the source runs on every build of the project — the book cannot drift from the machine without breaking the build.

That has one practical consequence for the reader: if you find a number here that does not match what your machine does, it is not a typo in the book. It is either a different ISA version, or a bug worth reporting.

This book describes **ISA 3.1**.

What this book does not cover

Deliberately absent: Tiny BASIC, the standard library, SAP-DOS as an operating system (only boot and the sector format, because those are part of the hardware contract), tutorials on writing games, and the details of the HDL or the circuit boards. The final chapter on hardware is an overview, not a service manual.

PART I

Architecture

CHAPTER 1

Architecture Overview

SAP-3/16 is an 8-bit processor with a 16-bit address bus. That single sentence decides almost everything else: data moves a byte at a time, addresses take two bytes, and because only one byte crosses the data bus per step, every 16-bit address costs the processor extra T-states. The whole microarchitecture rests on that asymmetry — and you can see it in the length table of every dictionary entry.

Key specifications

Parameter	Value
Data width	8 bits
Address width	16 bits, a flat 64 KB with no banking
Registers	A, B, X, Y (8 b) + SP, PC (16 b) + OUT
Flags	C, Z, N, V, I
Legal opcodes	179 of 256 (75 slots free)
Mnemonics	79
Addressing modes	8
Instruction length	1 to 3 bytes
Stack	in main memory, page 0x0100–0x01FF
Interrupts	vectored, three sources plus software `BRK`
Vectors	0xFFFA BRK, 0xFFFC RESET, 0xFFFE IRQ
Peripherals	0xFF00–0xFF7F, 16-byte stride per device
Video memory	0x8000–0xBFFF

Table 1. The SAP-3/16 architecture at a glance, ISA 3.1.

Three design goals

The architecture has three goals and they are ordered — when they conflict, the earlier one wins. Most decisions that later look strange can be explained by that order.

1. **Teachability** every opcode must be decodable from its own bits, every address transfer must be visible on the schematic, and boot must be single-steppable as ordinary memory reads. Hence no prefix bytes, hence reset behaving like an instruction, and hence even `NOP` carrying one idle microcode step.

2. **Regularity** orthogonal addressing modes, systematic encoding, few exceptions. An instruction is not a table lookup but a formula — `gg|ooo|mmm`: group, operation, mode.
3. **Sufficiency** programs of a few kilobytes must be pleasant to write. Hence the zero page as a pointer file, `MUL` / `DIV` in hardware, 16-bit operations on a zero-page word, and the blitter.

> **Note**
>
> The goals are ordered, not balanced. Wherever T-states could have been saved at the cost of no longer seeing what the processor does, visibility won. Speed comes fourth, deliberately.

Where it comes from

The name is a thank-you: SAP — **Simple As Possible** — is the family of teaching processors from Malvino's *Digital Computer Electronics*. SAP-1 adds two numbers, SAP-2 adds jumps and I/O, SAP-3 registers and a stack. They are machines designed to be drawn on a blackboard.

SAP-3/16 takes the approach from that tradition, not the schematic. It adds a second address byte (hence the "/16"), a real memory map with peripherals, vectored interrupts and systematic instruction encoding. From the 8-bit classics it borrows what worked: the zero page and post-indexed indirect addressing from the 6502, the `TXS` / `TSX` convention for the stack pointer, a full-descending stack.

What it does not borrow is irregularity. On a 6502 you have to learn which instructions support which modes, because there is no rule. Here there is a rule, and it is visible in the opcode.

The programming model in brief

The details are in the chapter on registers and flags; this is a map of the terrain so the examples in the following chapters make sense.

The programmer sees four 8-bit registers. **A** is the accumulator — the target of nearly all arithmetic. **B** is the ALU's second operand. **X** and **Y** are indices. Plus two 16-bit ones: **PC** and **SP**.

> **Caution**

B is not a place to park a value. Every binary ALU operation loads its operand into it, so whatever a program left there disappears at the next `ADD`, `CMP` or `AND`. It is the most common surprise for people arriving from other 8-bits — and the reason the `PHB` / `PLB` pair exists.

Inside the processor there are a few more registers that no instruction can address, but without which the microcode makes no sense: **IR** holds the opcode being executed, **OP** and **OPH** the operand bytes, **MAR** the address currently being touched, **TMP** is the ALU's working input and **addrCarry** a one-bit latch for the carry between address bytes. That last pair is why indexed addressing destroys no user register.

Three implementations, one behaviour

The same binary runs on three different machines:

The emulator TypeScript, running in the browser at sap-3.com. It steps T-state by T-state, travels back in time and shows the control signals and bus contents.

AGATA-1 a Verilog core on an FPGA (Sipeed Tang Nano 20K), the heart of the MAXIK console. HDMI output, SD card, NES controller ports, turbo mode.

The ESP32 port the same processor in C++ on a microcontroller with a small display.

These are not three independent implementations of one description — those would diverge. The microcode has a single source and both the HDL and the firmware receive it **generated**; the ALU is checked in simulation against an exhaustive vector set computed from the emulator, and whole programs are replayed against reference traces T-state by T-state. "The browser behaves like the board" is therefore a build artefact, not a promise in a document.

> **Note**
>
> That detail matters for the whole book: the T-state counts you find here are not indicative. They are the counts you will measure on the board.

It is also why the architecture lacks things you would otherwise expect at such a clock rate — caches, branch prediction, out-of-order execution. None of them is missing for reasons of difficulty; they are missing because they would break the determinism that stepping, time travel and reference traces rest on.

Where to go next

The rest of Part I works through the layers: memory (what lives where), encoding (what an instruction looks like), the data path (where the bytes flow), the instruction cycle (in what order) and interrupts (what happens when something cuts in). If you only want to program, skip to Part II; if you are looking up one instruction, you want the dictionary in Part III.

CHAPTER 2

The Memory Model

SAP-3/16 sees one flat 64 KB address space. No banking, no windows, no separate spaces for code and data. An address is a 16-bit number and it always means the same place.

The whole space is ordinary RAM with one exception: the block 0xFF00–0xFF7F, which the processor intercepts before a read or write reaches memory and hands to a device. Everything else — video memory included — behaves like memory. A write to VRAM is a normal memory write: it appears in the journal, you can put a watchpoint on it and you can rewind past it. The display only **reads** it.

The memory map

Address	Size	Region
`0x0000–0x00FF`	256 B	Zero page — fast access and the pointer file
`0x0100–0x01FF`	256 B	Stack
`0x0200–0x7FFF`	32256 B	Program and data
`0x8000–0x83E7`	1000 B	Character buffer (40×25)
`0x83E8–0x83FF`	24 B	reserved
`0x8400–0x8BFF`	2048 B	Glyph RAM (8 bytes per character)
`0x8C00–0x8FE7`	1000 B	Color RAM (one attribute per cell)
`0x8FE8–0x8FFF`	24 B	reserved
`0x9000–0xAF3F`	8000 B	Bitmap framebuffer
`0xAF40–0xAFDF`	160 B	reserved
`0xAFE0–0xAFFF`	32 B	Sprite attributes (8 × 4 bytes)
`0xB000–0xBFFF`	4096 B	Sprite patterns (32 × 128 bytes)
`0xC000–0xFEFF`	16128 B	Data RAM — large arrays, heap
`0xFF00–0xFF7F`	128 B	Memory-mapped I/O
`0xFF80–0xFFF9`	122 B	System RAM — ISR scratch, boot-ROM home
`0xFFFA–0xFFFF`	6 B	BRK / RESET / IRQ vectors

Figure 1. How the 64 KB is divided. The rows add up to the whole space — the ones marked reserved are holes nothing uses today.

The zero page

The first 256 bytes have two special properties, and both are worth knowing because they shape how programs are written on this machine.

It is cheaper. An instruction with a one-byte address is one byte shorter and three T-states faster than the same instruction with a two-byte address. That is not cosmetic: in a tight loop it is tens of percent.

It is the only place pointers can live. The `(zp)` and `(zp),Y` modes cannot take a pointer from anywhere — they always read it from the zero page. Two adjacent bytes there form a 16-bit pointer (low byte first, as everywhere else in this machine) and the `INW` and `DEW` instructions can step it.

The zero page is therefore not merely "fast memory" but a **pointer file**. Most non-trivial programs carve it into named slots right at the top:

```
ptr     EQU 0x10        ; pointer into a buffer (2 bytes)
count   EQU 0x12
rem     EQU 0x13
```

The stack

The stack lives on page 0x0100–0x01FF and **SP** resets to 0x0200 — that is, just **past** its end.

It is full-descending with pre-decrement: a push first lowers **SP** and only then writes, so the first pushed byte lands at 0x01FF and **SP** always points at the newest item. A pop is the reverse — read, then increment.

> **Note**
>
> The page 0x0100–0x01FF is a convention, not hardware. **SP** is a full 16-bit register and `TXYS` can write anything into it. Nothing stops you moving the stack elsewhere — you only lose the property that an overflow falls into the zero page, where it is conspicuous.

Little-endian, everywhere

16-bit values are stored low byte first. This holds without exception for:

- instruction operands (`JMP 0x1234` is stored as `C8 34 12`),
- the vectors at the top of memory,
- pointers read by the `(zp)` and `(zp),Y` modes,

- data emitted by the `.WORD` directive.

Vectors and boot

The last six bytes of memory hold three addresses:

Address	Vector	Read when
0xFFFA	BRK	the `BRK` instruction executes
0xFFFC	RESET	after power-up and after reset
0xFFFE	IRQ	a hardware interrupt is taken

Vectors are nothing special — they are ordinary bytes in RAM that the microcode reads over the same data bus as everything else. That is why boot can be single-stepped: reset clears the registers, sets **SP** and injects into the instruction register a pseudo-opcode whose entire microcode is two memory reads. The detail is in the chapter on interrupts.

The assembler fills the RESET vector itself, pointing it at the first emitted instruction. Vectors are written explicitly with the `.VECTOR` directive:

```
        .VECTOR IRQ, handler
        .VECTOR BRK, syscall
```

Where a program belongs

The assembler's default `.ORG` is 0x0200, immediately past the stack page. The space from there to 0x7FFF is roughly 31.5 KB and holds both code and data. Large arrays also fit past VRAM, in the region starting at 0xC000.

> **Pitfall**
>
> The assembler refuses to place code or data in the peripheral block 0xFF00–0xFF7F. This is not fussiness: the bytes would never reach memory, because the processor intercepts those addresses first — the program would look assembled while nothing was there.

Video memory

The region 0x8000–0xBFFF is 16 KB reserved for the picture, split into a character buffer, glyph memory, color RAM, a bitmap framebuffer and sprite data. Each part is described in the video chapters.

For the memory model only one thing matters: **none of it is registers**. It is bytes in RAM. On this machine there is no difference between "write a character to the screen" and "write a byte to memory" — which is why the picture can be built by a blitter that knows nothing about screens and only copies memory.

CHAPTER 3

Instruction Encoding

An opcode is not an entry in a table you have to memorise. It is a structure. Eight bits split into three fields, each answering one question:

```
7 6 5 4 3 2 1 0
g g o o o m m m
```

gg the group — what kind of instruction this is
ooo the operation within the group
mmm the addressing mode, where one makes sense

Four groups

`gg`	Range	Group
`00`	0x00–0x3F	implied instructions, 1 byte
`01`	0x40–0x7F	moves between registers and memory, plus `CPX` / `CPY`
`10`	0x80–0xBF	arithmetic and logic, always targeting **A**
`11`	0xC0–0xFF	control flow, read-modify-write, `BIT`, word operations

Groups 01 and 10 are fully systematic: the opcode is literally `gg | ooo << 3 | mmm`. That is why the same mnemonics sit in contiguous runs of eight in the opcode matrix (Appendix A) — one row is one operation, one column one addressing mode.

Groups 00 and 11 cannot be systematic, because their instructions either take no operand or take an unusual one. Regularity continues inside them in smaller blocks, though: the conditional jumps 0xC0–0xC7 have the form `11000 ff s`, where `ff` picks the flag (Z, C, N, V) and `s` the polarity — jump if set.

A worked decode

Take the byte 0x67:

Field	Bits	Means
`gg`	`01`	the load/store group
`ooo`	`100`	operation `STA` — store the accumulator
`mmm`	`111`	mode `(zp),Y`

So 0x67 is `STA (zp),Y`. Nothing had to be looked up — eight bits split into three fields was enough. This is exactly what the "teachability" goal from the first chapter buys: the instruction decoder can be described in one sentence, and therefore drawn on a blackboard.

Addressing modes and their encoding

The `mmm` field has eight values and all eight are used:

Mode	mmm	Syntax	Operand (B)	Effective address
`IMM`	`000`	`#n`	1	`—`
`ZP`	`001`	`zp`	1	`0x00:zp`
`ABS`	`010`	`abs`	2	`abs`
`ABS_X`	`011`	`abs,X`	2	`abs + X`
`ABS_Y`	`100`	`abs,Y`	2	`abs + Y`
`IND`	`101`	`(zp)`	1	`[zp, zp+1]`
`ZP_X`	`110`	`zp,X`	1	`0x00:((zp + X) & 0xFF)`
`IND_Y`	`111`	`(zp),Y`	1	`[zp, zp+1] + Y`

Figure 2. The addressing modes and their encoding in the `mmm` field.

Each mode's semantics are in the addressing-modes chapter; what matters here is that the value of `mmm` means the **same thing** across groups 01 and 10. Learn eight values and you can read half the opcode space.

Which combinations are legal

Not every operation supports every mode — some make no sense (`STA #n` would store into a constant) and some are not worth the encoding. The legality matrix is this:

Instruction	`#n`	`zp`	`abs`	`abs,X`	`abs,Y`	`(zp)`	`zp,X`	`(zp),Y`
`LDA`	•	•	•	•	•	•	•	•
`LDX`	•	•	•	—	•	—	—	—
`LDY`	•	•	•	•	—	—	•	—
`STA`	—	•	•	•	•	•	•	•
`STX`	—	•	•	—	—	—	—	—
`STY`	—	•	•	—	—	—	—	—
`CPX`	•	•	•	—	—	—	—	—
`CPY`	•	•	•	—	—	—	—	—
`ADD`	•	•	•	•	•	•	•	•
`ADC`	•	•	•	•	•	•	•	•
`SUB`	•	•	•	•	•	•	•	•
`SBC`	•	•	•	•	•	•	•	•
`AND`	•	•	•	•	•	•	•	•
`OR`	•	•	•	•	•	•	•	•
`XOR`	•	•	•	•	•	•	•	•
`CMP`	•	•	•	•	•	•	•	•

Figure 3. Legal operation × addressing-mode combinations in groups 01 and 10. A blank cell means the opcode exists as a byte but the processor rejects it.

The pattern is readable: **`A`** supports everything, the index registers have a narrower offering (indexing an index makes no sense), `STA` has no immediate form and `STX` / `STY` only the two basic address forms.

Instruction length

Length follows from the group and the mode, not from a table:

Group	Length
`00`	always 1 byte
`01`, `10`	2 bytes for `#n`, `zp`, `(zp)`, `zp,X`, `(zp),Y`; 3 bytes for `abs` and `abs,X` / `abs,Y`
`11`	per instruction: `RET` / `RTI` / `BRK` 1 byte, `zp` forms 2 bytes, `abs` forms 3 bytes

Put differently: an instruction is one byte plus however many bytes its operand needs — which is 0, 1 or 2.

Illegal opcodes

Of the 256 possible bytes, 179 are legal instructions. Two are reserved for pseudo-opcodes the hardware injects and which can never be fetched from memory (0xFD for reset, 0xFE for an interrupt). The remaining 75 slots are free.

A free slot is not a `NOP`. When the processor tries to execute a byte with no micro-program it stops and reports an illegal opcode together with the address it found it at. Two of these cases have their own diagnostics, because they are the most common ways a program runs away:

Byte	Diagnosis
0x00	"Executed empty memory" — the program jumped into a zero-filled region
0xFF	"Executed 0xFF-filled memory" — typically running off the end of the program

> **Pitfall**
>
> The difference between "illegal opcode" and "free slot" is only time. Free slots are where the ISA can grow — and several were taken in ISA 3.1 (`JGT`, `JLE`, `JGES`, `JLTS`, `BIT`, `ADW`, `SBW`, `LXY`, `SXY`, `PHB`, `PLB`). A program relying on a particular byte "doing nothing" will stop working in the next version.

CHAPTER 4

The Data Path and Control Signals

A processor does nothing except open and close a handful of gates each tick. Which register puts a byte on the bus, which one takes it, what the ALU does to it on the way — that is all of it. The control signals are the names of those gates, and the microcode is the list of which are closed on which tick.

This chapter is the glossary for that list.

Two buses

There are two paths through SAP-3/16 and they differ sharply in width.

The **data bus** is 8 bits wide and carries absolutely everything — transfers between registers, reads and writes to memory, operands into the ALU and results out of it.

The **address path** is 16 bits wide, but exactly three transfers use it:

Signal	Transfer
`PCM`	MAR := PC
`SPM`	MAR := SP
`JPW`	PC := OPH:OP (absolute jump)

Every other movement of an address happens eight bits at a time over the data bus. That is the most important sentence in this chapter, because the cost of 16-bit work on an 8-bit machine follows from it: every address byte that cannot take one of those three paths costs a T-state of its own.

It is where the entire difference between `LDA zp` (6 T) and `LDA abs` (9 T) comes from — the three extra ticks are fetching the second address byte from memory and putting it into the high half of **MAR**.

Naming convention

The signals have systematic names and it pays to read them as abbreviations:

- ending in `O` (**out**) — something **drives** the data bus,
- ending in `I` (**in**) — something **latches** from the data bus,
- `MI`, `MIL`, `MIH` load the address register **MAR** (zero-extended, low half, high half),
- `V…` points **MAR** at one of the vectors.

At most one bus source may be asserted in a T-state — otherwise two registers would drive the same wires. There may be several sinks: the byte on the bus can be taken by the accumulator and the flag latch at once.

Signal groups

There are 70 signals, split into six mutually exclusive groups. That split is not merely a documentation aid — it is the layout of the microword from which the FPGA's ROM is generated, and tests check that the partition is exact.

Data-bus sources

Signal	Description
`RO`	RAM Out - read memory at MAR onto the data bus
`AO`	A Register Out
`BO`	B Register Out
`XO`	X Register Out
`YO`	Y Register Out
`TO`	TMP Register Out
`ORO`	Operand (low) Register Out
`OHO`	Operand-high Register Out
`CLO`	PC low half onto the data bus (CALL/IRQ push)
`CHO`	PC high half onto the data bus (CALL/IRQ push)
`SLO`	SP low half onto the data bus (TSXY)
`SHO`	SP high half onto the data bus (TSXY)
`FLO`	Flags Out - pack C/Z/N/V/I into a status byte on the bus
`EO`	ALU Out - ALU result onto the data bus
`EXO`	ALU secondary result out (MUL high byte / DIV remainder)

Data-bus sinks

Signal	Description
`MI`	MAR In, zero-extended - MAR := 0x00:bus (zero-page address)
`MIL`	MAR low half := bus
`MIH`	MAR high half := bus
`RI`	RAM In - write the data bus to memory at MAR
`II`	Instruction Register In
`AI`	A Register In

Signal	Description
BI	B Register In
XI	X Register In
YI	Y Register In
TI	TMP Register In
ORI	Operand (low) Register In
OHI	Operand-high Register In
OI	Output Register In
JPL	PC low half := bus (RET/RTI pop)
JPH	PC high half := bus (RET/RTI pop)
SLI	SP low half := bus (TXYS)
SHI	SP high half := bus (TXYS)
FLI	Flags In - restore C/Z/N/V/I from a status byte on the bus

Address path (MAR)

Signal	Description
PCM	MAR := PC (16-bit address path)
SPM	MAR := SP (16-bit address path)
MRI	MAR := MAR + 1 (second byte of a pointer or vector)
VRS	MAR := 0xFFFC (RESET vector base)
VIR	MAR := 0xFFFE (IRQ vector base)
VBR	MAR := 0xFFFA (BRK vector base)

ALU modes

Signal	Description
SU	ALU Subtract mode
AND	ALU AND mode
OR	ALU OR mode
XOR	ALU XOR mode
NOT	ALU NOT mode
INC	ALU Increment mode
DEC	ALU Decrement mode
SHL	ALU Shift Left mode
SHR	ALU Shift Right mode

Signal	Description
`ROL`	ALU Rotate Left through carry
`ROR`	ALU Rotate Right through carry
`CMP`	ALU Compare mode (flags only, result discarded)
`MUL`	ALU Multiply mode - A×B, low byte out, high byte to EXO
`DIV`	ALU Divide mode - A÷B, quotient out, remainder to EXO
`AHC`	ALU := TMP + addrCarry (high-byte address adjust)
`AHB`	ALU := TMP - addrCarry (high-byte borrow adjust)

Side effects

Signal	Description
`CFC`	Clear Carry flag (CLC)
`CFS`	Set Carry flag (SEC)
`EIF`	Enable Interrupt flag (EI)
`DIF`	Disable Interrupt flag (DI)
`HLT`	Halt - stop until an enabled interrupt arrives
`RND`	Random byte into A (seedable xorshift32)

Standalone microword bits

Signal	Description
`JPW`	PC := OPH:OP (16-bit address path - absolute jump)
`CE`	Program Counter Enable - PC := PC + 1
`SD`	Stack Pointer Decrement (16-bit)
`SE`	Stack Pointer Increment (16-bit)
`CI`	ALU carry-in from the C flag (ADC/SBC)
`ATM`	ALU input mux - TMP instead of A
`AOP`	ALU input mux - operand register instead of B
`ACO`	Latch ALU carry/borrow-out into addrCarry (flags untouched)
`FI`	Flags In - latch flags (ALU flags with EO, Z/N of the bus otherwise)

The ALU and its two multiplexers

The arithmetic-logic unit has two inputs. By default the first comes from **A** and the second from **B** — which is why **B** is “the other operand” and why every binary operation overwrites it.

Two signals rewire that:

ATM take the first input from **TMP** instead of **A**,
AOP take the second input from the operand register instead of **B**.

Without those two multiplexers the effective-address calculation would have to pass through the accumulator — and every `LDA table,X` would silently destroy **A**. Thanks to them, indexing is computed aside, in registers the program cannot see.

When `EO` is asserted with no mode signal, the ALU adds. Addition is therefore the default operation, not a special case.

The carry between address bytes

A 16-bit sum on an 8-bit ALU takes two steps, and the carry has to be kept somewhere between them. That is the job of the one-bit **addrCarry** latch and three signals:

Signal	What it does
`ACO`	store the carry (or borrow) of the operation just performed into **addrCarry** flags untouched
`AHC`	ALU := TMP + addrCarry — add the carry into the high byte
`AHB`	ALU := TMP − addrCarry — the same for subtraction

The typical pair of steps from an `abs,X` calculation looks like this:

```
[ATM,AOP,EO,MI,ACO]     low byte: TMP + operand -> MAR, carry
latched
[ATM,AHC,EO,MIH]        high byte: TMP + addrCarry -> MAR high
half
```

> **Note**
>
> `ACO` is deliberately separate from `FI`. If address arithmetic wrote the `C` flag, you could not put `LDA table,X` between a compare and its conditional jump — and that is a pattern real code uses constantly.

When flags are written

Writing the flags is the job of a single signal, `FI`, and its behaviour depends on whether `EO` is asserted in the same T-state:

with EO the flags come from the ALU according to the operation performed — and only those the operation **defines** are written.

without EO `Z` and `N` are derived from the byte crossing the bus. This is how register transfers (`TAX`, `TYA`, …) and stack pops set flags.

The flag sets defined by each ALU mode:

Mode	Flags
`ADD`	`C Z N V`
`SU`	`C Z N V`
`AND`	`Z N`
`OR`	`Z N`
`XOR`	`Z N`
`NOT`	`Z N`
`INC`	`Z N`
`DEC`	`Z N`
`SHL`	`C Z N`
`SHR`	`C Z N`
`ROL`	`C Z N`
`ROR`	`C Z N`
`CMP`	`C Z N`
`MUL`	`C Z N`
`DIV`	`C Z N`

Figure 4. Which flags each ALU mode defines. Anything not in a row keeps its previous value.

> **Pitfall**
>
> `DIV` shows `C`, `Z` and `N` in that table — but only for a successful division. Dividing by zero defines `C` alone, so `Z` and `N` keep their values from the previous instruction. It is the only place in the ISA where the set of defined flags depends on the operands.

Ordering within one T-state

The signals in a T-state are not simultaneous — they have a documented order, and the correctness of some sequences rests on it:

SP adjust ⟶ address-path transfer ⟶ data-bus source ⟶ data-bus sinks ⟶ ALU latch (`ACO`) ⟶ PC increment (`CE`)

The clearest consequence is a stack push. The step `[SD,SPM]` first lowers **SP** and **then** sends it to **MAR** — which is why a push is pre-decrement and why **SP** always points at the byte most recently written.

The second consequence is subtler and concerns interrupt entry: in the step `[FLO,RI,DIF]` the status byte reaches the bus **before** `DIF` disables interrupts. The frame therefore stores the state with `I` still set — and `RTI` consequently re-enables interrupts on return without having to do so explicitly.

CHAPTER 5

The Instruction Cycle and T-States

A T-state is the smallest unit of time in this machine: one clock tick, one microword, one set of gates held open. An instruction is a sequence of T-states and nothing more — there is no further layer where something happens "in between".

The T-state counts in this book are totals, that is, they **include** fetching the instruction. When `LDA zp` is listed at 6 T, six ticks elapse between the instruction's turn coming round and the value being in the accumulator.

Fetch

Fetching an instruction has three variants, depending on how many operand bytes it has. They always look the same and always take the same time:

Template	T	Steps
`F1`	2	`[PCM] [RO,II,CE]`
`F2`	4	`F1` + `[PCM] [RO,ORI,CE]`
`F3`	6	`F2` + `[PCM] [RO,OHI,CE]`

Each pair of steps does the same thing: send **PC** to **MAR**, read a byte from memory, store it in the right register and increment **PC**. The first byte goes to the instruction register, the second to **OP**, the third to **OPH**.

One-byte instructions use `F1`, three-byte ones `F3`, two-byte ones `F2` — and instructions with an absolute address use `F3`, because their address is two bytes.

> **Note**
>
> Fetch is not overhead you could hide. For short instructions it is most of their time: `LDA #n` takes 5 T and four of them are fetch. The performance chapter calls this the fetch tax, and it is the main reason code density matters more on this machine than register count.

Building the effective address

Between fetch and the actual work, memory instructions spend steps computing which address will be touched. Their number is the entire difference between addressing modes:

Mode	Extra T	What happens
`#n`	0	the operand **is** the value; nothing is touched
`zp`	1	the operand byte goes straight into **MAR** (zero-extended)
`abs`	2	low and high address bytes into **MAR**
`zp,X`	2	**X** into **TMP**, sum with the operand into **MAR** carry discarded
`abs,X` / `abs,Y`	4	byte-wise sum with the carry through **addrCarry**
`(zp)`	5	read a two-byte pointer from the zero page
`(zp),Y`	8	read the pointer and add **Y** to it

A worked example: `LDA 0x50`

The simplest memory instruction, T-state by T-state:

T	Signal
`0`	`PCM`
`1`	`RO, II, CE`
`2`	`PCM`
`3`	`RO, ORI, CE`
`4`	`ORO, MI`
`5`	`RO, AI, FI`

Figure 5. The microprogram of `LDA zp` (opcode 0x41), 6 T-states.

T0 **PC** travels the address path into **MAR**.

T1 the opcode is read from memory into **IR** **PC** increments.

T2 **PC** into **MAR** again — this time pointing at the operand.

T3 the operand byte goes into **OP**, **PC** increments. Fetch ends here.

T4 the operand drives the bus and `MI` turns it into the address 0x00:`0x50`.

T5 the byte at that address is read into **A** and `FI` derives `Z` and `N`.

Note that in step T4 `MI` behaves differently from `MIL`: it writes the whole **zero-page** address, clearing the high half as it goes. That is precisely why the zero page is cheap — it needs no second byte.

Handing the fetch between instructions

This is the detail that catches out everyone reading the microcode for the first time.

The first two steps of every instruction are the fetch, but **it is not the one executing them**. The instruction register is not rewritten until step T1 — so while the processor performs T0 and T1 it is still running the microprogram of the **previous** instruction.

It works because every microprogram has those two steps identical. And because that must not break, a rule holds: no microcode array may be shorter than three steps. Hence the single oddity of `NOP` — it carries one empty step after the fetch to satisfy the rule:

```
[PCM] [RO,II,CE] []  (3)
```

> **Pitfall**
>
> The practical consequence: `NOP` takes 3 T, not 2. Anyone counting delay loops in T-states has to allow for it — and anyone wanting the cheapest possible stall reaches for `NOP` in a `%rep`, because no shorter instruction exists.

Conditional jumps

Conditional jumps carry a condition on the `JPW` signal in their final step. When the condition fails, the whole step is skipped — but **no time is saved**. A jump takes 7 T whether or not it was taken.

This is a deliberate trade in favour of determinism: a program's running time does not depend on its data, which is the property reference traces for comparing the emulator against the FPGA rely on. It also simplifies counting time in loops — you do not need to know how often a branch was taken.

Longer instructions

The machine's longest instructions take 14 T (the ALU forms with the `(zp),Y` mode). `INW abs`, `DEW abs`, `CALL (zp)`, `ADW` and `SBW` take 13 T.

No instruction is padded to a round number. An instruction takes exactly as long as its work takes — and the tools show those real, varying figures. It looks less tidy than a machine with a fixed cycle length, but it is honest and it is measurable.

CHAPTER 6

Interrupts, RESET and BRK

An interrupt is the one place where the processor does something the program does not contain. It is therefore worth knowing exactly what it does — and what it leaves to the programmer.

Boot

Reset does three things: clears the registers, sets **SP** to 0x0200 and clears the `I` flag. Then it injects into the instruction register the pseudo-opcode 0xFD, which cannot be fetched from memory and whose microprogram is this:

```
[VRS] [RO,JPL] [MRI] [RO,JPH]  (4)
```

Four steps, two of them ordinary memory reads. `VRS` points **MAR** at 0xFFFC, the low address byte is read into the low half of **PC**, `MRI` advances **MAR** by one and the high byte is read.

That is the entire boot. There is no ROM and no hidden state — two reads you can single-step and watch on the schematic.

> **Note**
>
> Pseudo-opcodes have no fetch. They are not read from memory, so there would be nothing to read them from — the hardware places them directly into **IR**. Their microprogram therefore starts straight at the work.

The assembler fills the RESET vector automatically with the address of the first emitted instruction, so an ordinary program need not know about boot at all.

Sources and masking

Interrupts have three hardware sources and two levels of enabling.

Individual sources are enabled in `IRQ_ENABLE` (0xFF40):

Bit	Source
0	timer

Bit	Source
1	keyboard — an unread character is waiting
2	serial line — the receive FIFO is not empty

The **master switch** is the `I` flag, controlled by `EI` and `DI`. While `I` is clear the processor takes no interrupt, however many sources are enabled.

`IRQ_STATUS` (0xFF41) shows what is currently pending; reading it clears it.

Interrupt entry

An interrupt is only taken at an instruction boundary — never in the middle. When `I` is set at a boundary and an enabled source is waiting, the processor injects the pseudo-opcode 0xFE:

```
[SD,SPM] [CHO,RI] [SD,SPM] [CLO,RI] [SD,SPM] [FLO,RI,DIF] [VIR]
[RO,JPL] [MRI] [RO,JPH]  (10)
```

Ten T-states, in three phases:

1. Three pushes: the high byte of **PC**, the low byte of **PC**, the status byte. In that order — so the status byte ends up on top.
2. The `DIF` signal, in the same step as `FLO`, disables interrupts.
3. `VIR` points **MAR** at 0xFFFE and the same pair of reads as at reset fills **PC**.

The frame on the stack looks like this (addresses decrease downwards):

`RTI` takes the frame off in the opposite order: the status byte first, then the low and high bytes of **PC**.

> **Note**
>
> `RTI` does not enable interrupts outright — it only restores the status byte. That interrupts end up enabled after a return follows from the ordering within a T-state: the saved status still has `I` set, because `DIF` takes effect only after the status byte has left the bus.

> The difference shows with nested interrupts: a handler that clears `I` with `DI` halfway through still returns to an enabled state, because what is restored is what held **before** the interrupt.

What the handler must do itself

The hardware saves **PC** and the flags. Nothing more. Every other register a handler uses it must save itself.

The most treacherous is **B**. Leaving it alone is not enough — every binary ALU operation overwrites it, including an innocent-looking `CMP #0`. A handler that touches the ALU anywhere must save **B**, or it silently destroys a value the interrupted code had staged with `TAB`.

```
handler:
        PUSH            ; save A
        PHB             ; save B
        PHX             ; save X if you use it
        ; ... handler body ...
        PLX
        PLB             ; restore B
        POP             ; restore A
        RTI
```

The hidden **TMP** and **addrCarry** need no saving — they carry no state across an instruction boundary, and a boundary is the only place an interrupt is taken.

Subroutines may be called: `CALL` and `RET` use the same stack, and an interrupt handler is in no way exceptional.

`HLT` is wait-for-interrupt

`HLT` stops the processor, but not necessarily forever. When an enabled interrupt arrives the processor wakes and the handler runs. A true stop happens only when `I` is clear or no source is enabled.

Hence the typical skeleton of an event-driven program:

```
        EI
loop:   HLT             ; sleep until something happens
        JMP loop
```

`BRK` — the software interrupt

`BRK` (0xCD) is an ordinary fetched instruction that pushes the same frame as a hardware interrupt — but it is triggered by the program, applies **regardless of the `I` flag**, and vectors through its own vector at 0xFFFA.

```
[PCM] [RO,II,CE] [SD,SPM] [CHO,RI] [SD,SPM] [CLO,RI] [SD,SPM]
[FLO,RI,DIF] [VBR] [RO,JPL] [MRI] [RO,JPH]  (12)
```

Twelve T-states: two more than hardware entry, because unlike a pseudo-opcode `BRK` has to be fetched from memory.

Having its own vector means a `BRK` handler is a different routine from the hardware interrupt handler. That makes `BRK` useful for two things: a system call (the service number passed in a register by convention) and a debugger breakpoint — one byte overwritten with `BRK` stops the program anywhere.

Return is by `RTI` and the program continues at the byte **after** `BRK`.

Pitfall

`BRK` cannot be masked. `DI` has no effect on it — it is an instruction, not an event. A program that has interrupts disabled and assumes it cannot reach a handler is wrong if it runs into a `BRK`.

PART II

The Programming Model

CHAPTER 7

Registers and Flags

A SAP-3/16 programmer sees six registers and five flags. That is few — and it is deliberate: the fewer pieces of state, the more easily the whole machine fits in your head. The price is that everything has to flow through that small set of registers, which is why this chapter has more pitfalls than any other.

The registers a program sees

Register	Width	Role
A	8 b	accumulator — target of nearly all arithmetic, the only register that supports all eight addressing modes
B	8 b	the ALU's second operand; **every binary operation overwrites it**
X	8 b	index, the main one for `mem,X`
Y	8 b	index, carries `(zp),Y` and `mem,Y`
SP	16 b	stack pointer, 0x0200 after reset
PC	16 b	program counter
OUT	8 b	the output port of the `OUT` instruction

None of them is redundant — which sounds like a platitude but can be measured. Across every program in the repository **X** carries over a thousand indexed accesses, **Y** roughly a third as many, and the `TAB` / `TBA` pair together with `PHB` / `PLB` make up almost seven percent of all instructions actually executed. **OUT** is a teaching inheritance from SAP-1 and is used over a hundred and twenty times.

Register B is not like the others

B is not a scratch register. It is the **ALU's input**, and the microcode therefore fills it every time arithmetic is touched:

```
        LDA divisor
        TAB             ; B := divisor
        CMP #0          ; <- B is gone here, overwritten by zero
```

```
        LDA dividend
        DIV             ; divides by zero, not by the divisor
```

The list of instructions that overwrite **B** is simply "every binary ALU operation": `ADD`, `ADC`, `SUB`, `SBC`, `AND`, `OR`, `XOR`, `CMP`, `BIT`, `CPX`, `CPY`, `ADW` and `SBW`.

> **Pitfall**
>
> The most treacherous item on that list is `CMP`. It looks non-destructive ("I am only comparing"), and towards **A** it is — but the operand has to land somewhere, and that somewhere is **B**.
>
> In practice: between a `TAB` and its `MUL` / `DIV` there must be **nothing** that touches the ALU. If you need a value in **B** to survive further, push it with `PHB`.

Hidden registers

These have no instruction that can address them, but without them the microcode cannot be read:

Register	Width	Purpose
IR	8 b	the opcode currently executing
OP	8 b	the first operand byte
OPH	8 b	the second operand byte
MAR	16 b	the address memory is being touched at
TMP	8 b	the ALU's working input for address maths and index operations
addrCarry	1 b	the carry between address bytes

The last two are worth attention because they solve a concrete problem: if the effective address were computed through **A** and the `C` flag, every `LDA table,X` would destroy both the accumulator and the result of the previous compare. Thanks to **TMP** and **addrCarry**, address arithmetic happens entirely out of the program's reach.

> **Note**

> The hidden registers carry no state across an instruction boundary — and a boundary is the only place an interrupt is taken. An interrupt handler therefore never needs to save them.

The flags

Five one-bit flags:

Flag	Bit	Meaning
`C`	0	carry on addition, **borrow** on subtraction, the shifted-out bit on shifts
`Z`	1	the result was zero
`N`	2	the top bit of the result (signed: negative)
`V`	3	signed overflow
`I`	4	interrupts enabled

The Bit column is the position in the status byte as pushed by interrupt entry, by `BRK` and by `PHF`. The other three bits are zero.

Only what an operation defines gets written

This is the rule half the book turns on: an instruction sets **only those flags that make sense for its operation**. The rest keep their previous value.

It is not sloppiness but a feature. Thanks to it you can write:

```
        CMP #10         ; sets C, Z, N
        AND #0x0F       ; sets Z, N — C survives
        JC  smaller     ; and the compare still applies
```

The complete split of every mnemonic by what it writes:

Flags	Instruction
none	`ADW` , `CALL` , `DEW` , `HLT` , `INW` , `JC` , `JGES` , `JGT` , `JLE` , `JLTS` , `JMP` , `JN` , `JNC` , `JNV` , `JNZ` , `JP` , `JV` , `JZ` , `LXY` , `NOP` , `OUT` , `PHB` , `PHF` , `PHX` , `PHY` , `PUSH` , `RET` , `RND` , `SBW` , `STA` , `STX` , `STY` , `SXY` , `TSXY` , `TXYS`
`C`	`CLC` , `SEC`
`I`	`BRK` , `DI` , `EI`
`Z N`	`AND` , `BIT` , `DEC` , `DEX` , `DEY` , `INC` , `INX` , `INY` , `LDA` , `LDX` , `LDY` , `NOT` , `OR` , `PLB` , `PLX` , `PLY` , `POP` , `TAB` , `TAX` , `TAY` , `TBA` , `TXA` , `TYA` , `XOR`
`C Z N`	`CMP` , `CPX` , `CPY` , `DIV` , `MUL` , `ROL` , `ROR` , `SHL` , `SHR`
`C Z N V`	`ADC` , `ADD` , `SBC` , `SUB`
`C Z N V I`	`PLF` , `RTI`

Figure 6. Which instructions write which flags. Anything not listed in a row keeps its previous value.

The first row is the one to read: instructions that change nothing. They are almost entirely operations on addresses and pointers (`INW` , `DEW` , `ADW` , `SBW` , `LXY` , `SXY` , `TXYS`) plus control flow. Counting with an address should not destroy the decision a program is in the middle of making.

At the other end sit `RTI` and `PLF` , which restore all five flags at once because they load a whole status byte.

What determines the flags of ALU operations

Mode	Flags
`ADD`	`C Z N V`
`SU`	`C Z N V`
`AND`	`Z N`
`OR`	`Z N`
`XOR`	`Z N`
`NOT`	`Z N`
`INC`	`Z N`
`DEC`	`Z N`
`SHL`	`C Z N`
`SHR`	`C Z N`
`ROL`	`C Z N`
`ROR`	`C Z N`
`CMP`	`C Z N`
`MUL`	`C Z N`
`DIV`	`C Z N`

Figure 7. The flag set defined by each ALU mode.

Two things from that table are worth memorising: logic operations leave `C` alone (ISA 2.0 still cleared it) and `CMP` leaves `V` alone. The second has a consequence that catches everyone — see below.

The borrow convention

After a subtraction `C = 1` means "the subtraction underflowed", that is `A < operand`. This is the opposite of the 6502, where `C` is set on "no borrow".

So that nobody has to remember it, the assembler provides aliases named the way people think:

Written	Real instruction	After `CMP` means
`JEQ`	`JZ`	equal
`JNE`	`JNZ`	not equal
`JLT`	`JC`	less than
`JGE`	`JNC`	greater or equal

A practical consequence of the convention: `SUB` and `SBC` fit together with no preparation. The lowest byte is subtracted with `SUB` (which ignores the incoming borrow),

the higher ones with `SBC`. That is why on this machine — unlike the 6502 — you never write `SEC` before a subtraction or `CLC` before an addition.

Unsigned and signed

The byte `0xFF` is 255, or −1. The processor does not distinguish; the jump the programmer picks does.

Comparison	Jumps
unsigned	`JLT` / `JC`, `JGE` / `JNC`, `JGT`, `JLE`
signed	`JLTS`, `JGES`

A signed comparison is governed by the condition `N ⊕ V`: a negative result **or** an overflow that flipped the sign.

Pitfall

And here is the trap: `CMP` does not set `V`. Signed jumps after it therefore read a `V` left over from some earlier instruction and can answer completely wrongly — silently, with no assembly error.

A signed comparison pairs with `SUB` or `SBC`, which do set `V`:

```
        LDA #-100       ; 0x9C
        SUB #100        ; overflows: N=0, V=1
        JLTS smaller    ; taken — and correctly so
```

The price is that `SUB`, unlike `CMP`, overwrites the accumulator.

For completeness, why `JN` alone is not enough: for `-1 < 1` the subtraction gives `0xFF − 0x01 = 0xFE`, so `N = 1` and `V = 0`, and `JN` answers correctly. But for `-100 < 100` the subtraction overflows, `N` comes out 0 and `JN` says −100 is not smaller. The condition `N ⊕ V` covers that case; `N` alone does not.

When flags are written

At the microcode level a single signal writes the flags, and its behaviour depends on whether the ALU output is asserted in the same T-state:

with ALU output the flags the operation performed defines are written.
without ALU output `Z` and `N` are derived from the byte currently on the data bus.

The second case is why instructions that "compute nothing" — `LDA`, `TAX`, `POP` and friends — still set flags. In practice it saves instructions:

```
        LDA state
        JZ  nothing     ; no need to write CMP #0
```

Measurement bears this out: `CMP #0` immediately after a load occurs three times in the entire corpus of programs. Everything else relies on `Z` and `N` from the load — correctly.

CHAPTER 8

Addressing Modes

An addressing mode answers the question "where is the operand". SAP-3/16 has eight of them and, unlike most 8-bits, has them **systematically**: the mode is the `mmm` field of the opcode and its value means the same thing in every instruction that supports it.

Mode	mmm	Syntax	Operand (B)	Effective address
`IMM`	`000`	`#n`	1	`—`
`ZP`	`001`	`zp`	1	`0x00:zp`
`ABS`	`010`	`abs`	2	`abs`
`ABS_X`	`011`	`abs,X`	2	`abs + X`
`ABS_Y`	`100`	`abs,Y`	2	`abs + Y`
`IND`	`101`	`(zp)`	1	`[zp, zp+1]`
`ZP_X`	`110`	`zp,X`	1	`0x00:((zp + X) & 0xFF)`
`IND_Y`	`111`	`(zp),Y`	1	`[zp, zp+1] + Y`

Figure 8. The eight addressing modes, their encoding and effective-address calculation.

The modes one by one

Immediate — `#n`

The value is part of the instruction. Nothing is touched, which makes it the fastest way to get a constant into a register.

> **Pitfall**
>
> The hash is mandatory and leaving it out is **not an assembly error**. `LDA 5` means "load the contents of address 5", `LDA #5` means "load the number 5". Both are legal, so the typo assembles and only shows up at run time.
>
> The older ISA had dedicated mnemonics for immediates (`LDI`, `ANI`, `CPI` …) which made the mistake impossible. ISA 3.0 dropped them in favour of regularity — this pitfall is the price.

Zero page — `zp`

A one-byte address; the high byte is implicitly zero. One byte shorter and three T-states faster than an absolute address.

Absolute — `abs`

A full 16-bit address. It reaches anywhere.

Indexed absolute — `abs,X` and `abs,Y`

An 8-bit index is added to a 16-bit base. The sum is a **full 16-bit** one, so crossing a page boundary works.

> **Note**
>
> Crossing a 256-byte boundary costs nothing extra. On a 6502 an indexed access across a page boundary costs an extra cycle; here the sum is always two-stepped, so the cost is constant — part of the promise that running time does not depend on data.

Indirect — `(zp)`

A two-byte pointer sits in the zero page (low byte first) and the operand is wherever it points.

Zero-page indexed — `zp,X`

The address is `(zp + X) & 0xFF` — the sum **wraps inside the zero page**. The carry is discarded.

> **Pitfall**
>
> The wrap is deliberate (inherited from the 6502) but surprising. `LDA 0xF0,X` with `X = 0x20` does not read from `0x0110` but from `0x0010`. Small tables in the zero page must therefore not be placed where they would run off its end.

Indirect indexed — `(zp),Y`

The most powerful and most expensive mode. The pointer is read from the zero page and only then is **Y** added to it. This is **post-indexing**: the index is added to the loaded address, not to the pointer's address.

It is how you walk a buffer whose address is computed at run time:

```
        LDY #0
.next:  LDA (source),Y
        STA (dest),Y
        INY
        CPY #LENGTH
        JNE .next
```

> **Note**
>
> The pointer is read with full 16-bit arithmetic, so a pointer at address `0xFF` reads its high byte from 0x0100 — the stack page, not the start of the zero page. It is not a wrap like `zp,X`.

Which instructions support which modes

Instruction	`#n`	`zp`	`abs`	`abs,X`	`abs,Y`	`(zp)`	`zp,X`	`(zp),Y`
`LDA`	•	•	•	•	•	•	•	•
`LDX`	•	•	•	—	•	—	—	—
`LDY`	•	•	•	•	—	—	•	—
`STA`	—	•	•	•	•	•	•	•
`STX`	—	•	•	—	—	—	—	—
`STY`	—	•	•	—	—	—	—	—
`CPX`	•	•	•	—	—	—	—	—
`CPY`	•	•	•	—	—	—	—	—
`ADD`	•	•	•	•	•	•	•	•
`ADC`	•	•	•	•	•	•	•	•
`SUB`	•	•	•	•	•	•	•	•
`SBC`	•	•	•	•	•	•	•	•
`AND`	•	•	•	•	•	•	•	•
`OR`	•	•	•	•	•	•	•	•
`XOR`	•	•	•	•	•	•	•	•
`CMP`	•	•	•	•	•	•	•	•

Figure 9. Legal combinations in groups 01 and 10.

The pattern has logic:

- **A** supports everything. It is the accumulator and everything revolves around it.
- `STA` has no immediate — storing into a constant makes no sense.
- `LDX` indexes with **Y** and `LDY` with **X**. Indexing a register by itself would be useless.
- `STX` and `STY` have only `zp` and `abs`.

Four modes that do not exist

The `mmm` field has eight values and all eight are taken. That is why the ISA has neither `zp,Y` nor `(zp,X)` — there was no room left in the encoding.

Pitfall

The absence of `zp,Y` has a concrete consequence: `STX table,Y` cannot be written. An indexed store of an index register has to go through the accumulator:

```
        TXA
        STA table,Y     ; A is gone as a result
```

What the modes cost

The differences are large enough to shape a program's design. Using `LDA` as the example:

Mode	Bytes	T	Where the time goes
`#n`	2	5	the operand is the value; nothing is touched
`zp`	2	6	one address byte
`abs`	3	9	two address bytes
`abs,X`	3	11	byte-wise sum with carry
`abs,Y`	3	11	byte-wise sum with carry
`(zp)`	2	10	reading a two-byte pointer
`zp,X`	2	7	index addition, carry discarded
`(zp),Y`	2	13	reading the pointer plus a 16-bit sum

Table 2. What each addressing mode costs, using `LDA` as the example.

The heuristic that follows: **whatever can live in the zero page, should**. The difference between `zp` and `abs` is three T-states and one byte per access, and in a tight loop that adds up to tens of percent.

Automatic size selection

The assembler picks between `zp` and `abs` itself, from the operand's value. It does so iteratively: it starts with short forms and widens them until the sizes settle. Sizes may only grow, so the process always terminates.

The same holds for `sym,X` — it assembles as `zp,X` if the base fits in a byte, otherwise as `abs,X`. By contrast `sym,Y` is **always** `abs,Y`, because `zp,Y` does not exist.

The choice can be forced with a suffix:

Written	Meaning
`LDA.B sym`	force zero page; an error if `sym > 0xFF`
`LDA.W 0x42`	force an absolute address even for a small value
`LDA.W sym,X`	force `abs,X` even for a zero-page base

> **Note**
>
> Forcing is useful when generated code refers to an address, or when a program relies on an instruction's **length** — for instance in a jump table, where every entry must be the same size.

CHAPTER 9

The Stack and Subroutines

The stack in SAP-3/16 is not special memory. It is ordinary RAM pointed at by **SP** — and everything that makes it a stack is how a handful of instructions treat that pointer.

How it works

SP resets to 0x0200, that is just **past** the end of the page 0x0100–0x01FF.

A push is pre-decrement: **SP** is lowered first, then the byte is written to the new address. The first pushed byte therefore lands at 0x01FF and **SP** always points directly at the newest item. A pop is the reverse — read, then increment.

```
0x01FF   first pushed byte
0x01FE         second
0x01FD          third          ← SP
   …             free
0x0100                         end of the page
```

The stack therefore grows towards lower addresses — into the zero page.

The instructions

Push	Pop	What
`PUSH`	`POP`	the accumulator
`PHX`	`PLX`	register X
`PHY`	`PLY`	register Y
`PHB`	`PLB`	register B
`PHF`	`PLF`	the status byte (flags)

Pushes do not change the flags. Pops set `Z` and `N` from the popped value — except `PLF`, which restores all five flags at once.

Each of these takes 4 T-states.

Calling subroutines

`CALL` pushes the return address — high byte first, then low — and jumps. `RET` pops them in the opposite order and returns.

```
        CALL print
        HLT

print:  LDA #'A'
        STA 0xFF10
        RET
```

The return address points at the instruction **after** the call. It occupies two bytes on the stack, so the stack page holds 128 nested calls — provided nothing else is pushed.

There is also an indirect form, `CALL (zp)`, which takes the target from a pointer in the zero page. It is how you build a callback or a function table:

```
        LDA #<handler
        STA vector
        LDA #>handler
        STA vector+1
        CALL (vector)
```

Pitfall

Nobody checks for stack overflow. When **SP** drops below 0x0100 the stack starts overwriting the zero page — typically a program's pointers and variables. It shows up as corrupted data somewhere else entirely, not as a crash.

That is exactly why the default placement is useful: it overflows into the zero page, where the damage usually surfaces quickly and conspicuously.

Passing parameters

The hardware enforces no convention. In practice three patterns are used, each for a different situation.

In registers

Fastest and most common. Three bytes cover almost anything a subroutine needs.

```
        LDA #10         ; x
        LDX #20         ; y
        CALL plot
```

In the zero page

For more parameters or for 16-bit values. The subroutine reads them from fixed addresses that are part of its interface.

```
        LXY #text
        SXY ptr         ; ptr := address of the string
        CALL print_text
```

The drawback: such a subroutine is not re-entrant. If an interrupt handler calls it in the middle of its own run, it overwrites its own parameters.

On the stack

The most general and most expensive. Parameters are pushed before the call and the subroutine reads them through **SP**, which `TSXY` makes available.

```
        LDA value
        PUSH
        CALL process
        POP             ; clean up the parameter
```

> **Note**
>
> The caller cleans up the parameters. The subroutine cannot, because its own return address lies underneath them.

Working with the stack pointer

The `TXYS` and `TSXY` pair moves **SP** through the **X**:**Y** pair, where **X** holds the high half.

```
        LXY #0x0200
        TXYS            ; SP := 0x0200 (the reset value)
```

Typical uses: initialisation at startup, and "unwinding" the stack when recovering from an error, where setting the pointer outright is simpler than returning from every call.

> **Note**

The transfer takes two T-states, because a 16-bit value crosses an 8-bit bus in two goes. Interrupts are only taken at an instruction boundary, though, so a handler never sees **SP** half-updated.

Recursion

Recursion works; you just have to budget the depth. Each level costs two bytes of return address plus whatever the subroutine pushes.

```
; factorial(A) into A, recursively
fact:   CMP #2
        JLT .base
        PUSH            ; save n
        DEC
        CALL fact       ; A := (n-1)!
        TAB             ; B := (n-1)!
        POP             ; A := n
        MUL             ; A := n * (n-1)!
        RET
.base:  LDA #1
        RET
```

With 128 levels available and two extra bytes per level, a safe depth lands somewhere around forty.

The stack and interrupts

Interrupts use the same stack. Entry pushes three bytes (the high byte of **PC**, the low byte of **PC**, the status byte), so a handler starts three bytes deeper than the code it interrupted.

Two things follow:

1. Handlers may call subroutines — the stack is shared and nothing special happens.
2. The handler has to be counted in the depth budget. A program running near the limit will overflow at the moment an interrupt arrives.

Pitfall

A handler must leave the stack balanced. Whatever it pushes it must pop — otherwise `RTI` takes something else as the return address. It is a silent fault that surfaces as a jump into nowhere on the **next** interrupt.

CHAPTER 10

The Assembly Language

The SAP-3/16 assembler is deliberately small. It has no sections, no modules and no linker — it translates source text straight into a 64 KB memory image. Everything it can do fits in one chapter.

The shape of a line

```
[label:]  MNEMONIC  [operand]   ; comment
```

All four parts are optional; a blank line or a comment-only line is fine. A label ends with a colon, a comment starts with a semicolon and runs to the end of the line.

The assembler is case-insensitive for mnemonics and registers. Labels and constants are case-sensitive.

Numbers

Written	Base	Note
`0x2A`	16	the usual form
`$2A`	16	the 6502-style form — the disassembler prints addresses this way
`0b101010`	2	handy for masks and bit fields
`42`	10	
`-1`	10	negative numbers are stored in two's complement

Character literals are written in single quotes and know the usual escapes:

```
        LDA #'A'
        LDA #'\n'       ; 0x0A
```

The supported sequences are `\n`, `\r`, `\t`, `\b`, `\f`, `\\`, `\"`, `\'` and `\0`. Anything else after a backslash is an error.

Expressions

An expression is a number, a character literal, a constant, a label, or a label with an offset:

```
        LDA table+2
        LDA end-1
```

An expression handles no more than one operator — this is not full arithmetic but a deliberately simple mechanism for addressing elements.

Byte selection

The `#<` and `#>` operators split a 16-bit value into bytes (low and high respectively). They appear wherever a pointer is filled in:

```
        LDA #<handler   ; low byte of the address
        STA vector
        LDA #>handler   ; high byte
        STA vector+1
```

> **Note**
>
> Since ISA 3.1 the same thing can be written with the shorter `LXY` / `SXY` pair, which moves the whole address at once. Byte selection remains useful where the bytes are handled separately.

Labels

A label is a name for an address. A global label starts with a letter or an underscore; a local one starts with a dot and is valid only inside the block delimited by the nearest preceding global label.

```
draw:   LDX #0
.loop:  STA (ptr),Y
        INX
        CPX #40
        JNE .loop       ; refers to this block's .loop
        RET

clear:  LDX #0
.loop:  STA (ptr),Y     ; a different .loop, no clash
```

```
        INX
        CPX #40
        JNE .loop
        RET
```

Local labels are why this language never needs names like `loop_draw_2`.

Directives

.ORG
Sets the address being assembled to. The default is 0x0200.

```
        .ORG 0x0300
```

The operand must be a constant expression — labels are not allowed, because their addresses would depend on the `.ORG`.

EQU
Names a constant. It is written **without a colon**, the one place the syntax differs from a label:

```
VRAM    EQU 0x8000
ptr     EQU 0x10
LENGTH  EQU 40
```

The value must be a constant expression in 16-bit range. A constant and a label may not share a name.

DATA
Emits bytes at the current address. It takes numbers and strings:

```
message: DATA "Hello", 0x0A, 0
table:   DATA 1, 2, 4, 8, 16, 32, 64, 128
```

.WORD *(**DW**)*
Emits 16-bit words, low byte first:

```
jumps:  .WORD start, finish, error
```

.VECTOR
Writes an address into one of the three vectors at the top of memory:

```
        .VECTOR RESET, start
        .VECTOR IRQ, handler
        .VECTOR BRK, syscall
```

The RESET vector is filled automatically with the address of the first emitted instruction, so an ordinary program need not write it.

Forcing the size

The `.B` and `.W` suffixes force zero-page or absolute encoding:

```
        LDA.B ptr       ; force zp (an error if it does not fit)
        LDA.W 0x42      ; force abs even for a small address
```

Without a suffix the assembler chooses for itself, iterating until the sizes settle.

What assembly produces

Besides the memory image the assembler returns the data the tools live on: a listing with addresses and bytes, a symbol table, an address-to-source-line map, lists of instruction, data and operand addresses, and the program's entry point.

The address-to-line map is what lets the debugger show which source line the program is on — including inside an expanded macro.

Assembly errors

Error messages carry the line number, and for code produced by a macro also the expansion context (`Line 12 (in macro DELAY expanded at line 30)`).

Structure and symbols

Message
`Unknown instruction: XYZ`
`Unknown directive: .XYZ`
`Invalid label name: …`
`Duplicate label: …`
`Duplicate constant: …`
`Label … conflicts with an EQU constant`
`Undefined label: …`
`Invalid expression: …`

Operands and addressing modes

Message
`Instruction XYZ requires an operand`
`XYZ does not support immediate mode`
`XYZ does not support zero-page mode`
`XYZ does not support absolute mode`
`XYZ does not support (zp) indirect mode`
`XYZ does not support indexed addressing with X` (resp. `Y`)
`XYZ does not support (zp),Y indexed-indirect mode`
`Indirect-indexed addressing is only (zp),Y`
`XYZ requires a memory operand`
`XYZ requires an absolute target address`
`XYZ requires two operands: XYZ zp, #imm`
`Missing value after #`
`Empty indirect operand ()`
`Invalid operand for XYZ`

Directives and ranges

Message
`EQU directive requires a name` / `… requires a value`
`EQU name cannot be a local label`
`EQU value must be a constant expression (labels are not allowed): …`
`.ORG requires an address` / `.ORG address must be a constant expression: …`
`.ORG address … out of range (0x0000-0xFFFF)`
`.VECTOR requires: .VECTOR RESET|IRQ|BRK, target`
`Unknown vector: … (expected RESET, IRQ or BRK)`
`DATA requires at least one value` / `.WORD requires at least one value`
`Value … out of byte range (-128 to 255)`
`Address … is out of range (0x0000-0xFFFF)`
`Program overflows past 0xFFFF (at 0x….)`
`Malformed character literal: …` / `Unknown escape sequence: \…`
`Character literal must be a single character: …`
`Unterminated quote in value list` / `Malformed string literal: …`

Pitfall

One message deserves separate mention because it is about the memory map rather than syntax: the assembler refuses to place code or data in the peripheral block 0xFF00–0xFF7F. Those bytes would never get there — the processor intercepts the addresses before they become memory.

CHAPTER 11

The Macro Preprocessor

A textual macro preprocessor in the NASM style runs before the assembler. It works purely on lines — it knows nothing about instructions or addresses — and its output is source text that the assembler then receives.

That separation is why macros are so free: a macro may contain anything, including directives or part of another macro.

Macros

```
%macro DELAY 1          ; name and parameter count
        LDA %1          ; %1 to %9 are the arguments
%%loop:                 ; %%name = a label unique per expansion
        DEC
        JNZ %%loop
%endmacro

        DELAY #10       ; invocation
```

Parameters are positional, at most nine. A macro is invoked by name and arguments are separated by commas.

Unique labels

A label inside a macro would clash with itself on the second expansion. The `%%` prefix solves that — `%%loop` becomes a name unique to each expansion, so a macro can be used as often as you like in one program.

> **Note**
>
> Macros may invoke each other. Nesting depth is limited to 16 expansions, which is a guard against infinite recursion rather than a design limit.

Text substitution

```
%define DEBUG 1
%define VRAM 0x8000
%undef DEBUG
```

`%define` replaces **whole identifiers**, not substrings — `VRAM_END` is not broken by a definition of `VRAM`. A definition with no value (`%define DEBUG`) stores one.

> **Pitfall**
>
> For naming addresses and constants the `EQU` directive is almost always better than `%define`. The assembler knows about `EQU`, so the constant turns up in the symbol table and in the debugger; `%define` is a purely textual substitution that leaves no trace in the output.
>
> `%define` earns its place where `EQU` cannot go — conditional assembly, and substitutions that are not numbers.

Conditional assembly

```
%ifdef DEBUG
        CALL dump_state
%endif

%if VERSION >= 2
        CALL new_routine
%elif VERSION == 1
        CALL old_routine
%else
%error Unsupported version
%endif
```

Available are `%ifdef`, `%ifndef`, `%if`, `%elif`, `%else` and `%endif`.

The expression in `%if` is simple: either a number, or two numbers joined by `==`, `!=`, `<`, `>`, `<=` or `>=`. A non-zero value means true. The preprocessor knows no richer logic — which is just as well, because conditional assembly you cannot read at a glance is a source of mysteries.

The `%error` directive stops assembly with a message of your own. An unquoted message goes through `%define` substitution; a fully quoted one is emitted verbatim.

Repetition

```
%rep 8
        SHL
%endrep
```

The block is expanded that many times, at most a thousand. Useful for unrolled loops and generated tables.

Including files

```
%include "stdlib.asm"
```

The include is textual. An included file may define macros and constants.

Conditionals must be closed inside the file they were opened in — but `%define` crosses file boundaries. That is exactly what the usual double-inclusion guard rests on:

```
%ifndef MYLIB_INCLUDED
%define MYLIB_INCLUDED 1

; ... library body ...

%endif
```

> **Note**
>
> How a file is found depends on the environment: the command-line tool looks on disk, the browser in a registry of bundled libraries. The preprocessor itself knows no file system — it is handed a function that supplies the contents.

Limits

Limit	Value	Why it exists
macro expansion depth	16	a guard against infinite recursion
`%define` substitution passes	8	a substitution can reveal another
`%rep` repetitions	1024	
total `%rep` output	65 536 lines	the preprocessor runs on every keystroke in the editor

The last row gives away why the limits are as low as they are: the preprocessor does not only run at assembly time but continuously in the editor, so that highlighting and error reporting work. It therefore has to finish quickly even over half-written, temporarily nonsensical code.

Errors and context

Every output line carries its original line number and, if it came from an expansion, a readable context. An error in a macro therefore does not point at generated text but at both relevant source lines:

```
Line 12 (in macro DELAY expanded at line 30): XYZ does not support
immediate mode
```

The same mechanism is what makes stepping and breakpoints work in code produced by macros.

> **Pitfall**
>
> Comment recognition is simple and quote-blind: a semicolon anywhere on a line starts a comment. A string containing a semicolon therefore behaves inside a macro in a way you would not expect.

CHAPTER 12

Idioms and Patterns

The patterns in this chapter are not a matter of taste. They are what the shape of the machine pushes a programmer towards: one accumulator, an 8-bit ALU, pointers only in the zero page. Knowing them means writing code that looks like other SAP-3/16 code — which on an 8-bit is a practical advantage, not an aesthetic one.

The figures quoted here come from measuring the whole corpus of programs in the repository: 28 288 written instruction lines and 1 305 057 instructions actually executed.

The accumulator shuffle

The most frequent pair of instructions in a real run is `STA zp` followed by `LDA zp` — **8.5 % of all consecutive pairs**. It is not inefficiency, it is the tax on having one accumulator: an intermediate result has to be put down so the next operand can be picked up.

```
        LDA a
        ADD b
        STA sum         ; put the result down
        LDA c           ; and load the next operand
```

For the same reason `LDA #constant` followed by `STA memory` is the most common static pair in the corpus — 2 616 occurrences. There is no "store an immediate to memory" instruction, so this pair is the only way.

16-bit arithmetic

The machine counts in bytes, but addresses and coordinates are 16-bit. There are four patterns for bridging that gap, each with its place.

Sums and differences with ADD/ADC

The lowest byte is added with `ADD`, which ignores the incoming carry and sets the outgoing one; the higher bytes with `ADC`, which uses it.

```
        LDA #0xF4               ; 500 = 0x01F4
        ADD #0x2C               ; + 300 = 0x012C
        STA a_lo                ; C carries into the high byte
```

```
        LDA #0x01
        ADC #0x01
        STA a_hi                ; result 0x0320 = 800
```

Subtraction works the same way with the `SUB` / `SBC` pair, where `C` is a borrow.

> **Note**
>
> On a 6502 this would need `CLC` before `ADD` and `SEC` before `SUB`. Not here — `ADD` and `SUB` do not read the incoming flag. It is a small thing, but it saves two instructions in every multi-byte operation and is why `CLC` and `SEC` barely appear in real programs.

Stepping a pointer with ADW and INW

When a constant has to be added to a **memory** word, there is no need to drag it through the accumulator. `ADW` and `SBW` do it in place, `INW` and `DEW` by one:

```
        ADW ptr, #40            ; one screen row down (40 chars)
        INW ptr                 ; one byte further
```

Neither touches the flags, so they can be mixed between a compare and a jump. `INW` is used 316 times in the corpus and `ADW` 78 — and `ADW` is an instruction only days old, which the raycaster took up immediately.

Setting a pointer with LXY and SXY

There are two ways to fill a two-byte pointer with an address. The older uses byte selection:

```
        LDA #<buffer
        STA ptr
        LDA #>buffer
        STA ptr+1
```

The newer is one pair of instructions and never touches the accumulator:

```
        LXY #buffer             ; X:Y := the address (X is high)
        SXY ptr                 ; store it as a zero-page word
```

Walking a buffer

A zero-page pointer plus the `(zp),Y` mode is the only way to walk memory whose address is computed at run time.

```
        LDY #0
.copy:  LDA (source),Y
        STA (dest),Y
        INY
        CPY #4
        JNE .copy
```

Dynamically, `STA (zp),Y` followed by `INY` is 1.4 % of all executed pairs — this pattern really is everywhere in running code.

> **Note**
>
> For longer stretches of contiguous memory there is a faster route: the blitter, which copies or fills a whole block on one register write. The loop above belongs where something is done with each byte.

Staging operands for MUL and DIV

`MUL` and `DIV` take their second operand from **B**, which is only reachable through the accumulator via `TAB`. Hence a pattern that reads backwards:

```
        LDA divisor
        TAB                     ; B := divisor
        LDA dividend            ; A := dividend
        DIV                     ; A := quotient, B := remainder
```

The `TAB` -> `LDA` -> `MUL` cluster is about 7 % of executed instructions — in mathematical code it is the densest part of the whole program.

> **Pitfall**
>
> Between `TAB` and its `MUL` / `DIV` there must be **nothing** that touches the ALU. And the same trap waits afterwards, because the secondary result stays in **B**:

```
        MUL
        CMP #0x04       ; <- the high byte is gone
        TBA             ; reads 0x04, not the high byte
```

The fix is to retrieve the second half before touching the ALU:

```
        MUL
        PHB             ; high byte onto the stack
        CMP #0x04       ; now B may vanish freely
        POP             ; and pull it back into A
```

This example is not hypothetical — the sample program for this chapter ran into exactly this bug while being written.

Shifts and fixed point

A shift is the cheapest multiplication and division by a power of two. Dynamically `SHR` and `SHL` on the accumulator are 6.0 % and 4.9 % of executed instructions — more than addition.

A multi-byte shift is built from a shift at one end and a rotate at the other, through the `C` flag:

```
        SHR a_hi                ; the shifted-out bit goes to C
        ROR a_lo                ; and C pulls it in from the top
```

Leftwards it is assembled the other way round: `SHL` on the low byte, `ROL` on the high one.

The pair `SHR` -> `SHR` is 3.9 % of executed pairs and `SHL` -> `SHL` 3.7 %. Behind that number is fixed-point arithmetic in Q8.8 format: after multiplying two such numbers the result has to be shifted back, and that is a chain of shifts.

Jump tables

A state machine or a dispatch by index is done with a table of addresses and an indirect jump:

```
        LDA state
        SHL                     ; index x 2 (an address is 2 B)
        TAX
```

```
        LDA table,X
        STA target
        LDA table+1,X
        STA target+1
        JMP (target)

table:  .WORD state0, state1, state2
```

For a call rather than a jump, use `CALL (zp)` with the same preparation.

Loop closers

Three variants, depending on where the counter lives:

Counter	Closer
in memory	`DEC count` + `JNZ` — touches no register, 148 occurrences in the corpus
in an index, counting down	`DEX` + `JNZ` — the shortest
in an index, counting up	`INX` + `CPX #N` + `JNE` — when the index also addresses

The third is the most frequent dynamically (`INY` / `CPY` / `JNZ` is 1.6 % of executed triples), because walking an array needs the index anyway.

Holding the frame rate

A program that draws must run at the same speed on a slow machine and a fast one. That is what the real-millisecond counter and an anchor the frame returns to are for:

```
wait_frame:                     ; hold a frame to FRAME_MS ms
.spin:  LDA CLK_MS_LO           ; elapsed = CLK_MS - anchor
        SUB ANCHOR
        STA EL
        LDA CLK_MS_HI
        SBC ANCHOR+1
        JNZ .done               ; over 256 ms - long past due
        LDA EL
        CMP #FRAME_MS           ; C = 1 while it is less
        JC .spin
.done:  LDA CLK_MS_LO           ; anchor only on release
        STA ANCHOR
        LDA CLK_MS_HI
```

```
        STA ANCHOR+1
        RET
```

> **Note**
>
> The anchor moves at the moment of release, not to a fixed multiple. If an absolute deadline were computed, a frame that missed it would carry debt, the program would try to catch up, and eventually the counter would wrap. This way a late frame is simply late.

What the corpus actually contains

Pattern	Occurrences	Note
`LDA #imm` -> `STA mem`	2 616	storing a constant to memory
`LDA mem` -> `STA mem`	1 469	a memory-to-memory copy
`LDA a` -> ALU op -> `STA`	933	read, modify, write through the accumulator
16-bit comparison	214	a cascade of `LDA` / `CMP` /jump, twice
`DEC mem` -> `JNZ`	148	a loop closer with the counter in memory
`CMP #0` after a load	3	almost nobody — the `Z` from the load is used correctly

The last row is the most interesting: hardly anyone writes a redundant `CMP #0` after loading a value, because `LDA` sets the flags itself. A small thing, but it shows the "even a move sets flags" convention has been taken up in practice.

CHAPTER 13

Performance, Cycles and Measurement

Performance on this machine does not have to be guessed at. T-states are part of the specification, the processor counts them itself and a program can read them — so the answer to "is this stretch faster?" is a measurement, not an opinion.

The model

An instruction's cost is the sum of two terms:

$$T = \text{fetch} + \text{execute}$$

Fetch costs **2 T-states per instruction byte** — one to send the program counter to the address register, one to read the byte. Execute is the rest: assembling the effective address plus the actual work.

From which comes the first and most important figure:

Instruction	Fetch	Execute	Fetch share
`LDA #n`	4 T	1 T	80 %
`ADD zp`	4 T	3 T	57 %
`LDA abs`	6 T	3 T	67 %
`JMP abs`	6 T	1 T	86 %

Table 3. Fetch as a share of an instruction's duration.

More than half of machine time goes on reading instructions, not on work. The practical consequence: code density pays off more than clever register use. A shorter instruction is a faster instruction, almost regardless of what it does.

What the addressing modes cost

The differences are in the table of every dictionary entry. Using `LDA` as the example:

Mode	Bytes	T	vs. `zp`
`#n`	2	5	−1 T
`zp`	2	6	—
`abs`	3	9	+3 T
`abs,X`	3	11	+5 T
`abs,Y`	3	11	+5 T
`(zp)`	2	10	+4 T
`zp,X`	2	7	+1 T
`(zp),Y`	2	13	+7 T

Table 4. The cost of the addressing modes, using `LDA` as the example.

The zero page is three T-states and one byte cheaper than an absolute address. In a loop that runs a thousand times that is a difference of tens of percent in total time — and it is the cheapest optimisation available on this machine.

Jumps cost the same taken or not

A conditional jump takes 7 T-states whether or not the condition held. A jump not taken saves nothing.

It looks like a wasted opportunity, but it is deliberate. A program's running time therefore does not depend on its data, which is a precondition for two things the project rests on: the reference traces that verify the FPGA behaves like the emulator, and time travel in the debugger.

> **Note**
>
> Games hardware of the 1980s had the same constraint — and that is exactly why you could "count cycles" on it. A program whose timing is predictable in advance can synchronise the picture with the processor, a trick that is impossible on a modern CPU with caches and branch prediction.

The counters

The processor offers four independent time bases. All the counters are read-only and are cleared only on reset.

Address	Register
0xFF30	TMR_DIV
0xFF31	TMR_CNT
0xFF32	CYC0
0xFF33	CYC1
0xFF34	CYC2
0xFF35	CYC3
0xFF36	UPT_LO
0xFF37	UPT_HI
0xFF38	CLK_TPMS_LO
0xFF39	CLK_TPMS_HI
0xFF3A	CLK_MS_LO
0xFF3B	CLK_MS_HI

Figure 10. Timer and counter registers (device 3).

Counter	Width	Purpose
CYC0 – CYC3	32 b	T-states — precise measurement of a stretch of code
UPT_LO / UPT_HI	16 b	frames — time measured in video frames
CLK_TPMS_LO / HI	16 b	how many T-states fit into a real millisecond
CLK_MS_LO / HI	16 b	real milliseconds — the base for pacing

> **Note**
>
> Reading the lowest byte **latches a snapshot** of the whole value. A program must therefore read CYC0 before CYC1, or it gets two bytes from different moments and nonsense across a 256 boundary. The same applies to UPT_LO and CLK_MS_LO.

CLK_TPMS is the only window into how fast the machine is running. A program that wants to behave the same in a browser and on the board computes from it how much work fits in a frame — or simply uses CLK_MS.

How to measure

The difference of two reads of the T-state counter is the exact number of ticks between them — including the overhead of the measurement itself. That has to be measured separately and subtracted:

```
        ; reference: two reads back to back
        CALL sample
        CALL delta
        LDA d_lo
        STA overhead

        ; the measured stretch
        CALL sample
        LDX #10
.loop:  DEX
        JNZ .loop
        CALL delta

        LDA d_lo
        SUB overhead            ; subtract the overhead
        STA d_lo
```

The complete program is in `book/reference/examples/13-mereni.asm`. It prints `0073`, that is 115 T-states — exactly what a hand count gives:

$$\underbrace{5}_{\text{LDA \#10}} + 10 \times \left(\underbrace{4}_{\text{DEX}} + \underbrace{7}_{\text{JNZ}} \right) = 115$$

Theory and measurement agreeing to the unit is no accident; it is the same property that makes the machine steppable and rewindable.

> **Pitfall**
>
> That program itself showed why measurements should be verified rather than assumed: in its first version the constant `overhead` lived at `0x15`, which is the second byte of the two-byte variable `tmp` at `0x14`. The `delta` subroutine overwrote the overhead with it and the measurement came out 46 T-states too high. The zero page is small and nobody reports collisions in it.

The frame budget

The frame base is 1 000 T-states (`VSYNC_PERIOD`), but for planning it is more useful to count in real time: how much work fits between two frames at the target clock.

An average instruction costs 7 to 9 T-states. On the board, with the AGATA-1 core running at 25.2 MHz with one T-state per clock, that works out at roughly **3 million**

instructions per second. For comparison, a 6502 at 1 MHz manages about 0.35 MIPS — the board is therefore about ten times a classic 8-bit.

At 30 frames per second the budget comes to around 100 000 instructions per frame. That is a lot — and it runs out quickly once you start drawing pixel by pixel.

Speed	Where it is used
fractions of a Hz	stepping with visible control signals — teaching mode
up to 2 MHz	the emulator in a browser
25.2 MHz	turbo mode on the FPGA; the ceiling the specification allows

> **Note**
>
> The specification does not pin a frequency. `CLK_TPMS` can read up to 25 200, which corresponds to one T-state per FPGA pixel-clock cycle. A program should not rely on a particular rate — it should read it.

What is worth optimising

Measurements on a real corpus of programs give a clear order.

1. **Move data into the zero page.** Three T-states and one byte per access, with no change to the logic.
2. **Shorten the code.** Fetch is the majority term; every instruction byte saved is a T-state saved on every pass.
3. **Let the blitter do the work.** A block copy through a control register runs with no instruction loop.
4. **Avoid needless shuffling.** The `STA zp` / `LDA zp` pair is 8.5 % of executed pairs; some of it can be removed by rearranging the computation so the intermediate result stays in the accumulator.

And what is not worth it: micro-optimising inside expressions. The difference between two ways of writing the same computation is a few T-states, while moving a variable into the zero page or removing one instruction from a loop is multiplied by the number of passes.

> **Pitfall**
>
> The T-state counts in this book are for ISA 3.1. If a future core shortened the fetch — the most promising direction, precisely because fetch dominates — the

> tables throughout the book would change. That is why they are generated from the source rather than transcribed.

PART III

The Instruction Dictionary

CHAPTER 14

How to Read the Dictionary

The dictionary in the next chapter describes every mnemonic separately, in alphabetical order. An entry is built to be read top to bottom and abandoned at any point: what you look up most often is at the top, what you need once a year at the bottom.

Anatomy of an entry

Every entry opens with the mnemonic, one sentence on what it does, and a table of all its encoded forms. The columns mean:

Column	Meaning
Mode	the addressing mode; `implied` means the instruction takes no operand
Syntax	how the form is written in assembly
Opcode	the byte the assembler emits
Bytes	the instruction's total length in memory
T	T-states including the fetch — exactly what the instruction costs
Flags	the flags this form **defines**

After the table come the semantics, the microcode for non-trivial instructions, an example and the pitfalls.

What an empty flags column means

This is the most important convention in the book. The Flags column lists only the flags an instruction sets to a new value. A flag that is not listed **keeps its previous value** — it is not cleared.

The difference is not cosmetic. It is what makes this possible:

```
        CMP #10         ; sets C, Z, N
        AND #0x0F       ; sets Z, N — C survives
        JC  smaller     ; the jump still follows the compare
```

For instructions that define no flag at all the table says "none" instead of a list. These are typically operations on addresses and pointers (`INW`, `DEW`, `ADW`, `SBW`, `LXY`, `SXY`, `TXYS`): counting with an address has no business destroying the flags a program is deciding on.

> **Note**
>
> `RTI` and `PLF` are the opposite extreme — they restore **all five** flags at once, including `I`, because they load a whole status byte off the stack. Their table therefore reads `C Z N V I`.

Microcode listings

Where **how** a result is reached is interesting, the entry carries a microcode listing. One pair of brackets is one T-state; inside are the control signals asserted in it. The number at the end is the total T-state count.

The first two to six steps are always the fetch and look the same across instructions (`[PCM] [RO,II,CE]` plus any operand-byte reads). The interesting part starts after them.

The meaning of each signal is in the appendix on control signals; to read a listing it is enough to know that a signal ending in `O` typically **drives** the bus and one ending in `I` **takes** from it.

Examples

Code examples are excerpts from real programs, not pseudocode. Where an example prints something, that is the output the machine actually produces.

Pitfalls

A box marked as a pitfall does not warn about a mistake but about behaviour that is correct and surprising at the same time. Most of them share an origin: the processor does exactly what the microcode says, and human intuition sometimes expects something else.

> **Pitfall**
>
> An example of such a box. The most common pitfall of the whole machine: every binary ALU operation overwrites register **B** with its operand. A value you put there with `TAB` will not survive the next `ADD`, `CMP` or `AND`.

See also: `LDA`, `CMP`, `DIV`

CHAPTER 15

The Instruction Dictionary

Entries are ordered alphabetically by mnemonic. Forms that differ only in addressing mode share one entry; the jump aliases (`JEQ`, `JNE`, `JGE`, `JLT`) have their own short entries pointing at the instruction they stand for.

The dictionary covers all 79 mnemonics the processor can decode. The anatomy of an entry and the notation are described in the previous chapter.

Conditional jumps at a glance

There are twelve of them and they are easy to confuse, so here is the whole family in one place. All have only an absolute form, are three bytes long and take 7 T whether or not the jump happens.

Instruction	Condition	After `CMP` means
`JZ` / `JEQ`	`Z`	equal
`JNZ` / `JNE`	`¬Z`	not equal
`JC` / `JLT`	`C`	less (unsigned)
`JNC` / `JGE`	`¬C`	greater or equal (unsigned)
`JGT`	`¬C ∧ ¬Z`	strictly greater (unsigned)
`JLE`	`C ∨ Z`	less or equal (unsigned)
`JN`	`N`	negative — reliable only when `V = 0`
`JP`	`¬N`	positive or zero — **not** an unconditional jump
`JLTS`	`N ⊕ V`	less, signed — pair with `SUB`, not `CMP`
`JGES`	`¬(N ⊕ V)`	greater or equal, signed — likewise
`JV`	`V`	signed overflow
`JNV`	`¬V`	no overflow

Three rows of that table are the most common sources of error: `JP` is not `JMP`, the signed jumps do not work after `CMP` (which does not set `V`), and `C` after a subtraction is a borrow, not a carry.

ADC

Adds the operand to the accumulator including the C flag. The higher bytes of a multi-byte sum.

Mode	Syntax	Opcode	Bytes	T	Flags
`#n`	`ADC #n`	`0x88`	2	6	`C Z N V`
`zp`	`ADC addr8`	`0x89`	2	7	`C Z N V`
`abs`	`ADC addr16`	`0x8A`	3	10	`C Z N V`
`abs,X`	`ADC addr16,X`	`0x8B`	3	12	`C Z N V`
`abs,Y`	`ADC addr16,Y`	`0x8C`	3	12	`C Z N V`
`(zp)`	`ADC (addr8)`	`0x8D`	2	11	`C Z N V`
`zp,X`	`ADC addr8,X`	`0x8E`	2	8	`C Z N V`
`(zp),Y`	`ADC (addr8),Y`	`0x8F`	2	14	`C Z N V`

`ADC` is `ADD` plus the carry: `A := A + operand + C`. It exists for adding numbers wider than one byte.

Splitting the work between `ADD` and `ADC` is simple and needs no flag preparation: add the lowest byte with `ADD` (which ignores the incoming carry and sets the outgoing one) and every higher byte with `ADC`.

```
        LDA a_lo
        ADD b_lo        ; C := carry out of the low byte
        STA sum_lo
        LDA a_hi
        ADC b_hi        ; adds that carry in
        STA sum_hi
```

The `V` flag reports signed overflow — that the result is wrong if the operands are read as signed numbers.

See also: `ADD`, `SBC`, `ADW`, `CLC`

ADD

Adds the operand to the accumulator. Ignores the incoming carry, sets the outgoing one.

Mode	Syntax	Opcode	Bytes	T	Flags
`#n`	`ADD #n`	`0x80`	2	6	`C Z N V`
`zp`	`ADD addr8`	`0x81`	2	7	`C Z N V`
`abs`	`ADD addr16`	`0x82`	3	10	`C Z N V`
`abs,X`	`ADD addr16,X`	`0x83`	3	12	`C Z N V`
`abs,Y`	`ADD addr16,Y`	`0x84`	3	12	`C Z N V`
`(zp)`	`ADD (addr8)`	`0x85`	2	11	`C Z N V`
`zp,X`	`ADD addr8,X`	`0x86`	2	8	`C Z N V`
`(zp),Y`	`ADD (addr8),Y`	`0x87`	2	14	`C Z N V`

`A := A + operand`. The `C` flag plays no part on input — unlike the 6502, **there is no need to `CLC` before adding**.

```
        LDA price
        ADD tax
        STA total
        JC  overflowed  ; C = 1 means it did not fit in a byte
```

> **Pitfall**
>
> The operand ends up in register **B**. This is true of every binary ALU operation and is the most common reason a value stored with `TAB` mysteriously disappears.

See also: `ADC`, `SUB`, `ADW`, `MUL`

ADW

Adds an 8-bit constant to a 16-bit word in the zero page. Changes no flags.

Mode	Syntax	Opcode	Bytes	T	Flags
`zp`	`ADW addr8,#n`	`0xE8`	3	13	*none*

`ADW zp, #imm8` adds the constant to the whole two-byte word stored at `zp` (low byte first) and writes the result back. It all happens in memory — neither the accumulator nor the indices are used.

It is the instruction for stepping pointers by more than one. Typically moving to the next screen row:

```
        ADW ptr, #40    ; one row down (40 chars)
```

No flag changes. That is deliberate: advancing a pointer should not destroy the compare a loop is deciding on.

> **Note**
>
> The constant is 8-bit and **non-negative**. Subtraction is done with `SBW`, not by adding a negative number — that would propagate the borrow wrongly.

See also: `SBW`, `INW`, `DEW`, `LXY`, `SXY`

AND

Bitwise AND of the accumulator with the operand.

Mode	Syntax	Opcode	Bytes	T	Flags
`#n`	`AND #n`	`0xA0`	2	6	`Z N`
`zp`	`AND addr8`	`0xA1`	2	7	`Z N`
`abs`	`AND addr16`	`0xA2`	3	10	`Z N`
`abs,X`	`AND addr16,X`	`0xA3`	3	12	`Z N`
`abs,Y`	`AND addr16,Y`	`0xA4`	3	12	`Z N`
`(zp)`	`AND (addr8)`	`0xA5`	2	11	`Z N`
`zp,X`	`AND addr8,X`	`0xA6`	2	8	`Z N`
`(zp),Y`	`AND (addr8),Y`	`0xA7`	2	14	`Z N`

`A := A & operand`. Sets `Z` and `N`; `C` and `V` keep their values.

It is used for masking — keeping only part of a byte:

```
        LDA status
        AND #0x0F       ; keep the low nibble
```

> **Note**

That `AND` does not touch `C` is a deliberate ISA 3.0 decision (earlier versions cleared it). Thanks to it you can mask bits between a compare and a conditional jump without losing the compare's result.

See also: `OR`, `XOR`, `NOT`, `BIT`

BIT

Tests the operand's bits against the accumulator. Sets flags only, leaves A alone.

Mode	Syntax	Opcode	Bytes	T	Flags
`#n`	`BIT #n`	`0xE4`	2	6	`Z N`
`zp`	`BIT addr8`	`0xE5`	2	7	`Z N`
`abs`	`BIT addr16`	`0xE6`	3	10	`Z N`

`BIT` does exactly what `AND` does but discards the result — only `Z` and `N` remain. It is to `AND` what `CMP` is to `SUB`.

```
        LDA mask
        BIT status      ; is any bit of the mask set?
        JZ  none
```

The most common use is testing a single bit without losing the accumulator:

```
        LDA #0b00000100
        BIT flags
        JNZ bit2_is_set
```

Pitfall

`BIT` does leave **A** alone, but it overwrites **B** like every binary ALU operation. "Non-destructive" means only towards the accumulator.

See also: `AND`, `CMP`

BRK

Software interrupt. Pushes the same frame as a hardware IRQ but has its own vector.

Mode	Syntax	Opcode	Bytes	T	Flags
implied	`BRK`	`0xCD`	1	12	`I`

`BRK` is an ordinary fetched instruction that pushes the high byte of **PC**, the low byte of **PC** and the status byte, disables interrupts and jumps to the address in the BRK vector (`0xFFFA`).

It differs from a hardware interrupt in three ways: a program triggers it, it **cannot be masked** by the `I` flag, and it vectors elsewhere — so a `BRK` handler is a different routine from the IRQ handler.

```
        .VECTOR BRK, syscall

        LDA #SERVICE_PRINT
        BRK             ; call a system service
        ; execution resumes here after RTI
```

Return is by `RTI` , continuing at the byte **after** `BRK` . Besides system calls, `BRK` serves as a debugger breakpoint — one byte overwritten with `0xCD` stops the program anywhere.

See also: `RTI` , `EI` , `DI`

CALL

Calls a subroutine: pushes the return address and jumps.

Mode	Syntax	Opcode	Bytes	T	Flags
`abs`	`CALL addr16`	`0xCA`	3	11	*none*
`(zp)`	`CALL (addr8)`	`0xCE`	2	13	*none*

`CALL` pushes the address of the following instruction (high byte first, then low) and jumps to the target. `RET` brings it back.

There are two forms. The direct one (`CALL label`) is the common case. The indirect one (`CALL (zp)`) takes the target from a pointer in the zero page — which is how you build a function table or a callback on this machine:

```
        LDA #<handler   ; low byte of the address
        STA vector
        LDA #>handler   ; high byte
        STA vector+1
        CALL (vector)
```

The stack is shared with interrupts, so calling from inside a handler is fine.

Pitfall

Nesting is not bounded from above by anything — the stack has 256 bytes, that is 128 return addresses, and nobody checks for overflow. Deep recursion overwrites the zero page and shows up as corrupted data, not as a crash.

See also: `RET`, `JMP`, `BRK`

CLC

Clears the carry flag.

Mode	Syntax	Opcode	Bytes	T	Flags
implied	`CLC`	`0x04`	1	3	`C`

`C := 0`. Nothing else is affected.

Note

Unlike the 6502, `CLC` is **not needed before an addition**. `ADD` ignores the incoming carry and so does `SUB`; only `ADC` and `SBC` read it, and those are used exclusively on higher bytes, where the carry should be the one from the previous step.

Measurement bears this out thoroughly: across every example program and game in the repository, `CLC` and `SEC` never appear as instructions — the only uses are in a test that deliberately exercises every opcode. One program even uses `CLC` as the name of a constant, which can only work because the instruction is never written.

That does not make them useless: there is no other way to set the input bit for `ROL` or `ROR`. It means the division of labour between `ADD` and `ADC` came out so that the carry needs no manual attention.

See also: `SEC` , `ADC` , `SBC` , `ROL` , `ROR`

CMP

Compares A with the operand: sets flags from A − operand but leaves A alone.

Mode	Syntax	Opcode	Bytes	T	Flags
`#n`	`CMP #n`	`0xB8`	2	6	`C Z N`
`zp`	`CMP addr8`	`0xB9`	2	7	`C Z N`
`abs`	`CMP addr16`	`0xBA`	3	10	`C Z N`
`abs,X`	`CMP addr16,X`	`0xBB`	3	12	`C Z N`
`abs,Y`	`CMP addr16,Y`	`0xBC`	3	12	`C Z N`
`(zp)`	`CMP (addr8)`	`0xBD`	2	11	`C Z N`
`zp,X`	`CMP addr8,X`	`0xBE`	2	8	`C Z N`
`(zp),Y`	`CMP (addr8),Y`	`0xBF`	2	14	`C Z N`

`CMP` performs the subtraction `A − operand`, discards the result and keeps only the flags. It is the only way to ask "is A smaller?" on SAP-3/16 without destroying the accumulator.

Flags after `CMP`:

Flag	Meaning
`Z`	`A == operand`
`C`	borrow: set exactly when `A < operand` (unsigned)
`N`	the top bit of the difference
`V`	**unchanged** — `CMP` does not compute overflow

The convention for `C` is the opposite of the 6502: here `C` is a **borrow**, meaning "the subtraction underflowed". So that this need not be memorised, the assembler provides aliases that read the way people think:

Written	Real instruction	Jumps when
`JEQ`	alias for `JZ`	`A == operand`
`JNE`	alias for `JNZ`	`A != operand`
`JLT`	alias for `JC`	`A < operand` (unsigned)
`JGE`	alias for `JNC`	`A >= operand` (unsigned)
`JGT`	its own opcode	`A > operand` (unsigned)

Written	Real instruction	Jumps when
`JLE`	its own opcode	`A <= operand` (unsigned)

The first four rows are just other names for the same opcodes — the assembler turns them into `JZ`, `JNZ`, `JC` and `JNC`. `JGT` and `JLE`, by contrast, are separate instructions with a composite condition (`C v Z`), because that cannot be assembled from a single flag.

```
        LDA count
        CMP #10
        JLT few          ; count < 10
        JEQ exactly      ; count == 10
                         ; falls through for count > 10
```

Microcode — `CMP zp  (0xB9)`

```
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [ORO,MI] [RO,BI] [CMP,EO,FI]
(7)
```

The last step is the only one where anything happens: the ALU in `CMP` mode subtracts and `FI` writes the flags. The missing `AI` is the whole difference from `SUB` — the result is stored nowhere.

Pitfall

`CMP` does not set `V`, so the signed jumps `JGES` and `JLTS` read a **stale** overflow value after it and can answer wrongly. Signed comparison uses `SUB` or `SBC`, which do set `V`:

```
        LDA #-100        ; 0x9C
        SUB #100         ; overflows: N=0, V=1
        JLTS smaller     ; taken — −100 < 100 signed
```

The price is that `SUB`, unlike `CMP`, overwrites `A`.

Pitfall

The operand of `CMP` ends up — as with every binary ALU operation — in register **B**. Whatever was there is gone.

See also: `SUB`, `SBC`, `BIT`, `CPX`, `CPY`

CPX

Compares register X with the operand. The accumulator is untouched.

Mode	Syntax	Opcode	Bytes	T	Flags
`#n`	`CPX #n`	`0x58`	2	7	`C Z N`
`zp`	`CPX addr8`	`0x59`	2	8	`C Z N`
`abs`	`CPX addr16`	`0x5A`	3	11	`C Z N`

The same as `CMP` but comparing **X**. Sets `C`, `Z` and `N` from `X - operand`; neither **X** nor **A** changes.

It typically closes a loop counted by an index:

```
loop:   LDA source,X
        STA dest,X
        INX
        CPX #LENGTH
        JNE loop
```

Microcode — `CPX #n  (0x58)`

```
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [XO,TI] [ORO,BI]
[ATM,CMP,EO,FI]  (7)
```

The listing shows how **A** is avoided: the index goes through **TMP** and the `ATM` multiplexer feeds it into the ALU in place of the accumulator.

> **Pitfall**
>
> **B** is overwritten here too. `CPX` is non-destructive towards the accumulator, not towards **B**.

See also: `CPY`, `CMP`, `INX`, `DEX`

CPY

Compares register Y with the operand. The accumulator is untouched.

Mode	Syntax	Opcode	Bytes	T	Flags
`#n`	`CPY #n`	`0x78`	2	7	`C Z N`
`zp`	`CPY addr8`	`0x79`	2	8	`C Z N`
`abs`	`CPY addr16`	`0x7A`	3	11	`C Z N`

The mirror image of `CPX` for register **Y**. Sets `C`, `Z` and `N` from `Y − operand`.

See also: `CPX`, `CMP`, `INY`, `DEY`

DEC

Decrements by one: the accumulator, or a byte directly in memory.

Mode	Syntax	Opcode	Bytes	T	Flags
implied	`DEC`	`0x09`	1	3	`Z N`
`zp`	`DEC addr8`	`0xD2`	2	7	`Z N`
`abs`	`DEC addr16`	`0xD3`	3	10	`Z N`

It has two forms. Without an operand it works on **A** with an address it is a read-modify-write straight in memory — the accumulator is not used at all.

```
        DEC             ; A := A - 1
        DEC count       ; the byte at count decrements
```

Sets `Z` and `N`, leaves `C` alone. Underflow past zero is therefore **not** visible in `C`; the value simply wraps from `0x00` to `0xFF`.

The memory form is the basis of a counted loop that touches no register:

```
loop:   ; ... body ...
        DEC count
        JNZ loop
```

See also: `INC`, `DEW`, `DEX`, `DEY`

DEW

Decrements a 16-bit word in memory by one. Changes no flags.

Mode	Syntax	Opcode	Bytes	T	Flags
`zp`	`DEW addr8`	`0xD6`	2	10	*none*
`abs`	`DEW addr16`	`0xD7`	3	13	*none*

`DEW zp` (or `DEW abs`) subtracts one from the two-byte word stored low byte first. The borrow between bytes is handled inside the instruction through the **`addrCarry`** latch.

No flag changes — not even `Z`. A test for zero has to be made by hand:

```
        DEW length
        LDA length
        OR  length+1    ; are both halves zero?
        JZ  done
```

Pitfall

The missing `Z` is the most common surprise with `INW` / `DEW`. It follows from the rule that pointer operations must not disturb the flags — but it does mean a 16-bit counter cannot be closed with a single jump.

See also: `INW`, `SBW`, `DEC`

DEX

Decrements register X by one.

Mode	Syntax	Opcode	Bytes	T	Flags
implied	`DEX`	`0x11`	1	4	`Z N`

`X := X - 1`, sets `Z` and `N`. The accumulator is not used — the index goes through the hidden **`TMP`**.

See also: `INX`, `DEY`, `CPX`

DEY

Decrements register Y by one.

Mode	Syntax	Opcode	Bytes	T	Flags
implied	`DEY`	`0x13`	1	4	`Z N`

`Y := Y - 1`, sets `Z` and `N`.

See also: `INY`, `DEX`, `CPY`

DI

Disables interrupts (clears the I flag).

Mode	Syntax	Opcode	Bytes	T	Flags
implied	`DI`	`0x07`	1	3	`I`

After `DI` the processor takes no hardware interrupt until `EI` re-enables them. The `BRK` instruction cannot be masked this way — it is an instruction, not an event.

It is used to protect a critical section, typically while reading a multi-byte value an interrupt handler modifies:

```
        DI
        LDA time_lo
        LDX time_hi
        EI
```

See also: `EI`, `RTI`, `HLT`, `BRK`

DIV

Divides A by B: quotient into A, remainder into B. Division by zero does not trap.

Mode	Syntax	Opcode	Bytes	T	Flags
implied	`DIV`	`0x29`	1	4	`C Z N`

`DIV` takes no operand — both inputs come from registers. The dividend is in **A**, the divisor in **B** afterwards **A** holds the quotient and **B** the remainder. Both are 8-bit and integer.

Because no load-style instruction can address **B**, it is filled through the accumulator with `TAB`. Hence the typical order, which looks backwards but is the only correct one:

```
        LDA divisor
        TAB             ; B := divisor
        LDA dividend    ; A := dividend
        DIV             ; A := quotient, B := remainder
        STA quotient
        TBA             ; A := remainder
        STA remainder
```

Microcode — `DIV (0x29)`

```
[PCM] [RO,II,CE] [DIV,EO,AI,FI] [EXO,BI]  (4)
```

Two steps after the fetch: the first has the ALU compute the quotient, drive it onto the bus into **A** and write the flags; the second pulls the secondary result — the remainder — through `EXO` into **B**. `MUL` uses the same pair, only there the secondary result is the high byte of the product.

Division by zero

Dividing by zero neither crashes the processor nor has a vector of its own. Instead it behaves like an operation announcing that it did nothing:

After `DIV` with `B = 0`	State
`C`	1 — the error flag
`A`	unchanged (the dividend stays)
`B`	unchanged (the zero stays)
`Z`, `N`	**unchanged** — the operation did not define them

On a successful division `C` is 0. The test is therefore one jump right after the instruction:

```
        DIV
        JC  divide_by_zero
```

Pitfall

`Z` and `N` after a division by zero hold values from the **previous** operation, not from `DIV`. It follows from the general rule that the flag latch writes only what the operation defined — and dividing by zero defines nothing but `C`. Anyone wanting to test for zero after a failed division must test `A` themselves.

> **Pitfall**
>
> Quotient and remainder are both 8-bit, so `DIV` cannot divide 16-bit numbers. Wider division is assembled byte by byte from `SHR` and `SBC`.

See also: `MUL`, `TAB`, `TBA`, `SBC`

EI

Enables interrupts (sets the I flag).

Mode	Syntax	Opcode	Bytes	T	Flags
implied	`EI`	`0x06`	1	3	`I`

The master switch for interrupts. For one actually to be taken, its source must additionally be enabled in `IRQ_ENABLE`.

```
        LDA #0b00000001
        STA IRQ_ENABLE  ; enable the timer
        EI              ; and unlock interrupts as such
```

> **Note**
>
> After reset `I` is clear, so a program must call `EI` itself. Conversely, on entering an interrupt handler `I` is cleared automatically and `RTI` restores it — the handler need do nothing.

See also: `DI`, `RTI`, `HLT`

HLT

Stops the processor. An enabled interrupt wakes it again.

Mode	Syntax	Opcode	Bytes	T	Flags
implied	`HLT`	`0x02`	1	3	*none*

`HLT` is not necessarily the end of a program. If interrupts are enabled and a source becomes active, the processor wakes and runs the handler; after `RTI` it continues at the instruction after `HLT`.

A true stop happens only when `I` is clear or no source is enabled — which is how `HLT` is used at the end of a program.

```
        EI
wait:   HLT             ; sleeps until something arrives
        JMP wait
```

> **Note**
>
> A stopped processor does not mean a stopped machine. The terminal's queue keeps draining, so text written just before `HLT` still gets printed — exactly like a real serial line finishing its sentence after the processor stops working.

See also: `EI`, `DI`, `NOP`

INC

Increments by one: the accumulator, or a byte directly in memory.

Mode	Syntax	Opcode	Bytes	T	Flags
implied	`INC`	`0x08`	1	3	`Z N`
`zp`	`INC addr8`	`0xD0`	2	7	`Z N`
`abs`	`INC addr16`	`0xD1`	3	10	`Z N`

The mirror image of `DEC`. Without an operand it works on **A** with an address it is a read-modify-write in memory.

Sets `Z` and `N`, leaves `C` alone — a wrap from `0xFF` to `0x00` does not show in `C`. If you need the carry, add with `ADD #1`.

See also: `DEC`, `INW`, `INX`, `INY`, `ADD`

INW

Increments a 16-bit word in memory by one. Changes no flags.

Mode	Syntax	Opcode	Bytes	T	Flags
`zp`	`INW addr8`	`0xD4`	2	10	*none*
`abs`	`INW addr16`	`0xD5`	3	13	*none*

Stepping a pointer by one. The word is stored low byte first and the carry between bytes is handled inside the instruction.

```
        LDA (ptr)       ; read the byte pointed at
        INW ptr         ; and advance to the next
```

Changes no flag — not even `Z`.

See also: `DEW`, `ADW`, `INC`

INX

Increments register X by one.

Mode	Syntax	Opcode	Bytes	T	Flags
implied	`INX`	`0x10`	1	4	`Z N`

`X := X + 1`, sets `Z` and `N`. The most common instruction for walking an array:

```
loop:   LDA data,X
        STA dest,X
        INX
        CPX #LENGTH
        JNE loop
```

See also: `DEX`, `INY`, `CPX`

INY

Increments register Y by one.

Mode	Syntax	Opcode	Bytes	T	Flags
implied	`INY`	`0x12`	1	4	`Z N`

`Y := Y + 1`, sets `Z` and `N`. Used mainly with the `(zp),Y` mode, where **Y** serves as an offset inside a buffer.

See also: `DEY`, `INX`, `CPY`

JC

Jumps when the C flag is set. After a compare this means less than.

Mode	Syntax	Opcode	Bytes	T	Flags
`abs`	`JC addr16`	`0xC3`	3	7	*none*

A conditional jump on the carry flag. Because `C` after a subtraction is a **borrow**, after `CMP` a set `C` means exactly "A was smaller". The readable name for that case is the alias `JLT`.

Away from comparisons, `JC` is used where the carry means overflow: after `ADD`, after `SHL`, after `DIV` (where it signals division by zero).

> **Note**
>
> A conditional jump takes 7 T whether or not it was taken. A jump not taken saves no time — which is what makes a program's running time independent of its data.

See also: `JNC`, `CMP`, `JLT`, `SHL`

JEQ

Jumps when the compared values were equal.

Alias for `JZ` — the assembler emits the same opcode; encoding and timing are identical.

JGE

Jumps when A was greater than or equal to the operand (unsigned).

Alias for `JNC` — the assembler emits the same opcode; encoding and timing are identical.

JGES

Jumps when the value is greater or equal — signed.

Mode	Syntax	Opcode	Bytes	T	Flags
`abs`	`JGES addr16`	`0xE2`	3	7	*none*

The signed counterpart of `JGE`. Jumps when `N` and `V` have the same value.

A signed comparison cannot be made after `CMP`, because `CMP` does not set `V`. It therefore pairs with `SUB` or `SBC`:

```
        LDA temperature
        SUB #0          ; sets both N and V
        JGES not_below_zero
```

> **Pitfall**

> After `CMP`, `JGES` reads a stale `V` from some earlier instruction and answers nonsense. It is a silent fault — the program assembles and runs.

See also: `JLTS`, `SUB`, `JGE`, `CMP`

JGT

Jumps when A was strictly greater than the operand (unsigned).

Mode	Syntax	Opcode	Bytes	T	Flags
`abs`	`JGT addr16`	`0xE0`	3	7	*none*

A composite condition: it jumps when neither `C` nor `Z` is set — that is, "neither smaller nor equal". It is not an alias; it is its own opcode, because that condition cannot be assembled from a single flag.

```
        LDA score
        CMP #100
        JGT record
```

See also: `JLE`, `CMP`, `JGE`

JLE

Jumps when A was less than or equal to the operand (unsigned).

Mode	Syntax	Opcode	Bytes	T	Flags
`abs`	`JLE addr16`	`0xE1`	3	7	*none*

The opposite of `JGT`: it jumps when `C` **or** `Z` is set. Also its own opcode, not an alias.

See also: `JGT`, `CMP`, `JLT`

JLT

Jumps when A was less than the operand (unsigned).

Alias for `JC` — the assembler emits the same opcode; encoding and timing are identical.

JLTS

Jumps when the value is less — signed.

Mode	Syntax	Opcode	Bytes	T	Flags
`abs`	`JLTS addr16`	`0xE3`	3	7	*none*

Jumps when `N` and `V` differ. Like `JGES` it pairs with `SUB` or `SBC`, not with `CMP`.

```
        LDA #-100       ; 0x9C
        SUB #100        ; overflows: N=0, V=1 -> N!=V
        JLTS smaller    ; taken, and correctly so
```

Without `JLTS` a signed comparison would have to be written as a pair of jumps on `N` and `V` — which is why this instruction was added in ISA 3.1.

See also: `JGES`, `SUB`, `JN`

JMP

Unconditional jump.

Mode	Syntax	Opcode	Bytes	T	Flags
`abs`	`JMP addr16`	`0xC8`	3	7	*none*
`(zp)`	`JMP (addr8)`	`0xC9`	2	9	*none*

Two forms. The direct one jumps to the address written in the instruction. The indirect one (`JMP (zp)`) takes the target from a pointer in the zero page — this is how a jump table or a state machine is built:

```
        LDA state
        SHL             ; index x 2 (an address is 2 B)
        TAX
        LDA table,X
        STA target
        LDA table+1,X
        STA target+1
        JMP (target)
```

No flag changes.

See also: `CALL`, `JZ`, `JNZ`

JN

Jumps when the N flag is set (the top bit of the result).

Mode	Syntax	Opcode	Bytes	T	Flags
`abs`	`JN addr16`	`0xC5`	3	7	*none*

After an operation on signed numbers `N = 1` means negative. After `CMP`, `JN` is reliable only when no overflow occurred — the general signed test is `JLTS`.

See also: `JP`, `JLTS`, `JGES`

JNC

Jumps when C is clear. After a compare this means greater or equal.

Mode	Syntax	Opcode	Bytes	T	Flags
`abs`	`JNC addr16`	`0xC2`	3	7	*none*

The opposite of `JC`. The readable name for use after a comparison is the alias `JGE`.

See also: `JC`, `JGE`, `CMP`

JNE

Jumps when the compared values were not equal.

Alias for `JNZ` — the assembler emits the same opcode; encoding and timing are identical.

JNV

Jumps when the overflow flag is clear.

Mode	Syntax	Opcode	Bytes	T	Flags
`abs`	`JNV addr16`	`0xC6`	3	7	*none*

The opposite of `JV`. Used after signed arithmetic to check that the result fitted.

See also: `JV`, `ADD`, `SUB`

JNZ

Jumps when Z is clear — that is, when the last result was not zero.

Mode	Syntax	Opcode	Bytes	T	Flags
`abs`	`JNZ addr16`	`0xC0`	3	7	*none*

Together with `JZ` the most used conditional jump on the machine — in the repository's sources they account for roughly two thirds of all conditional jumps. It closes counted loops:

```
        LDX #10
loop:   ; ... body ...
        DEX
        JNZ loop
```

After a comparison it reads as "not equal" and has the alias `JNE` for that case.

See also: `JZ`, `JNE`, `DEC`, `DEX`

JP

Jumps when N is clear — that is, when the result is positive or zero.

Mode	Syntax	Opcode	Bytes	T	Flags
`abs`	`JP addr16`	`0xC4`	3	7	*none*

Short for **jump if positive**.

> **Pitfall**
>
> On some other processors — the Z80, for one — `JP` means an unconditional jump. Here it is a **conditional** jump on the sign bit; the unconditional jump is called `JMP`. It is the easiest typo in the whole instruction set, because it assembles without complaint.

See also: `JN`, `JMP`, `JGES`

JV

Jumps when the overflow flag is set.

Mode	Syntax	Opcode	Bytes	T	Flags
`abs`	`JV addr16`	`0xC7`	3	7	*none*

`V` is set by addition and subtraction and means the result is wrong when the operands are read as signed. Logic operations, `CMP` and the shifts leave `V` alone.

See also: `JNV`, `ADD`, `SUB`, `CMP`

JZ

Jumps when Z is set — that is, when the last result was zero.

Mode	Syntax	Opcode	Bytes	T	Flags
`abs`	`JZ addr16`	`0xC1`	3	7	*none*

After a comparison it reads as "equal" and has the alias `JEQ` for that case. It is useful without a comparison too, because `Z` is set by almost every operation including a load into a register:

```
        LDA state
        JZ  nothing_to_do   ; no CMP #0 needed
```

See also: `JNZ`, `JEQ`, `LDA`, `BIT`

LDA

Loads a byte into the accumulator. The only instruction supporting all eight addressing modes.

Mode	Syntax	Opcode	Bytes	T	Flags
`#n`	`LDA #n`	`0x40`	2	5	`Z N`
`zp`	`LDA addr8`	`0x41`	2	6	`Z N`
`abs`	`LDA addr16`	`0x42`	3	9	`Z N`
`abs,X`	`LDA addr16,X`	`0x43`	3	11	`Z N`
`abs,Y`	`LDA addr16,Y`	`0x44`	3	11	`Z N`
`(zp)`	`LDA (addr8)`	`0x45`	2	10	`Z N`
`zp,X`	`LDA addr8,X`	`0x46`	2	7	`Z N`
`(zp),Y`	`LDA (addr8),Y`	`0x47`	2	13	`Z N`

`LDA` is the most used instruction on the machine — together with `STA` it makes up roughly half of all instructions in real programs. It supports the complete set of addressing modes, which makes it the best place to see them together.

```
        LDA #42         ; a constant
        LDA value       ; from memory (assembler picks zp/abs)
        LDA table,X     ; an array element
        LDA (pointer)   ; through a zero-page pointer
        LDA (row),Y     ; through a pointer, plus an offset in Y
```

It sets `Z` and `N` from the loaded byte, so you can branch straight afterwards without comparing:

```
        LDA state
        JZ  nothing_to_do
```

What the modes cost

The difference between modes is in the T-state column above and is worth remembering: the zero page is three T-states cheaper than an absolute address, because neither the second address byte nor its handling is needed. The dearest is `(zp),Y` — reading a pointer and adding an index to it takes more than twice as long as `zp`.

Microcode — `LDA zp  (0x41)`

```
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [ORO,MI] [RO,AI,FI]  (6)
```

Microcode — `LDA (zp),Y  (0x47)`

```
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [ORO,MI] [RO,ORI] [YO,TI] [MRI]
[RO,OHI] [ATM,AOP,EO,MI,ACO] [OHO,TI] [ATM,AHC,EO,MIH] [RO,AI,FI]
(13)
```

The second listing shows where that time goes: eight steps after the fetch only assemble the address, and the ninth finally reads the value. The `ACO` and `AHC` pair is the carry between address bytes — `ACO` keeps the carry out of the low byte in the **`addrCarry`** latch and `AHC` adds it to the high one. That is how the processor manages a 16-bit sum in 8-bit pieces without needing a programmer's register.

> **Note**
>
> This is exactly why the zero page is described as a "pointer file" in SAP-3/16. It is not merely faster memory — it is the only place pointers for the `(zp)` and `(zp),Y` modes can live.

> **Pitfall**
>
> The hash on a constant is not optional. `LDA 42` means "load the contents of address 42", `LDA #42` means "load the number 42". The assembler translates both without warning, because both are legal.

> **Pitfall**
>
> `LDA` changes neither `C` nor `V`. Loading a value therefore does not destroy the result of a previous comparison — which can be exploited, but also means `C` may be "stale" if you rely on it after a long run of moves.

See also: `STA`, `LDX`, `LDY`, `BIT`

LDX

Loads a byte into register X.

Mode	Syntax	Opcode	Bytes	T	Flags
`#n`	`LDX #n`	`0x48`	2	5	`Z N`
`zp`	`LDX addr8`	`0x49`	2	6	`Z N`
`abs`	`LDX addr16`	`0x4A`	3	9	`Z N`
`abs,Y`	`LDX addr16,Y`	`0x4C`	3	11	`Z N`

Sets `Z` and `N`. The choice of addressing modes is narrower than `LDA` — and the asymmetry has a logic: `LDX` can index with **Y**, not with itself.

```
        LDX #0
        LDX length
        LDX table,Y     ; indexed by the other index register
```

> **Pitfall**
>
> There is no `LDX zp,X` and no `LDX zp,Y` — the `zp,Y` mode does not exist in the ISA at all. If you need a short indexed form, only `LDY` has one (`LDY zp,X`).

See also: `LDY`, `STX`, `TAX`, `TXA`

LDY

Loads a byte into register Y.

Mode	Syntax	Opcode	Bytes	T	Flags
`#n`	`LDY #n`	`0x50`	2	5	`Z N`
`zp`	`LDY addr8`	`0x51`	2	6	`Z N`
`abs`	`LDY addr16`	`0x52`	3	9	`Z N`
`abs,X`	`LDY addr16,X`	`0x53`	3	11	`Z N`
`zp,X`	`LDY addr8,X`	`0x56`	2	7	`Z N`

The mirror image of `LDX`, but with a different set of modes: `LDY` can index with **X**, including in the short `zp,X` form.

See also: `LDX`, `STY`, `TAY`, `TYA`

LXY

Loads a 16-bit constant into the X:Y pair. Changes no flags.

Mode	Syntax	Opcode	Bytes	T	Flags
`#n`	`LXY #nn`	`0xEA`	3	8	*none*

`LXY #imm16` splits a 16-bit constant into two bytes: the high one goes to **X**, the low one to **Y**. It is the same convention `TXYS` uses for the stack pointer — **X** holds the high half.

Without this instruction setting up an address would take two `LDA` / `TAX` / `LDY` sequences; this way it is one three-byte instruction.

```
        LXY #buffer     ; X:Y := the buffer's address
        SXY ptr         ; and store it as a zero-page pointer
```

No flag changes.

See also: `SXY`, `TXYS`, `TSXY`, `ADW`

MUL

Multiplies A by B: the low byte of the product into A, the high one into B.

Mode	Syntax	Opcode	Bytes	T	Flags
implied	`MUL`	`0x28`	1	4	`C Z N`

Like `DIV`, `MUL` takes no operand — both factors come from **A** and **B**, and the second gets there with `TAB`.

```
        LDA width
        TAB             ; B := width
        LDA height
        MUL             ; A := low byte, B := high byte
        STA area_lo
        TBA
        STA area_hi
```

`C` is set exactly when the high byte is non-zero — that is, when the product did not fit in one byte. `Z` and `N` follow the **low** byte.

> **Pitfall**
>
> `Z` after `MUL` does not mean "the product is zero". It means "the low byte of the product is zero", which holds for 16 × 16 = 256, among others. A test for a genuine zero must take `C` or the high byte into account.

See also: `DIV`, `TAB`, `TBA`, `SHL`

NOP

Does nothing. Takes three T-states.

Mode	Syntax	Opcode	Bytes	T	Flags
implied	`NOP`	`0x01`	1	3	*none*

The only instruction with no effect. It is used to fill space (after removing code, so addresses need not be recomputed) and for fine timing.

> **Note**
>
> `NOP` takes 3 T, not 2. The reason is structural: the first two microcode steps are the fetch, which belongs to the **following** instruction, and no microprogram may be shorter than three steps. `NOP` therefore carries one empty step after the fetch.
>
> No shorter instruction exists in the ISA — 3 T is the floor for anything.

See also: `HLT`

NOT

Inverts every bit of the accumulator.

Mode	Syntax	Opcode	Bytes	T	Flags
implied	`NOT`	`0x0E`	1	3	`Z N`

`A := ~A`. Sets `Z` and `N`; `C` and `V` do not change.

Together with `ADD #1` it forms a sign change:

```
        NOT
        ADD #1          ; A := -A (two's complement)
```

See also: `AND`, `OR`, `XOR`, `SUB`

OR

Bitwise OR of the accumulator with the operand.

Mode	Syntax	Opcode	Bytes	T	Flags
`#n`	`OR #n`	`0xA8`	2	6	`Z N`
`zp`	`OR addr8`	`0xA9`	2	7	`Z N`
`abs`	`OR addr16`	`0xAA`	3	10	`Z N`
`abs,X`	`OR addr16,X`	`0xAB`	3	12	`Z N`
`abs,Y`	`OR addr16,Y`	`0xAC`	3	12	`Z N`
`(zp)`	`OR (addr8)`	`0xAD`	2	11	`Z N`
`zp,X`	`OR addr8,X`	`0xAE`	2	8	`Z N`
`(zp),Y`	`OR (addr8),Y`	`0xAF`	2	14	`Z N`

`A := A | operand`. Sets `Z` and `N`, leaves `C` and `V` alone.

It is used to set bits and — in pairs — to test whether a 16-bit value is zero:

```
        LDA length
        OR  length+1    ; Z = 1 exactly when both bytes are zero
        JZ  empty
```

See also: `AND`, `XOR`, `NOT`

OUT

Copies the accumulator to the output register.

Mode	Syntax	Opcode	Bytes	T	Flags
implied	`OUT`	`0x03`	1	3	*none*

`OUT` sends the contents of **A** to the OUT register — a simple output port shown on the emulator's display and captured by the `npm run asm` tool. It is the oldest way to get something out of a program and dates back to the original SAP machines.

Changes no flags.

```
        LDA result
        OUT             ; show the number on the display
        HLT
```

> **Note**
>
> Unlike the terminal, `OUT` is purely numeric and one byte wide — no queue, no waiting, no ASCII translation. For text there is the terminal at `TERM_DATA`.

See also: `HLT`

PHB

Pushes register B onto the stack.

Mode	Syntax	Opcode	Bytes	T	Flags
implied	`PHB`	`0x2A`	1	4	*none*

Without this instruction **B** could only be saved through the accumulator, destroying it. `PHB` was therefore added in ISA 3.1 and is essential in interrupt handlers: every binary ALU operation (even `CMP #0`) overwrites **B**, so a handler that touches the ALU must rescue it.

```
handler:
        PUSH            ; A
        PHB             ; B
        ; ... body ...
        PLB
```

```
        POP
        RTI
```

Changes no flags.

See also: `PLB` , `PUSH` , `TAB` , `RTI`

PHF

Pushes the status byte (the flags) onto the stack.

Mode	Syntax	Opcode	Bytes	T	Flags
implied	`PHF`	`0x26`	1	4	*none*

Pushes `C` , `Z` , `N` , `V` and `I` packed into one byte — the same format interrupt entry pushes.

Changes no flags.

See also: `PLF` , `RTI` , `PUSH`

PHX

Pushes register X onto the stack.

Mode	Syntax	Opcode	Bytes	T	Flags
implied	`PHX`	`0x22`	1	4	*none*

Changes no flags.

See also: `PLX` , `PHY` , `PUSH`

PHY

Pushes register Y onto the stack.

Mode	Syntax	Opcode	Bytes	T	Flags
implied	`PHY`	`0x24`	1	4	*none*

Changes no flags.

See also: `PLY` , `PHX` , `PUSH`

PLB

Pops register B off the stack.

Mode	Syntax	Opcode	Bytes	T	Flags
implied	`PLB`	`0x2B`	1	4	`Z N`

The counterpart of `PHB`. Sets `Z` and `N` from the popped byte.

> **Pitfall**
>
> That `PLB` sets flags can surprise inside an interrupt handler. It only seems to matter, though: `RTI` immediately afterwards restores the whole status byte from the stack, so the change is discarded. It would be a problem only if `PLB` were used outside a handler between a compare and a jump.

See also: `PHB`, `POP`, `TBA`

PLF

Restores the flags from the stack.

Mode	Syntax	Opcode	Bytes	T	Flags
implied	`PLF`	`0x27`	1	4	`C Z N V I`

Pops a byte and unpacks **all five** flags from it, including `I` — unlike `POP`, which merely derives `Z` and `N` from the value. It is the same mechanism `RTI` uses.

> **Pitfall**
>
> `PLF` restores `I` as well, so it can unexpectedly enable or disable interrupts. Anyone saving the flags in the middle of a critical section must expect to get the whole lot back on restore.

See also: `PHF`, `RTI`, `POP`

PLX

Pops register X off the stack.

Mode	Syntax	Opcode	Bytes	T	Flags
implied	`PLX`	`0x23`	1	4	`Z N`

Sets `Z` and `N` from the popped value.

See also: `PHX`, `PLY`, `POP`

PLY

Pops register Y off the stack.

Mode	Syntax	Opcode	Bytes	T	Flags
implied	`PLY`	`0x25`	1	4	`Z N`

Sets `Z` and `N` from the popped value.

See also: `PHY`, `PLX`, `POP`

POP

Pops the accumulator off the stack.

Mode	Syntax	Opcode	Bytes	T	Flags
implied	`POP`	`0x21`	1	4	`Z N`

Reads the byte at the top of the stack into **A** and increments **SP**. Sets `Z` and `N` from the popped value.

See also: `PUSH`, `PLB`, `PLX`, `PLY`

PUSH

Pushes the accumulator onto the stack.

Mode	Syntax	Opcode	Bytes	T	Flags
implied	`PUSH`	`0x20`	1	4	*none*

Lowers **SP** and writes the contents of **A** to the new address. A push is therefore **pre-decrement** — afterwards **SP** points directly at the byte written.

Changes no flags.

```
        PUSH            ; put A aside
        ; ... something that overwrites A ...
        POP             ; and get it back
```

See also: `POP`, `PHX`, `PHY`, `PHB`, `PHF`

RET

Returns from a subroutine.

Mode	Syntax	Opcode	Bytes	T	Flags
implied	`RET`	`0xCB`	1	6	*none*

Pops the low and high bytes of the return address off the stack and jumps to it. Changes no flags, so a subroutine can return a result not only in a register but in the flags:

```
is_empty:
        LDA length
        OR  length+1
        RET             ; Z carries the answer
```

> **Pitfall**
>
> `RET` does not check that a return address is actually on the stack. If a subroutine pushes something and forgets to pop it, `RET` jumps into data — and the program usually runs away somewhere else entirely, reporting an illegal opcode at a completely different place.

See also: `CALL`, `RTI`, `PUSH`, `POP`

RND

Puts a random byte into the accumulator.

Mode	Syntax	Opcode	Bytes	T	Flags
implied	`RND`	`0x0F`	1	3	*none*

The source is an xorshift32 generator whose seed can be set — so a run is reproducible. The `npm run asm` tool has a `--seed` switch for that.

```
        RND
        AND #0x0F       ; a random number 0-15
```

> **Pitfall**

`RND` sets no flag, not even `Z`. A test for a particular value therefore needs `CMP` — a bare `RND` followed by `JZ` branches on the result of the **previous** instruction.

Note

The generator is part of the processor's state and is journalled. Time travel therefore rewinds the random numbers too — replaying the same stretch gives the same values.

See also: `AND`, `CMP`

ROL

Rotate left through the C flag: the accumulator, or a byte in memory.

Mode	Syntax	Opcode	Bytes	T	Flags
implied	`ROL`	`0x0C`	1	3	`C Z N`
`zp`	`ROL addr8`	`0xDC`	2	7	`C Z N`
`abs`	`ROL addr16`	`0xDD`	3	10	`C Z N`

Bit 7 goes into `C` and `C` goes into bit 0. Unlike `SHL` nothing is lost — the value turns in a circle through the carry flag.

The main use is shifting numbers wider than a byte. `SHL` on the lowest byte pushes a bit into `C` and `ROL` on the higher ones pulls it back in:

```
        SHL value       ; low byte, the bit shifts out into C
        ROL value+1     ; high byte, C pulls it in
```

Sets `C`, `Z` and `N`.

See also: `ROR`, `SHL`, `SHR`, `CLC`, `SEC`

ROR

Rotate right through the C flag: the accumulator, or a byte in memory.

Mode	Syntax	Opcode	Bytes	T	Flags
implied	`ROR`	`0x0D`	1	3	`C Z N`
`zp`	`ROR addr8`	`0xDE`	2	7	`C Z N`
`abs`	`ROR addr16`	`0xDF`	3	10	`C Z N`

Bit 0 goes into `C` and `C` goes into bit 7. A multi-byte right shift is assembled the other way round from `ROL` — from the highest byte down to the lowest:

```
        SHR value+1     ; high byte
        ROR value       ; low byte
```

Sets `C`, `Z` and `N`.

See also: `ROL`, `SHR`, `SHL`

RTI

Returns from an interrupt handler.

Mode	Syntax	Opcode	Bytes	T	Flags
implied	`RTI`	`0xCC`	1	8	`C Z N V I`

Pops the status byte off the stack and then the low and high bytes of the return address — exactly what interrupt entry, or the `BRK` instruction, put there.

It restores **all five** flags, `I` included.

> **Note**
>
> `RTI` does not enable interrupts outright. That they are enabled after a return follows from the saved status byte still having `I` set — the `DIF` signal takes effect during the push only after the byte has left the bus.
>
> In practice this means a handler may use `DI` and `EI` internally as it pleases; after `RTI` the state that held **before** the interrupt applies.

See also: `BRK`, `EI`, `DI`, `RET`, `PLF`

SBC

Subtracts the operand from the accumulator including the borrow. The higher bytes of a multi-byte difference.

Mode	Syntax	Opcode	Bytes	T	Flags
`#n`	`SBC #n`	`0x98`	2	6	`C Z N V`
`zp`	`SBC addr8`	`0x99`	2	7	`C Z N V`
`abs`	`SBC addr16`	`0x9A`	3	10	`C Z N V`
`abs,X`	`SBC addr16,X`	`0x9B`	3	12	`C Z N V`
`abs,Y`	`SBC addr16,Y`	`0x9C`	3	12	`C Z N V`
`(zp)`	`SBC (addr8)`	`0x9D`	2	11	`C Z N V`
`zp,X`	`SBC addr8,X`	`0x9E`	2	8	`C Z N V`
`(zp),Y`	`SBC (addr8),Y`	`0x9F`	2	14	`C Z N V`

`A := A − operand − C`. Here `C` is a **borrow**, not a carry.

The division of labour is the same as with `ADD` / `ADC`: subtract the lowest byte with `SUB`, the higher ones with `SBC`.

```
        LDA a_lo
        SUB b_lo        ; C := borrow
        STA diff_lo
        LDA a_hi
        SBC b_hi
        STA diff_hi
```

Sets `C`, `Z`, `N` and `V`.

See also: `SUB`, `ADC`, `SBW`, `CMP`

SBW

Subtracts an 8-bit constant from a 16-bit word in memory. Changes no flags.

Mode	Syntax	Opcode	Bytes	T	Flags
`zp`	`SBW addr8,#n`	`0xE9`	3	13	*none*

The counterpart of `ADW`. It subtracts the constant from the two-byte word at `zp` and writes the result back; the borrow between bytes is handled inside the instruction.

```
        SBW ptr, #40    ; one screen row up
```

No flag changes.

See also: `ADW` , `DEW` , `SBC`

SEC

Sets the carry flag.

Mode	Syntax	Opcode	Bytes	T	Flags
implied	`SEC`	`0x05`	1	3	`C`

`C := 1` . Used mainly to prepare the input bit for `ROL` or `ROR` . It is not needed before a subtraction — `SUB` ignores the incoming borrow.

See also: `CLC` , `ROL` , `ROR` , `SBC`

SHL

Shift left by one bit: the accumulator, or a byte in memory.

Mode	Syntax	Opcode	Bytes	T	Flags
implied	`SHL`	`0x0A`	1	3	`C Z N`
`zp`	`SHL addr8`	`0xD8`	2	7	`C Z N`
`abs`	`SHL addr16`	`0xD9`	3	10	`C Z N`

Bit 7 goes into `C` and a zero is shifted in from the right. It is a multiplication by two — and the cheapest way to multiply by a power of two on this machine.

```
        LDA index
        SHL             ; x 2
        SHL             ; x 4
```

Sets `C` , `Z` and `N` .

> **Note**
>
> The memory forms (`SHL zp` , `SHL abs`) work directly in memory and never touch the accumulator. They exist precisely for multi-byte shifts — the pair `SHL` low byte, `ROL` high byte is the basic building block of 16-bit arithmetic.

See also: `SHR` , `ROL` , `MUL`

SHR

Shift right by one bit: the accumulator, or a byte in memory.

Mode	Syntax	Opcode	Bytes	T	Flags
implied	`SHR`	`0x0B`	1	3	`C Z N`
`zp`	`SHR addr8`	`0xDA`	2	7	`C Z N`
`abs`	`SHR addr16`	`0xDB`	3	10	`C Z N`

Bit 0 goes into `C` and a zero is shifted in from the left. It is an unsigned integer division by two.

Sets `C`, `Z` and `N`.

Pitfall

The shift is **logical**, not arithmetic — the top bit is not replicated. For signed numbers `SHR` therefore does not divide by two correctly: it turns `0xFE` (−2) into `0x7F` (127), not `0xFF` (−1).

A signed shift has to be assembled by hand. The trick is to get the original bit 7 into `C` and then use `ROR`, which pushes it back to the top:

```
        CLC
        BIT #0x80       ; Z = 1 exactly when bit 7 is clear
        JZ  .positive
        SEC             ; negative: C := 1
.positive: ROR          ; shift, the sign returns into bit 7
```

This is also one of the few places where `CLC` and `SEC` genuinely earn their keep.

See also: `SHL`, `ROR`, `DIV`

STA

Stores the accumulator to memory.

Mode	Syntax	Opcode	Bytes	T	Flags
`zp`	`STA addr8`	`0x61`	2	6	*none*
`abs`	`STA addr16`	`0x62`	3	9	*none*
`abs,X`	`STA addr16,X`	`0x63`	3	11	*none*
`abs,Y`	`STA addr16,Y`	`0x64`	3	11	*none*
`(zp)`	`STA (addr8)`	`0x65`	2	10	*none*
`zp,X`	`STA addr8,X`	`0x66`	2	7	*none*
`(zp),Y`	`STA (addr8),Y`	`0x67`	2	13	*none*

The other half of the most common pair of instructions. It supports all seven memory modes — only the immediate is missing, which would make no sense.

```
        STA value
        STA screen,X
        STA (row),Y
```

Changes no flag. That matters: a store between a compare and a jump does not destroy the compare's result.

See also: `LDA`, `STX`, `STY`

STX

Stores register X to memory.

Mode	Syntax	Opcode	Bytes	T	Flags
`zp`	`STX addr8`	`0x69`	2	6	*none*
`abs`	`STX addr16`	`0x6A`	3	9	*none*

Only two modes: `zp` and `abs`. Changes no flags.

> **Pitfall**
>
> `STX table,Y` does not exist. Storing an indexed element has to go through the accumulator (`TXA` then `STA table,Y`), which destroys **A**.

See also: `LDX`, `STY`, `STA`, `TXA`

STY

Stores register Y to memory.

Mode	Syntax	Opcode	Bytes	T	Flags
`zp`	`STY addr8`	`0x71`	2	6	*none*
`abs`	`STY addr16`	`0x72`	3	9	*none*

Only `zp` and `abs`, like `STX`. Changes no flags.

See also: `LDY`, `STX`, `STA`, `TYA`

SUB

Subtracts the operand from the accumulator. Ignores the incoming borrow, sets the outgoing one.

Mode	Syntax	Opcode	Bytes	T	Flags
`#n`	`SUB #n`	`0x90`	2	6	`C Z N V`
`zp`	`SUB addr8`	`0x91`	2	7	`C Z N V`
`abs`	`SUB addr16`	`0x92`	3	10	`C Z N V`
`abs,X`	`SUB addr16,X`	`0x93`	3	12	`C Z N V`
`abs,Y`	`SUB addr16,Y`	`0x94`	3	12	`C Z N V`
`(zp)`	`SUB (addr8)`	`0x95`	2	11	`C Z N V`
`zp,X`	`SUB addr8,X`	`0x96`	2	8	`C Z N V`
`(zp),Y`	`SUB (addr8),Y`	`0x97`	2	14	`C Z N V`

`A := A - operand`. After the operation `C` means a **borrow** — it is set exactly when the subtraction underflowed.

Sets `C`, `Z`, `N` and `V`. That `V` is precisely why the signed jumps `JGES` and `JLTS` pair with `SUB` rather than with `CMP`.

```
        LDA a
        SUB b
        JC  underflowed
```

See also: `SBC`, `ADD`, `CMP`, `JLTS`, `JGES`

SXY

Stores the X:Y pair as a 16-bit word into the zero page. Changes no flags.

Mode	Syntax	Opcode	Bytes	T	Flags
`zp`	`SXY addr8`	`0xEB`	2	8	*none*

`SXY zp` writes **Y** to address `zp` and **X** to `zp+1` — low byte first, the way every word is stored on this machine.

Together with `LXY` it is the shortest way to set up a pointer:

```
        LXY #buffer
        SXY ptr          ; ptr now points at the buffer
        LDA (ptr),Y
```

See also: `LXY`, `TXYS`, `TSXY`, `INW`

TAB

Copies the accumulator into register B.

Mode	Syntax	Opcode	Bytes	T	Flags
implied	`TAB`	`0x1C`	1	3	`Z N`

The only way to get something into **B** on purpose. It is used to stage the operand for `MUL` and `DIV`, which take their second factor from there.

```
        LDA divisor
        TAB
        LDA dividend
        DIV
```

Sets `Z` and `N` from the transferred value.

> **Pitfall**
>
> A value in **B** is short-lived. **Every** binary ALU operation overwrites it — `ADD`, `SUB`, `AND`, `OR`, `XOR`, `CMP`, `BIT` and `ADW` included. Between `TAB` and `MUL` / `DIV` there must therefore be nothing that touches the ALU. To carry **B** across a longer stretch, push it with `PHB`.

See also: `TBA`, `MUL`, `DIV`, `PHB`, `PLB`

TAX

Copies the accumulator into register X.

Mode	Syntax	Opcode	Bytes	T	Flags
implied	`TAX`	`0x18`	1	3	`Z N`

Sets `Z` and `N`. The usual way to get a computed index where an addressing mode can use it:

```
        LDA state
        SHL
        TAX
        LDA table,X
```

See also: `TXA`, `TAY`, `LDX`

TAY

Copies the accumulator into register Y.

Mode	Syntax	Opcode	Bytes	T	Flags
implied	`TAY`	`0x1A`	1	3	`Z N`

Sets `Z` and `N`.

See also: `TYA`, `TAX`, `LDY`

TBA

Copies register B into the accumulator.

Mode	Syntax	Opcode	Bytes	T	Flags
implied	`TBA`	`0x1D`	1	3	`Z N`

The way to collect a secondary result: after `MUL` **B** holds the high byte of the product, after `DIV` the remainder.

```
        DIV
        TBA             ; A := remainder
```

Sets `Z` and `N`.

See also: `TAB`, `MUL`, `DIV`

TSXY

Reads the stack pointer into the X:Y pair. Changes no flags.

Mode	Syntax	Opcode	Bytes	T	Flags
implied	`TSXY`	`0x1F`	1	4	*none*

X receives the high half of **SP**, **Y** the low one — the same convention as `LXY` and `TXYS`.

It is used to save the stack's state, or to reach parameters passed on the stack.

> **Note**
>
> The transfer takes two T-states (a 16-bit value over an 8-bit bus), but interrupts are only taken at an instruction boundary — so a handler can never see **SP** half-updated.

See also: `TXYS`, `LXY`, `SXY`

TXA

Copies register X into the accumulator.

Mode	Syntax	Opcode	Bytes	T	Flags
implied	`TXA`	`0x19`	1	3	`Z N`

Sets `Z` and `N`. A necessary step wherever an indexed store form is missing — `STX table,Y` does not exist, for instance, so it has to be done through **A**.

See also: `TAX`, `TYA`, `STX`

TXYS

Sets the stack pointer from the X:Y pair. Changes no flags.

Mode	Syntax	Opcode	Bytes	T	Flags
implied	`TXYS`	`0x1E`	1	4	*none*

`SP := X:Y`, where **X** is the high half. The mirror of `TSXY`.

It is used at initialisation, when a program wants the stack somewhere other than the default, and when recovering from an error, where the stack has to be tidied in one go.

```
        LXY #0x0200
        TXYS            ; SP back to its reset value
```

Pitfall

`TXYS` checks nothing. Setting **SP** into the middle of data is legal and only shows up when the first `CALL` overwrites whatever was there.

See also: `TSXY`, `LXY`, `PUSH`, `POP`

TYA

Copies register Y into the accumulator.

Mode	Syntax	Opcode	Bytes	T	Flags
implied	`TYA`	`0x1B`	1	3	`Z N`

Sets `Z` and `N`.

See also: `TAY`, `TXA`, `STY`

XOR

Bitwise exclusive OR of the accumulator with the operand.

Mode	Syntax	Opcode	Bytes	T	Flags
`#n`	`XOR #n`	`0xB0`	2	6	`Z N`
`zp`	`XOR addr8`	`0xB1`	2	7	`Z N`
`abs`	`XOR addr16`	`0xB2`	3	10	`Z N`
`abs,X`	`XOR addr16,X`	`0xB3`	3	12	`Z N`
`abs,Y`	`XOR addr16,Y`	`0xB4`	3	12	`Z N`
`(zp)`	`XOR (addr8)`	`0xB5`	2	11	`Z N`
`zp,X`	`XOR addr8,X`	`0xB6`	2	8	`Z N`
`(zp),Y`	`XOR (addr8),Y`	`0xB7`	2	14	`Z N`

`A := A ^ operand`. Sets `Z` and `N`, leaves `C` and `V` alone.

Two usual purposes: flipping selected bits with a mask, and an equality test that also says **which bits** differ.

```
        LDA state
        XOR #0b00000001 ; flip the lowest bit
        STA state
```

> **Note**
>
> Anyone who knows other architectures will look for the "zero a register with itself" trick here. It does not work — `XOR` has no register-operand form, so there is no way to write "`XOR` with **A**". Zeroing is simply `LDA #0`, which is the same length and faster than anything else.

See also: `AND`, `OR`, `NOT`, `BIT`

PART IV

Peripherals and the System

CHAPTER 16

The MMIO Map and Access Rules

Peripherals in SAP-3/16 have no instructions of their own. They sit in memory — in the block 0xFF00–0xFF7F — and you talk to them with ordinary `LDA` and `STA`. The processor intercepts those addresses before they become memory and hands them to a device.

The practical consequence: everything you can do with memory you can do with a peripheral. A device register can be addressed with an index, you can put a watchpoint on it, and its value shows up in a memory dump.

How the address is decoded

The block is 128 bytes, split into eight devices of sixteen. The device number is address bits 6 to 4:

```
0xFF00 + (device << 4) + register
```

Base	No.	Device
`0xFF00`	0	Keyboard
`0xFF10`	1	Terminal
`0xFF20`	2	Audio
`0xFF30`	3	Timer and counters
`0xFF40`	4	Interrupt control
`0xFF50`	5	Video
`0xFF60`	6	Storage and serial
`0xFF70`	7	Gamepads and blitter

Figure 11. How the peripheral block splits into eight devices.

Not every device fills its sixteen bytes, and the spare room has come in handy twice: the counters live in the upper half of the timer's slot, the serial line in the upper half of storage's, and the blitter in the upper half of the gamepads'.

> **Note**

Unassigned addresses inside the block read zero and ignore writes. A program therefore has no way to notice it reached beside a register — which is why registers are always named with `EQU` constants rather than written as numbers.

Rules that hold across devices

The devices differ, but a few conventions are shared and pay to know before the individual registers.

Error bits are read once

Status registers with an error bit (`DSK_STATUS`, `BLT_STATUS`, `SER_STATUS`, `TERM_STATUS`) are **cleared by reading**. A program must therefore read the register once, keep the value and test every bit from that copy:

```
.wait:  LDA DSK_STATUS
        STA tmp             ; one read, both bits from it
        AND #0x01
        JZ  .wait
        LDA tmp
        AND #0x02
        JNZ error
```

Pitfall

The most common mistake when working with peripherals is reading a status register twice — once to test readiness, once to test the error. The second read no longer sees the error, because the first one cleared it.

The transmitter is not always ready

Both the terminal and the serial line have a queue that drains at its own pace. A write into a full queue **drops the byte** and sets an overflow bit. A short burst passes without asking; sustained output does not — the ready bit should be read before every write.

Transfers take time

Storage acknowledges a command immediately, but the data moves only after a while. A program must wait on the ready bit rather than count cycles. The blitter, by contrast, is finished inside the instruction that started it.

Level versus edge

The distinction input design rests on:

Kind	Behaviour
level	the register shows the **current** state; reading does not change it (gamepads, ready bits)
edge	the event is held in a queue until the program takes it; reading consumes it (keyboard)

A game therefore reads the gamepad every frame and does not care about history, whereas text input picks characters out of a queue and must not miss one.

What is and is not part of machine state

Almost everything in the machine is journalled and therefore rewinds with time travel. Three things do not, all for the same reason — they are **media or the outside world**, not processor state:

Does not rewind	Why
the disk's contents	stepping back past a disk write does not un-write it — as with real hardware
bytes already sent on the serial line	what went to the peer cannot be recalled
the link itself	the cable survives a reset and a rewind

Every device's registers, queue contents, in-flight transfer countdowns and the gamepad's held keys **are** part of machine state, so they replay exactly.

> **Note**
>
> That gives debugging an unusual property: if a program rewinds past sending a byte and sends it again, the peer sees it twice. The timeline branches on this side of the cable, not the other.

Device addresses

The rest of Part IV takes the devices one at a time. For quick reference, here is the complete register list by device.

Keyboard

Address	Register
`0xFF00`	`KBD_DATA`
`0xFF01`	`KBD_STATUS`

Interrupt control

Address	Register
`0xFF40`	`IRQ_ENABLE`
`0xFF41`	`IRQ_STATUS`

Terminal

Address	Register
0xFF10	TERM_DATA
0xFF11	TERM_STATUS

Audio

Address	Register
0xFF20	AUD_CTRL
0xFF21	AUD_FREQ_LO
0xFF22	AUD_FREQ_HI
0xFF23	AUD_WAVE
0xFF24	AUD_AD
0xFF25	AUD_SR

Timer and counters

Address	Register
0xFF30	TMR_DIV
0xFF31	TMR_CNT
0xFF32	CYC0
0xFF33	CYC1
0xFF34	CYC2
0xFF35	CYC3
0xFF36	UPT_LO
0xFF37	UPT_HI
0xFF38	CLK_TPMS_LO
0xFF39	CLK_TPMS_HI
0xFF3A	CLK_MS_LO
0xFF3B	CLK_MS_HI

Video

Address	Register
0xFF50	VID_CTRL
0xFF51	VID_STATUS
0xFF52	VID_CURX
0xFF53	VID_CURY
0xFF54	VID_BORDER
0xFF55	VID_SCROLLX
0xFF56	VID_SCROLLY
0xFF57	VID_MCOLOR
0xFF58	SPR_ENABLE

Storage and serial

Address	Register
0xFF60	DSK_CMD
0xFF61	DSK_STATUS
0xFF62	DSK_BLK
0xFF63	DSK_ADDR_LO
0xFF64	DSK_ADDR_HI
0xFF65	SER_DATA
0xFF66	SER_STATUS
0xFF67	SER_CTRL
0xFF68	DSK_DISK

Gamepads and blitter

Address	Register
0xFF70	PAD1
0xFF71	PAD2
0xFF72	BLT_SRC_LO
0xFF73	BLT_SRC_HI
0xFF74	BLT_DST_LO
0xFF75	BLT_DST_HI
0xFF76	BLT_LEN_LO
0xFF77	BLT_LEN_HI

Address	Register
0xFF78	BLT_FILL
0xFF79	BLT_CTRL
0xFF7A	BLT_STATUS

CHAPTER 17

Keyboard and Gamepads

The machine has two input devices and they differ not in technology but in their **model of time**. The keyboard delivers events, the gamepad state. Which one is right for a job is settled by a single question: does it matter if the program misses a press?

The keyboard

Address	Register
`0xFF00`	`KBD_DATA`
`0xFF01`	`KBD_STATUS`

`KBD_DATA` holds the **ASCII code** of the key pressed. Not a scancode — the keyboard controller translates before the byte reaches the processor. Printable characters have their code, Enter is 0x0D, Backspace 0x08, Escape 0x1B.

They are **key-down events only**. A release is not reported anywhere and the duration of a press is not visible at all.

`KBD_STATUS` bit 0 says "an unread character is waiting". Reading `KBD_DATA` consumes the character — it empties the buffer, drops the flag and clears any pending keyboard interrupt.

```
read_key:
        LDA KBD_STATUS
        AND #0x01
        JZ  read_key        ; nothing waiting, keep waiting
        LDA KBD_DATA        ; and this consumes it
        RET
```

Pitfall

The buffer holds **one byte**. If the user presses a second key before the program reads the first, the first is gone for good. A program doing something lengthy between reads loses characters.

The remedy is either to read the keyboard from an interrupt (bit 1 of `IRQ_ENABLE`) or to accept that fast typing will not all be recorded.

The keyboard on an interrupt

The interrupt source is **level-sensitive**: it stays asserted while an unread character sits in the buffer. The handler must therefore read it, or the interrupt fires again immediately after the return.

```
        LDA #0b00000010     ; bit 1 = keyboard
        STA IRQ_ENABLE
        EI

handler:
        PUSH
        PHB
        LDA KBD_DATA        ; reading clears the source
        CALL enqueue
        PLB
        POP
        RTI
```

Gamepads

Address	Register
0xFF70	PAD1
0xFF71	PAD2
0xFF72	BLT_SRC_LO
0xFF73	BLT_SRC_HI
0xFF74	BLT_DST_LO
0xFF75	BLT_DST_HI
0xFF76	BLT_LEN_LO
0xFF77	BLT_LEN_HI
0xFF78	BLT_FILL
0xFF79	BLT_CTRL
0xFF7A	BLT_STATUS

Two independent 8-bit registers, one per player. A set bit means the button is **down right now**. The registers are level-sensitive — reading does not clear them and you may read them as often as you like.

Bit	Button	Bit	Button
0	up	4	Fire / A

Bit	Button	Bit	Button
1	down	5	B
2	left	6	Start
3	right	7	Select

The typical use is one read per frame:

```
        LDA PAD1
        AND #0x04           ; left?
        JZ  .not_left
        DEC player_x
.not_left:
```

A one-player game that should accept either controller merges them in a single instruction:

```
        LDA PAD1
        OR  PAD2
```

The gamepad **raises no interrupt**. It is meant to be polled and nothing else would make sense — a game draws every frame anyway.

> **Note**
>
> Why two input devices side by side? Because the keyboard cannot tell you a key is **held**. Key repeat on the host depends on the user's settings and, in the emulator, on their browser; smooth character movement would therefore depend on things the program cannot control. The gamepad removes that problem by reporting state rather than events.

In the emulator the host keyboard maps onto the ports by physical key: WASD, space and left shift to `PAD1`; the arrow keys, Enter and right shift to `PAD2`. On the board they are two physical connectors.

> **Note**
>
> Both registers are writable in the emulator as a test hook — a program can set a held state and read it back. On hardware they are read-only. The held state is part of the journal, so it replays exactly under time travel.

CHAPTER 18

Terminal and Serial Line

Two devices that send bytes out, and they are easy to mix up. The difference is who is at the other end: the terminal talks to **the human at this machine**, the serial line to **another machine**.

The terminal

Address	Register
`0xFF10`	`TERM_DATA`
`0xFF11`	`TERM_STATUS`

A write to `TERM_DATA` queues a character in a transmit FIFO of 64 bytes. The character reaches the screen when it **leaves** the queue — one entry per `max(1, CLK_TPMS/8)` T-states, which is roughly 8 000 characters per second of real time at any clock above 8 kHz.

```
        LDA #'A'
        STA TERM_DATA
```

`TERM_STATUS` is read-only:

Bit	Meaning
0	transmitter ready — the queue is **not** full
1	overflow: a write was dropped (sticky, cleared on read)

Pitfall

The transmitter is not always ready. A write into a full queue **drops the byte** and sets bit 1. A burst of up to 64 characters passes without asking; longer output does not.

A correctly written print routine asks:

```
putc:   LDA TERM_STATUS
        AND #0x01
```

```
        JZ  putc            ; queue full, wait
        LDA char
        STA TERM_DATA
        RET
```

Control characters

Code	Escape	Effect
0x0A	`\n`	new line
0x0D	`\r`	return to the start of the line
0x08	`\b`	erase the previous character
0x0C	`\f`	clear the terminal screen
0x09	`\t`	tab to the next 8-column stop

The terminal copes with every usual line-ending convention: `\n`, `\r\n` and `\n\r` each produce one new line. A lone `\r` returns the cursor to the start of the line, which is handy for overwriting — a progress indicator, say.

> **Note**
>
> The queue keeps draining after a `HLT`. Text written just before the stop is therefore still printed — exactly like a real serial line finishing its sentence after the processor stops working.

The serial line

Address	Register
`0xFF60`	`DSK_CMD`
`0xFF61`	`DSK_STATUS`
`0xFF62`	`DSK_BLK`
`0xFF63`	`DSK_ADDR_LO`
`0xFF64`	`DSK_ADDR_HI`
`0xFF65`	`SER_DATA`
`0xFF66`	`SER_STATUS`
`0xFF67`	`SER_CTRL`
`0xFF68`	`DSK_DISK`

The `SER_DATA`, `SER_STATUS` and `SER_CTRL` registers share a slot with the storage device — they live in its spare upper half.

The link is bidirectional, with FIFOs of 16 bytes in each direction.

`SER_STATUS` bit	Meaning
0	the receive FIFO is not empty (level)
1	the transmit FIFO is not full (level)
2	receive overrun: a byte from the peer was lost (sticky, cleared on read)
3	transmit overflow: a write was dropped (sticky, cleared on read)
4	a link is attached (level)

Reading `SER_DATA` pops the oldest received byte; an empty FIFO reads zero, so bit 0 should be tested first. A write queues a byte for sending.

`SER_CTRL` is a command register (reads zero): bit 0 flushes the receiver, bit 1 the transmitter. The bits combine.

```
send:   LDA SER_STATUS
        AND #0x02           ; transmitter ready?
        JZ  send
        LDA byte
        STA SER_DATA
        RET

recv:   LDA SER_STATUS
        AND #0x01           ; anything arrived?
        JZ  recv
        LDA SER_DATA
        RET
```

Interrupts

The serial line can raise an interrupt on receive — bit 2 of `IRQ_ENABLE`. The source is **level-sensitive**: it stays asserted while at least one byte sits in the receive FIFO, so the handler must drain it.

A “transmitter empty” interrupt deliberately does not exist. Sending is waited on by polling bit 1 — just as with a real UART.

Speed

One byte leaves the transmit FIFO every `max(128, CLK_TPMS/8)` T-states. At clocks up to 1 MHz that is an unchanged 128 T-states; above it the rate is capped at roughly 8 kB/s of real time.

> **Note**
>
> That cap is deliberately **slower** than a real UART at 115 200 baud. The reasoning is the same as for the terminal: a program that keeps up in the emulator will keep up on the board. The other way round would mean the bugs only appear on hardware.

The cable is a shared bus

With more than two machines connected, every one hears every byte. A point-to-point link is a convention, not a property of the device.

> **Pitfall**
>
> Time travel stops at the edge of the cable. The FIFOs, the countdown of the byte in flight and the sticky bits are all machine state and replay exactly — but **the peer does not rewind**. What has been sent cannot be recalled, and if a program rewinds past a send and sends again, the peer sees the byte twice.
>
> Bytes arriving while paused or rewound land in the FIFO of the state the machine currently sits at — exactly like a keypress.

CHAPTER 19

Time, the Timer and Interrupts

The machine offers four independent time bases, differing in one essential respect: three of them derive from T-states and therefore speed up when the clock does. Only one is tied to real time.

Address	Register
0xFF30	TMR_DIV
0xFF31	TMR_CNT
0xFF32	CYC0
0xFF33	CYC1
0xFF34	CYC2
0xFF35	CYC3
0xFF36	UPT_LO
0xFF37	UPT_HI
0xFF38	CLK_TPMS_LO
0xFF39	CLK_TPMS_HI
0xFF3A	CLK_MS_LO
0xFF3B	CLK_MS_HI

The timer

`TMR_DIV` sets the divider, `TMR_CNT` the counter. Any write to `TMR_CNT` reloads it. When the counter reaches zero it raises an interrupt (bit 0 of `IRQ_ENABLE`).

```
        LDA #100
        STA TMR_DIV
        STA TMR_CNT
        LDA #0b00000001     ; enable the timer interrupt
        STA IRQ_ENABLE
        EI
```

The counters

All the counters are read-only, run continuously and are cleared only on reset.

Counter	Width	Step
`CYC0` – `CYC3`	32 b	one T-state
`UPT_LO` / `UPT_HI`	16 b	one video frame
`CLK_TPMS_LO` / `HI`	16 b	does not change — it is the clock rate in T-states per millisecond
`CLK_MS_LO` / `HI`	16 b	one real millisecond

Pitfall

Reading the lowest byte **latches a snapshot** of the whole value. A program must therefore read `CYC0` before `CYC1`, `UPT_LO` before `UPT_HI` and `CLK_MS_LO` before `CLK_MS_HI` — otherwise it gets two halves from different moments and nonsense across a 256 boundary.

The difference between the bases shows the moment the clock rate changes:

Base	Behaviour when the clock speeds up
`CYC`	ticks faster — it counts **work**, not time
`UPT`	ticks faster — a frame derives from T-states
`CLK_MS`	ticks the same — it is tied to real time

Hence the division of labour: `CYC` is for **measuring code** (what something cost), `CLK_MS` for **pacing** (how long to wait).

Frame pacing

`VID_STATUS` bit 0 is a frame tick, cleared on read. The period is 1000 T-states.

```
wait_frame:
        LDA VID_STATUS
        AND #0x01
        JZ  wait_frame
        RET
```

That approach is simple but flawed: the frame rate depends on the clock rate. A program that should behave the same in a browser and on the board paces against `CLK_MS` instead:

```
wait_frame:                     ; hold a frame to FRAME_MS ms
.spin:  LDA CLK_MS_LO           ; elapsed = CLK_MS - anchor
        SUB ANCHOR
        STA EL
        LDA CLK_MS_HI
        SBC ANCHOR+1
        JNZ .done               ; over 256 ms - long past due
        LDA EL
        CMP #FRAME_MS           ; C = 1 while it is less
        JC .spin
.done:  LDA CLK_MS_LO           ; anchor only on release
        STA ANCHOR
        LDA CLK_MS_HI
        STA ANCHOR+1
        RET
```

> **Note**
>
> The anchor moves at the moment of release, not to a fixed multiple. With an absolute deadline a missed frame would carry debt, the program would try to catch up and the counter would eventually wrap. This way a late frame is simply late.

Interrupt control

Address	Register
`0xFF40`	`IRQ_ENABLE`
`0xFF41`	`IRQ_STATUS`

`IRQ_ENABLE` enables individual sources:

Bit	Source
0	timer
1	keyboard — an unread character is waiting
2	serial line — the receive FIFO is not empty

Only the low three bits are stored; the rest are masked away.

`IRQ_STATUS` shows what is currently pending; reading it clears it.

Above all of that sits the master switch — the `I` flag, controlled by `EI` and `DI`. For an interrupt to be taken, the source must be enabled **and** `I` set.

> **Pitfall**
>
> The sources differ in how they are cleared. The timer is edge-like: the interrupt happens and that is that. The keyboard and the serial line are **level-sensitive** — they stay asserted until the handler reads `KBD_DATA` or drains the receive FIFO.
>
> A handler that does not touch its source is therefore called again immediately after `RTI`, and the program spins without anything crashing.

The detail of interrupt entry, the stack frame and `RTI` is in the interrupts chapter of Part I; this chapter describes only the device that raises them.

CHAPTER 20

Video I: Text, Glyphs and Colour

Video in SAP-3/16 has no “draw” command. The display simply **reads memory** — and whatever a program writes there appears. There is no difference here between “write a character to the screen” and “write a byte to memory”.

Address	Register
0xFF50	VID_CTRL
0xFF51	VID_STATUS
0xFF52	VID_CURX
0xFF53	VID_CURY
0xFF54	VID_BORDER
0xFF55	VID_SCROLLX
0xFF56	VID_SCROLLY
0xFF57	VID_MCOLOR
0xFF58	SPR_ENABLE

There are only nine registers and none of them carries picture data. That lives in VRAM, which is ordinary RAM: writes to it are journalled, you can put a watchpoint on them and you can rewind past them.

Text mode

The default after reset. The grid is 40 × 25 characters, each 8 × 8 pixels — 320 × 200 dots in total, which is the machine’s native resolution.

The character buffer

It lives at 0x8000–0x83E7 and a cell’s address is computed like this:

```
address = 0x8000 + y × 40 + x
```

Writing a character is therefore one memory write:

```
        LDA #'A'
        STA 0x8000          ; top-left corner
```

Glyphs

The shapes of the characters are not in ROM but in memory at 0x8400–0x8BFF: 256 glyphs of eight bytes, one byte per row, **most significant bit leftmost**.

```
GLYPH_A EQU 0x8608          ; 0x8400 + 'A' * 8, computed ahead
        LDA #0xFF           ; a solid row
        STA GLYPH_A         ; the first of eight rows of 'A'
```

> **Note**
>
> On reset and on program load the glyphs are filled with the built-in font — **unless** the loaded image put a non-zero byte anywhere in the glyph region. A program carrying its own font therefore wins and need not switch anything off.

Colour RAM

At 0x8C00–0x8FE7 sits one attribute byte per cell, laid out exactly like the character buffer. A colour's address is the character's address plus 0x0C00 — which is neat in practice, because a computed screen pointer converts with a single `ADD #0x0C` on the high byte.

Nibble	Meaning
low	foreground colour — an index into the palette
high	background colour — an index into the palette

The value 0x00 is special: it means the **default look** (light on dark), not black on black. That matters because RAM clears to zero on reset — a program that never touches colour therefore looks exactly as it did before.

> **Pitfall**
>
> This exception costs one combination: genuine black on black cannot be written, because 0x00 is taken. For a black rectangle, draw a space on a black background.

The palette

Sixteen colours, shared by colour RAM, the `VID_MCOLOR` register and the border `VID_BORDER`.

Index	Color	Index	Color
0	#000000	8	#8b5429
1	#ffffff	9	#574200
2	#883932	10	#b86962
3	#67b6bd	11	#505050
4	#8b3f96	12	#787878
5	#55a049	13	#94e089
6	#40318d	14	#7869c4
7	#bfce72	15	#9f9f9f

Figure 12. The 16-colour palette, inspired by the Commodore 64.

Cursor and border

`VID_CURX` and `VID_CURY` place the blinking underline. The cursor is **not stored in the buffer** — the display draws it — and it appears only while it is on the grid, that is `VID_CURX` < 40 and `VID_CURY` < 25.

> **Pitfall**
>
> Both registers reset to zero, so a graphics program that never touches them shows a blinking cursor in the top-left cell. You switch it off by parking it off the grid — conventionally `VID_CURY = 25`:
>
> ```
> LDA #25
> STA VID_CURY        ; cursor gone
> ```

`VID_BORDER` picks the border colour from the palette (masked to the low four bits) and works in every mode.

Switching modes

`VID_CTRL` has an enable in bit 0 and the mode in bits 1–2:

Mode	Description
0	text 40 × 25
1	bitmap 160 × 100, 16 colours
2	bitmap 320 × 200, monochrome

Mode	Description
3	reserved — draws a blank raster

The bitmap modes read a different region of VRAM from text, so you can switch back and forth with the text screen left untouched. They are described in the next chapter.

> **Note**
>
> That VRAM is ordinary memory has one unexpectedly useful consequence: the character buffer can be read as a **collision map**. A game need not ask "what is at this coordinate" — it reads a byte off the screen. Colour RAM does not interfere: it lives elsewhere and reading a character is unaffected.

CHAPTER 21

Video II: Bitmaps, Sprites and the Blitter

Text mode is for characters. When something has to be drawn dot by dot, you switch `VID_CTRL` and the same memory starts being read differently.

A shared framebuffer

Both bitmap modes read the same 8000 bytes starting at 0x9000. Switching modes therefore does not **clear** memory, it only starts interpreting it differently.

Mode 1 — 160 × 100 in 16 colours

Two pixels per byte, **the high nibble is the left pixel**. A nibble is a direct palette index.

```
address = 0x9000 + y × 80 + x/2
```

The pixels are square — each covers a 2 × 2 block of the native 320 × 200 raster.

The 0x00 exception does **not** apply here: cleared memory is black, because a nibble selects one colour rather than a pair. The sentinel exists only where one byte picks two colours — in colour RAM and in `VID_MCOLOR`.

Mode 2 — 320 × 200 monochrome

Eight pixels per byte, most significant bit leftmost.

```
address = 0x9000 + y × 40 + x/8
```

A set bit is drawn in the foreground colour, a clear one in the background; both come from `VID_MCOLOR` in the same way as a colour-RAM attribute (low nibble foreground, high nibble background, 0x00 the default look).

Fine scroll

`VID_SCROLLX` and `VID_SCROLLY` shift the picture by 0 to 7 dots (only the low three bits are stored). It is not a move of memory — it is a shift of **where the display reads from**, with wraparound:

output dot (x, y) shows playfield dot ((x + SCROLLX) mod W, (y + SCROLLY) mod H)

Eight steps correspond exactly to one character cell, one byte in mode 2 or two bytes in mode 1. Smooth scrolling is therefore built in two layers: the fine shift in the register, and once it reaches eight, a coarse move of the whole memory with the blitter.

Pitfall

Scroll moves **the playfield only**. Sprites, the cursor and the border do not move — their coordinates are in screen space. That is usually what a game wants (the character stays where it is), but it surprises you the first time.

Sprites

Eight sprites, each 16 × 16 "fat" pixels — one such pixel covers a 32 × 32 block of the native raster. They are drawn above the playfield in all three display modes and below the cursor.

All sprite data is ordinary VRAM. The only register is `SPR_ENABLE`, whose bit **i** turns on sprite **i**.

The attribute table

At 0xAFE0, four bytes per sprite (sprite **i** at `0xAFE0 + i*4`):

Offset	Byte	Meaning
+0	X	position in 160 × 100 space, **offset by +16**
+1	Y	the same vertically
+2	pattern	pattern index, taken modulo 32
+3	flags	bit 0 = flip horizontally, bit 1 = vertically

The +16 offset is the trick that lets one byte cover every position including partial clipping: X = 16 is flush with the left edge, X = 0 is entirely off screen.

Pattern memory

At 0xB000, 32 patterns of 128 bytes (pattern **p** at `0xB000 + p*128`): 16 rows of eight bytes, two 4-bit pixels per byte, high nibble on the left — the same packing as mode 1.

Nibble 0 is transparent; values 1–15 are palette indices.

```
        ; sprite 0 in the middle of the screen
        LDA #96             ; X = 80 + 16 (coordinate offset)
        STA 0xAFE0
```

```
        LDA #66             ; Y = 50 + 16
        STA 0xAFE1
        LDA #0
        STA 0xAFE2          ; pattern 0
        STA 0xAFE3          ; no flipping
        LDA #0b00000001
        STA SPR_ENABLE
```

> **Note**
>
> A sprite is shown only while its bit in `SPR_ENABLE` is set — there is no enable bit in the attributes. Sprite 0 has the highest priority and is painted over sprites 1 to 7.

The blitter

Address	Register
`0xFF70`	`PAD1`
`0xFF71`	`PAD2`
`0xFF72`	`BLT_SRC_LO`
`0xFF73`	`BLT_SRC_HI`
`0xFF74`	`BLT_DST_LO`
`0xFF75`	`BLT_DST_HI`
`0xFF76`	`BLT_LEN_LO`
`0xFF77`	`BLT_LEN_HI`
`0xFF78`	`BLT_FILL`
`0xFF79`	`BLT_CTRL`
`0xFF7A`	`BLT_STATUS`

The blitter is a DMA engine that copies or fills a run of memory by itself. It is the hardware equivalent of `memcpy` and `memset`, which the processor would otherwise pay for with an `LDA` / `STA` loop per byte.

The procedure is always the same: fill in the addresses and the length, then write the command.

Register	Meaning
`BLT_SRC_LO` / `HI`	source address (copy only)

Register	Meaning
`BLT_DST_LO` / `HI`	destination address
`BLT_LEN_LO` / `HI`	length in bytes; zero is a no-op
`BLT_FILL`	the byte to fill with
`BLT_CTRL`	1 = copy, 2 = fill — writing it triggers the operation
`BLT_STATUS`	bit 0 ready (always 1), bit 1 error (cleared on read)

```
        ; clear the text screen with spaces
        LDA #<0x8000
        STA BLT_DST_LO
        LDA #>0x8000
        STA BLT_DST_HI
        LDA #<1000
        STA BLT_LEN_LO
        LDA #>1000
        STA BLT_LEN_HI
        LDA #' '
        STA BLT_FILL
        LDA #2              ; FILL
        STA BLT_CTRL
```

The whole transfer happens inside one instruction

Unlike storage, the blitter **has no busy window**. The entire block moves within the write to `BLT_CTRL` and costs not one extra T-state. There is therefore nothing to poll — the ready bit is a constant one.

Copying behaves like memmove

The engine picks the copy direction from the addresses: descending when the destination is above the source, ascending otherwise. An **overlapping** copy is therefore always correct — which is exactly what a hardware scroll needs, copying the screen onto itself shifted by a row. No direction flag is set.

A fill always ascends.

Pitfall

The transfer window must stay entirely below the peripheral block. When `dst + len` (and for a copy `src + len` as well) reaches 0xFF00, the engine sets the error bit and **transfers nothing**.

This is not pedantry: that single condition keeps DMA out of the peripherals, the boot ROM and the vectors, and simultaneously makes a wrap past the end of memory impossible.

Note

Every byte the blitter writes goes into the journal by the same path as a processor store. Time travel therefore replays a blit exactly — one large blit can only shorten the rewind horizon, because it consumes many journal entries.

CHAPTER 22

Audio

Three independent voices, each with its own frequency, waveform and envelope. The audio device is **write and forget** — the processor sets its registers and takes no further part; the envelope generator and the mixing live in the host and nothing comes back to the processor from them.

Address	Register
`0xFF20`	`AUD_CTRL`
`0xFF21`	`AUD_FREQ_LO`
`0xFF22`	`AUD_FREQ_HI`
`0xFF23`	`AUD_WAVE`
`0xFF24`	`AUD_AD`
`0xFF25`	`AUD_SR`

Channel layout

Three channels of five registers fill the device's slot exactly. Channel **n** starts at `0xFF21 + n*5`:

Channel	Range	Registers
0	`0xFF21–0xFF25`	`FREQ_LO FREQ_HI WAVE AD SR`
1	`0xFF26–0xFF2A`	`FREQ_LO FREQ_HI WAVE AD SR`
2	`0xFF2B–0xFF2F`	`FREQ_LO FREQ_HI WAVE AD SR`

Figure 13. The three audio channels and their registers.

Offset	Register	Meaning
+0 / +1	`AUD_FREQ_LO` / `HI`	frequency, 16 bits, low byte first
+2	`AUD_WAVE`	waveform
+3	`AUD_AD`	high nibble attack, low nibble decay
+4	`AUD_SR`	high nibble sustain level, low nibble release

> **Note**

The five-byte stride is chosen so that any channel is reachable with a single indexed instruction:

```
        LDX #5              ; 0, 5 or 10 = channel 0, 1, 2
        STA AUD_FREQ_LO,X
```

Frequency is in hertz

The register pair **is** the pitch: `f = {AUD_FREQ_HI, AUD_FREQ_LO}` hertz, at 1 Hz resolution. Zero means silence.

```
        LDA #<440
        STA AUD_FREQ_LO
        LDA #>440
        STA AUD_FREQ_HI     ; concert A
```

Pitfall

In the older version of the audio device there was a single channel and the frequency followed the formula `100 + (FREQ_LO/255) × 1900`, with the high byte dead. A program written against the old curve still assembles and runs — it just plays differently: `AUD_FREQ_LO = 46` used to mean about 443 Hz and now means 46 Hz.

The fix is always to write the frequency as a 16-bit value.

Waveforms

`AUD_WAVE` value	Waveform
0	sine
1	square
2	triangle
3	sawtooth
4	noise
5–7	reserved — they play silence

Noise replaces the oscillator with a shift register clocked at the channel's frequency, so `AUD_FREQ` selects the **colour** of the noise rather than a pitch: low values rumble, high ones hiss.

The gate and its edge

`AUD_CTRL` is a bitmap of gates — bit **n** gates channel **n** (bits 3–7 read zero). The gate is **edge-triggered**:

Transition	What happens
0 ⟶ 1	the attack begins — the note sounds
1 ⟶ 0	the release begins — the note dies away

> **Pitfall**
>
> Writing a bit that is already set does **not** retrigger the note. A tracker wanting to repeat the same note must clear the bit and set it again.
>
> Conversely, one write to `AUD_CTRL` can start or stop all three voices on the same edge — which is exactly what a chord needs.

The envelope

Each channel has its own ADSR envelope, read at every gate edge.

The time nibbles (attack, decay, release) convert like this: zero means instant, otherwise

$$t = 8\ \text{ms} \times 2^{(n-1)/2}$$

So 8 ms at 1, doubling every two steps, 1024 ms at 15.

The sustain nibble is the level `s/15` the note holds at until the gate closes. It is also the only way to set a channel's **volume** — there is no volume register. A sustain of zero makes a percussive sound that dies by itself.

```
        ; a click: instant attack, fast decay, no sustain
        LDA #0x03           ; attack 0, decay 3
        STA AUD_AD
        LDA #0x02           ; sustain 0, release 2
        STA AUD_SR
```

Defaults

After reset each channel has its gate closed, a frequency of 440 Hz, a sine wave, `AUD_AD` = 0x00 and `AUD_SR` = 0xF0 — instant attack, full sustain, instant release. The envelope is therefore a plain on/off switch until a program shapes it.

The three channels mix at equal weight with headroom for all three at full level.

> **Note**
>
> The registers are ordinary machine state and are journalled — time travel replays the sound exactly as it was. The envelope and the mixing live in the host, though, and nothing comes back from them, so a program has no way to ask whether a note is still sounding.

CHAPTER 23

Storage, the Boot Sector and SAP-DOS

Storage is the one device you cannot talk to instantly. A transfer has **duration** — and that is the most important thing a program has to know about it.

Address	Register
`0xFF60`	`DSK_CMD`
`0xFF61`	`DSK_STATUS`
`0xFF62`	`DSK_BLK`
`0xFF63`	`DSK_ADDR_LO`
`0xFF64`	`DSK_ADDR_HI`
`0xFF65`	`SER_DATA`
`0xFF66`	`SER_STATUS`
`0xFF67`	`SER_CTRL`
`0xFF68`	`DSK_DISK`

The media

The media is a **card of up to 256 disks**, not one large drive. Each disk has 256 blocks of 256 bytes, that is 64 KB.

`DSK_DISK` selects which disk is "in the drive" for the next command. It reads back the last value written and resets to zero — so a program that never touches it lives its whole life on disk 0.

How many slots actually hold media is a property of the inserted card; the emulator offers 16. A command aimed at a slot the card does not have is an error.

> **Note**
>
> The device knows nothing about formats. Whether a disk holds a directory, a boot sector or nothing is a matter for software — and each slot may be laid out differently.

Commands

The procedure is always the same: set the disk, block and memory address, then write the command.

Register	Meaning
`DSK_DISK`	the slot number on the card
`DSK_BLK`	the block number on the disk
`DSK_ADDR_LO` / `HI`	the address in RAM
`DSK_CMD`	1 = read a block into RAM, 2 = write a block from RAM
`DSK_STATUS`	bit 0 ready, bit 1 error (cleared on read)

Transfers take time

A valid write to `DSK_CMD` latches the **current** values of the other registers and starts a busy window of `max(2, CLK_TPMS × 2)` T-states — about 2 ms of real time.

> **Pitfall**
>
> Never count cycles; always ask. Real SD cards need anywhere from hundreds of microseconds to several milliseconds per block, and that figure is not guaranteed.
>
> Data from a read is **not in memory** until `DSK_STATUS` bit 0 returns to one.

Rewriting `DSK_DISK`, `DSK_BLK` or `DSK_ADDR` while busy is legal and affects only the **next** command.

Polling with a single read

Because reading `DSK_STATUS` clears the error bit, the status must be read once and both bits tested from that same value:

```
        STA DSK_CMD
.wait:  LDA DSK_STATUS
        STA tmp             ; one read, both bits from it
        AND #0x01
        JZ  .wait           ; still working
        LDA tmp
        AND #0x02
        JNZ error           ; the command was rejected
```

Errors happen at command time

An error is set **immediately** and no transfer starts at all. It happens in four cases:

1. an invalid command value,
2. a RAM address above 0xFE00,
3. a `DSK_DISK` slot beyond the inserted media,
4. a write to `DSK_CMD` while a transfer is in flight (the one in flight completes).

The address ceiling is why DMA can never reach the peripherals, the system RAM or the vectors.

> **Note**
>
> A write samples memory at an **unspecified point** inside the busy window — the emulator at its end, hardware at command time. A program that changes its source buffer while waiting gets implementation-defined media contents. So it simply is not done.

Time travel and the media

Bytes DMA writes into RAM are journalled one by one, so rewinding replays them exactly and without re-reading the disk.

The media itself does not rewind. Stepping back past a disk write does not unwrite it — as with real hardware. And if a program rewinds past a read and performs it again, it reads the media as it is **now**.

Booting from disk

Normally the host places the assembled program straight into RAM and the RESET vector points at it. **Boot mode** instead makes the machine start up like a real computer: a small boot ROM is installed at 0xFF80, the RESET vector points at it, and RAM is empty.

The boot ROM is **real SAP-3 machine code**, so the whole startup can be single-stepped in the debugger.

What it does:

1. DMAs block 0 to the scratch page 0xFE00,
2. checks the two-byte signature `"S3"` — a mismatch prints `NO OS` and halts,
3. reads the header,
4. DMAs blocks 1..N into RAM at the stated address,
5. jumps to the entry point via `JMP (zp)`.

> **Note**

The ROM lives in the window 0xFF80–0xFFF9, which DMA cannot reach. The load it triggers therefore cannot overwrite it. It keeps its working bytes in the top of the zero page, so payloads load from 0x0200 upwards.

The boot sector

Block 0 of a bootable disk, low byte first:

Offset	Length	Meaning
0x00	2	the signature `"S3"`
0x02	2	load address, block-aligned
0x04	1	number of payload blocks
0x05	2	entry point
0x07	1	data disk to mount; 0 = keep the booted one
0x08	16	program name, ASCII, zero-padded

The payload lives in blocks 1 to N.

The last two fields are **ignored** by the boot ROM — only a second-stage loader, the SAP-DOS menu, reads them. It builds its program list from the names and mounts the right slot from the data-disk field before jumping.

Note

A boot disk and a Tiny BASIC data disk differ only in the format of block 0. The device does not distinguish them; software does, by the signature.

Pitfall

Neither reset nor loading a program **touches the media** — only the registers reset and `DSK_DISK` returns to zero. It is as though nothing were ejected. A program assuming the disk will be “clean” after a reset is wrong.

CHAPTER 24

From Emulator to Silicon

This chapter is an overview. It is not here so that anyone can build a board from it — the design documents and schematics are for that. It answers a different question: **what it means that this processor exists in three implementations, and why they have not drifted apart.**

Three machines, one specification

Implementation	What it is
the emulator	TypeScript in a browser at sap-3.com — T-state stepping, time travel, visible control signals
AGATA-1	a Verilog core on a Sipeed Tang Nano 20K FPGA; the heart of the MAXIK console
the ESP32 port	the same processor in C++ on a microcontroller with a small display

If these were three independent implementations of one description they would diverge within weeks. They are not — and that rests on three mechanisms.

The microcode has a single source

The microcode is not written in HDL. It is **generated** from the same source the emulator uses and loaded into a ROM. Whoever wants to change the microcode changes it in one place; the hardware gets it at the next build.

Related to that is one unusual property of the control-signal taxonomy from Chapter 4: the order of the signals within a group **is part of the microword encoding**. A signal may be appended but not reordered — otherwise the ROM would change without the microcode changing.

The ALU is verified exhaustively

An 8-bit ALU has finitely many inputs, so it can be tested **completely**. A set of vectors — every combination of operands for every operation — is generated from the source, and the HDL simulation has to match them to the bit, flags included.

Programs are replayed against reference traces

The emulator can produce a trace: for every T-state, the state of the registers, buses and signals. The hardware simulation then runs the same program and must pass through the same sequence. Not the same result — the same sequence.

> **Note**
>
> This is why the whole book can claim that its T-state counts hold on the board as well. “The browser behaves like the board” is not a promise in a document; it is a condition the build has to satisfy.

What determinism costs

The price is concrete and was already visible in the performance chapter: the classic speed-up tricks of modern processors — caches, branch prediction, out-of-order execution — are ruled out **in principle**. All of them make running time depend on history, and that would break trace comparison, stepping and time travel.

What is allowed is deterministic speed-up only: fixed shorter T-state counts, a wider instruction fetch, possibly a pipeline with a fixed schedule.

What it buys: a program can be stepped gate by gate, rewound, and a cycle count measured in the browser holds on the board. For a machine whose main job is to teach how a computer works, that is a good trade.

AGATA-1 on the Tang Nano 20K

The core occupies roughly 700 logic cells, the whole system-on-chip about 8 % of the available logic. The bottleneck is not logic but block memory: 45 of 46 blocks are in use.

Resource	State
block memory	45 of 46 blocks — one free
logic	roughly 8 % occupied
maximum frequency	about 32 MHz after place and route

The critical path runs from memory through the bus multiplexer and the ALU into the flag chain, all within one clock.

> **Note**
>
> That single free memory block is the hardest constraint in the whole design. Anything wanting new memory — a cache, a wider ROM, buffers — has nowhere to live on this board. Logic, by contrast, is nearly free.

Turbo

In turbo mode the core runs at 25.2 MHz with one T-state per clock. With an average instruction at 7 to 9 T-states that is roughly 3 million instructions per second.

25 200 is precisely the ceiling the specification allows for the `CLK_TPMS` register — one T-state per pixel-clock cycle. The register's sixteen bits leave room up to somewhere near 65 MHz for future implementations.

The MAXIK console

The board around the core adds what turns a processor into a computer: HDMI output, an SD card holding a multi-disk card, two NES controller ports and a serial line.

A chapter of its own is the front panel — a board covered in LEDs and seven-segment displays that shows the contents of the buses, registers and flags in real time. The colour language is uniform: amber for addresses, green for data, red for flags.

The panel is not decoration. It is the physical realisation of the same schematic the emulator draws — and the only way to see what the processor is doing without looking into a computer.

What follows for a program

Practically nothing — and that is the point. A program written according to this book runs identically on all three machines. The one thing to watch is the **clock rate**, because it differs by an order of magnitude:

Machine	Typical speed
emulator, teaching mode	fractions of a Hz to kilohertz
emulator, normal use	hundreds of kHz to 2 MHz
AGATA-1, turbo	25.2 MHz

A program that reads the rate from `CLK_TPMS`, or paces against `CLK_MS`, runs at the same speed everywhere. A program with a baked-in constant will run twelve times faster on the board than it expected.

> **Pitfall**
>
> The most common fault when moving a program from the browser to the board is not a logic error but a delay loop counted in iterations. At 25.2 MHz it finishes practically instantly.

Appendices

APPENDIX A

Opcode Matrix

The whole 8-bit opcode space in one table. The row is the high nibble, the column the low nibble, so cell `4` × `1` is opcode `0x41`. A dot marks a free slot — such a byte traps as an illegal opcode.

	_0	_1	_2	_3	_4	_5	_6	_7	_8	_9	_A	_B	_C	_D	_E	_F
0_	·	NOP	HLT	OUT	CLC	SEC	EI	DI	INC	DEC	SHL	SHR	ROL	ROR	NOT	RND
1_	INX	DEX	INY	DEY	·	·	·	·	TAX	TXA	TAY	TYA	TAB	TBA	TXYS	TSXY
2_	PUSH	POP	PHX	PLX	PHY	PLY	PHF	PLF	MUL	DIV	PHB	PLB	·	·	·	·
3_	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·
4_	LDA #n	LDA zp	LDA abs	LDA abs,X	LDA abs,Y	LDA (zp)	LDA zp,X	LDA (zp),Y	LDX #n	LDX zp	LDX abs	·	LDX abs,Y	·	·	·
5_	LDY #n	LDY zp	LDY abs	LDY abs,X	·	·	LDY zp,X	·	CPX #n	CPX zp	CPX abs	·	·	·	·	·
6_	·	STA zp	STA abs	STA abs,X	STA abs,Y	STA (zp)	STA zp,X	STA (zp),Y	·	STX zp	STX abs	·	·	·	·	·
7_	·	STY zp	STY abs	·	·	·	·	·	CPY #n	CPY zp	CPY abs	·	·	·	·	·
8_	ADD #n	ADD zp	ADD abs	ADD abs,X	ADD abs,Y	ADD (zp)	ADD zp,X	ADD (zp),Y	ADC #n	ADC zp	ADC abs	ADC abs,X	ADC abs,Y	ADC (zp)	ADC zp,X	ADC (zp),Y
9_	SUB #n	SUB zp	SUB abs	SUB abs,X	SUB abs,Y	SUB (zp)	SUB zp,X	SUB (zp),Y	SBC #n	SBC zp	SBC abs	SBC abs,X	SBC abs,Y	SBC (zp)	SBC zp,X	SBC (zp),Y
A_	AND #n	AND zp	AND abs	AND abs,X	AND abs,Y	AND (zp)	AND zp,X	AND (zp),Y	OR #n	OR zp	OR abs	OR abs,X	OR abs,Y	OR (zp)	OR zp,X	OR (zp),Y
B_	XOR #n	XOR zp	XOR abs	XOR abs,X	XOR abs,Y	XOR (zp)	XOR zp,X	XOR (zp),Y	CMP #n	CMP zp	CMP abs	CMP abs,X	CMP abs,Y	CMP (zp)	CMP zp,X	CMP (zp),Y
C_	JNZ abs	JZ abs	JNC abs	JC abs	JP abs	JN abs	JNV abs	JV abs	JMP abs	JMP (zp)	CALL abs	RET	RTI	BRK	CALL (zp)	·
D_	INC zp	INC abs	DEC zp	DEC abs	INW zp	INW abs	DEW zp	DEW abs	SHL zp	SHL abs	SHR zp	SHR abs	ROL zp	ROL abs	ROR zp	ROR abs
E_	JGT abs	JLE abs	JGES abs	JLTS abs	BIT #n	BIT zp	BIT abs	·	ADW zp	SBW zp	LXY #n	SXY zp	·	·	·	·
F_	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·

Of the 256 possible bytes, 179 are legal instructions, 2 are reserved pseudo-opcodes injected by hardware (`0xFD RESET`, `0xFE IRQ`), and 75 slots remain free.

> **Note**
>
> The layout is not arbitrary. The horizontal bands are the `gg` groups: rows 0–3 are implied instructions, 4–7 moves between registers and memory, 8–B arithmetic and logic on **A**, C–F control flow and read-modify-write operations. Within groups 01 and 10 the column is the addressing mode, which is why the same mnemonics run in runs of eight.

APPENDIX B

Instructions by Group

Every one of the 179 legal opcodes, in byte order and split by the `gg` group field. The Flags column lists only the flags a form **defines**; anything absent keeps its previous value.

Group 00 — implied instructions

One byte, no operand. Range 0x00–0x3F.

Opcode	Instruction	Syntax	Bytes	T	Flags
`0x01`	`NOP`	—	1	3	*none*
`0x02`	`HLT`	—	1	3	*none*
`0x03`	`OUT`	—	1	3	*none*
`0x04`	`CLC`	—	1	3	`C`
`0x05`	`SEC`	—	1	3	`C`
`0x06`	`EI`	—	1	3	`I`
`0x07`	`DI`	—	1	3	`I`
`0x08`	`INC`	—	1	3	`Z N`
`0x09`	`DEC`	—	1	3	`Z N`
`0x0A`	`SHL`	—	1	3	`C Z N`
`0x0B`	`SHR`	—	1	3	`C Z N`
`0x0C`	`ROL`	—	1	3	`C Z N`
`0x0D`	`ROR`	—	1	3	`C Z N`
`0x0E`	`NOT`	—	1	3	`Z N`
`0x0F`	`RND`	—	1	3	*none*
`0x10`	`INX`	—	1	4	`Z N`
`0x11`	`DEX`	—	1	4	`Z N`
`0x12`	`INY`	—	1	4	`Z N`
`0x13`	`DEY`	—	1	4	`Z N`
`0x18`	`TAX`	—	1	3	`Z N`
`0x19`	`TXA`	—	1	3	`Z N`

Opcode	Instruction	Syntax	Bytes	T	Flags
0x1A	TAY	—	1	3	Z N
0x1B	TYA	—	1	3	Z N
0x1C	TAB	—	1	3	Z N
0x1D	TBA	—	1	3	Z N
0x1E	TXYS	—	1	4	*none*
0x1F	TSXY	—	1	4	*none*
0x20	PUSH	—	1	4	*none*
0x21	POP	—	1	4	Z N
0x22	PHX	—	1	4	*none*
0x23	PLX	—	1	4	Z N
0x24	PHY	—	1	4	*none*
0x25	PLY	—	1	4	Z N
0x26	PHF	—	1	4	*none*
0x27	PLF	—	1	4	C Z N V I
0x28	MUL	—	1	4	C Z N
0x29	DIV	—	1	4	C Z N
0x2A	PHB	—	1	4	*none*
0x2B	PLB	—	1	4	Z N

Group 01 — moves between registers and memory

Range 0x40–0x7F. The opcode is `01 | ooo | mmm`, so the same mnemonics run in contiguous groups of eight.

Opcode	Instruction	Syntax	Bytes	T	Flags
0x40	LDA	#n	2	5	Z N
0x41	LDA	addr8	2	6	Z N
0x42	LDA	addr16	3	9	Z N
0x43	LDA	addr16,X	3	11	Z N
0x44	LDA	addr16,Y	3	11	Z N
0x45	LDA	(addr8)	2	10	Z N
0x46	LDA	addr8,X	2	7	Z N
0x47	LDA	(addr8),Y	2	13	Z N

Opcode	Instruction	Syntax	Bytes	T	Flags
`0x48`	`LDX`	`#n`	2	5	`Z N`
`0x49`	`LDX`	`addr8`	2	6	`Z N`
`0x4A`	`LDX`	`addr16`	3	9	`Z N`
`0x4C`	`LDX`	`addr16,Y`	3	11	`Z N`
`0x50`	`LDY`	`#n`	2	5	`Z N`
`0x51`	`LDY`	`addr8`	2	6	`Z N`
`0x52`	`LDY`	`addr16`	3	9	`Z N`
`0x53`	`LDY`	`addr16,X`	3	11	`Z N`
`0x56`	`LDY`	`addr8,X`	2	7	`Z N`
`0x58`	`CPX`	`#n`	2	7	`C Z N`
`0x59`	`CPX`	`addr8`	2	8	`C Z N`
`0x5A`	`CPX`	`addr16`	3	11	`C Z N`
`0x61`	`STA`	`addr8`	2	6	*none*
`0x62`	`STA`	`addr16`	3	9	*none*
`0x63`	`STA`	`addr16,X`	3	11	*none*
`0x64`	`STA`	`addr16,Y`	3	11	*none*
`0x65`	`STA`	`(addr8)`	2	10	*none*
`0x66`	`STA`	`addr8,X`	2	7	*none*
`0x67`	`STA`	`(addr8),Y`	2	13	*none*
`0x69`	`STX`	`addr8`	2	6	*none*
`0x6A`	`STX`	`addr16`	3	9	*none*
`0x71`	`STY`	`addr8`	2	6	*none*
`0x72`	`STY`	`addr16`	3	9	*none*
`0x78`	`CPY`	`#n`	2	7	`C Z N`
`0x79`	`CPY`	`addr8`	2	8	`C Z N`
`0x7A`	`CPY`	`addr16`	3	11	`C Z N`

Group 10 — arithmetic and logic on A

Range 0x80–0xBF, encoded `10 | ooo | mmm`.

Opcode	Instruction	Syntax	Bytes	T	Flags
`0x80`	`ADD`	`#n`	2	6	`C Z N V`

Opcode	Instruction	Syntax	Bytes	T	Flags
0x81	ADD	addr8	2	7	C Z N V
0x82	ADD	addr16	3	10	C Z N V
0x83	ADD	addr16,X	3	12	C Z N V
0x84	ADD	addr16,Y	3	12	C Z N V
0x85	ADD	(addr8)	2	11	C Z N V
0x86	ADD	addr8,X	2	8	C Z N V
0x87	ADD	(addr8),Y	2	14	C Z N V
0x88	ADC	#n	2	6	C Z N V
0x89	ADC	addr8	2	7	C Z N V
0x8A	ADC	addr16	3	10	C Z N V
0x8B	ADC	addr16,X	3	12	C Z N V
0x8C	ADC	addr16,Y	3	12	C Z N V
0x8D	ADC	(addr8)	2	11	C Z N V
0x8E	ADC	addr8,X	2	8	C Z N V
0x8F	ADC	(addr8),Y	2	14	C Z N V
0x90	SUB	#n	2	6	C Z N V
0x91	SUB	addr8	2	7	C Z N V
0x92	SUB	addr16	3	10	C Z N V
0x93	SUB	addr16,X	3	12	C Z N V
0x94	SUB	addr16,Y	3	12	C Z N V
0x95	SUB	(addr8)	2	11	C Z N V
0x96	SUB	addr8,X	2	8	C Z N V
0x97	SUB	(addr8),Y	2	14	C Z N V
0x98	SBC	#n	2	6	C Z N V
0x99	SBC	addr8	2	7	C Z N V
0x9A	SBC	addr16	3	10	C Z N V
0x9B	SBC	addr16,X	3	12	C Z N V
0x9C	SBC	addr16,Y	3	12	C Z N V
0x9D	SBC	(addr8)	2	11	C Z N V
0x9E	SBC	addr8,X	2	8	C Z N V
0x9F	SBC	(addr8),Y	2	14	C Z N V

Opcode	Instruction	Syntax	Bytes	T	Flags
0xA0	AND	#n	2	6	Z N
0xA1	AND	addr8	2	7	Z N
0xA2	AND	addr16	3	10	Z N
0xA3	AND	addr16,X	3	12	Z N
0xA4	AND	addr16,Y	3	12	Z N
0xA5	AND	(addr8)	2	11	Z N
0xA6	AND	addr8,X	2	8	Z N
0xA7	AND	(addr8),Y	2	14	Z N
0xA8	OR	#n	2	6	Z N
0xA9	OR	addr8	2	7	Z N
0xAA	OR	addr16	3	10	Z N
0xAB	OR	addr16,X	3	12	Z N
0xAC	OR	addr16,Y	3	12	Z N
0xAD	OR	(addr8)	2	11	Z N
0xAE	OR	addr8,X	2	8	Z N
0xAF	OR	(addr8),Y	2	14	Z N
0xB0	XOR	#n	2	6	Z N
0xB1	XOR	addr8	2	7	Z N
0xB2	XOR	addr16	3	10	Z N
0xB3	XOR	addr16,X	3	12	Z N
0xB4	XOR	addr16,Y	3	12	Z N
0xB5	XOR	(addr8)	2	11	Z N
0xB6	XOR	addr8,X	2	8	Z N
0xB7	XOR	(addr8),Y	2	14	Z N
0xB8	CMP	#n	2	6	C Z N
0xB9	CMP	addr8	2	7	C Z N
0xBA	CMP	addr16	3	10	C Z N
0xBB	CMP	addr16,X	3	12	C Z N
0xBC	CMP	addr16,Y	3	12	C Z N
0xBD	CMP	(addr8)	2	11	C Z N
0xBE	CMP	addr8,X	2	8	C Z N

Opcode	Instruction	Syntax	Bytes	T	Flags
`0xBF`	`CMP`	`(addr8),Y`	2	14	`C Z N`

Group 11 — control flow and read-modify-write

Range 0xC0–0xFF. The one group that is not systematic: its instructions either take no operand or take an unusual one.

Opcode	Instruction	Syntax	Bytes	T	Flags
`0xC0`	`JNZ`	`addr16`	3	7	*none*
`0xC1`	`JZ`	`addr16`	3	7	*none*
`0xC2`	`JNC`	`addr16`	3	7	*none*
`0xC3`	`JC`	`addr16`	3	7	*none*
`0xC4`	`JP`	`addr16`	3	7	*none*
`0xC5`	`JN`	`addr16`	3	7	*none*
`0xC6`	`JNV`	`addr16`	3	7	*none*
`0xC7`	`JV`	`addr16`	3	7	*none*
`0xC8`	`JMP`	`addr16`	3	7	*none*
`0xC9`	`JMP`	`(addr8)`	2	9	*none*
`0xCA`	`CALL`	`addr16`	3	11	*none*
`0xCB`	`RET`	—	1	6	*none*
`0xCC`	`RTI`	—	1	8	`C Z N V I`
`0xCD`	`BRK`	—	1	12	`I`
`0xCE`	`CALL`	`(addr8)`	2	13	*none*
`0xD0`	`INC`	`addr8`	2	7	`Z N`
`0xD1`	`INC`	`addr16`	3	10	`Z N`
`0xD2`	`DEC`	`addr8`	2	7	`Z N`
`0xD3`	`DEC`	`addr16`	3	10	`Z N`
`0xD4`	`INW`	`addr8`	2	10	*none*
`0xD5`	`INW`	`addr16`	3	13	*none*
`0xD6`	`DEW`	`addr8`	2	10	*none*
`0xD7`	`DEW`	`addr16`	3	13	*none*
`0xD8`	`SHL`	`addr8`	2	7	`C Z N`
`0xD9`	`SHL`	`addr16`	3	10	`C Z N`

Opcode	Instruction	Syntax	Bytes	T	Flags
`0xDA`	`SHR`	`addr8`	2	7	`C Z N`
`0xDB`	`SHR`	`addr16`	3	10	`C Z N`
`0xDC`	`ROL`	`addr8`	2	7	`C Z N`
`0xDD`	`ROL`	`addr16`	3	10	`C Z N`
`0xDE`	`ROR`	`addr8`	2	7	`C Z N`
`0xDF`	`ROR`	`addr16`	3	10	`C Z N`
`0xE0`	`JGT`	`addr16`	3	7	*none*
`0xE1`	`JLE`	`addr16`	3	7	*none*
`0xE2`	`JGES`	`addr16`	3	7	*none*
`0xE3`	`JLTS`	`addr16`	3	7	*none*
`0xE4`	`BIT`	`#n`	2	6	`Z N`
`0xE5`	`BIT`	`addr8`	2	7	`Z N`
`0xE6`	`BIT`	`addr16`	3	10	`Z N`
`0xE8`	`ADW`	`addr8,#n`	3	13	*none*
`0xE9`	`SBW`	`addr8,#n`	3	13	*none*
`0xEA`	`LXY`	`#nn`	3	8	*none*
`0xEB`	`SXY`	`addr8`	2	8	*none*

APPENDIX C

Microcode Listing

The complete contents of the microcode memory: 181 microprograms in opcode order. One pair of brackets is one T-state and the signals inside it are the ones asserted then. A question mark after a bracket marks a step the flags gate.

The first two to six steps of every program are the fetch and look identical across instructions. The signals are explained in Appendix D.

> **Note**
>
> The last two entries (`0xFD` and `0xFE`) are pseudo-opcodes the hardware injects on reset and on taking an interrupt. They cannot be fetched from memory, which is why they have no fetch prefix.

```
0x01  NOP  (3 T)
[PCM] [RO,II,CE] []

0x02  HLT  (3 T)
[PCM] [RO,II,CE] [HLT]

0x03  OUT  (3 T)
[PCM] [RO,II,CE] [AO,OI]

0x04  CLC  (3 T)
[PCM] [RO,II,CE] [CFC]

0x05  SEC  (3 T)
[PCM] [RO,II,CE] [CFS]

0x06  EI  (3 T)
[PCM] [RO,II,CE] [EIF]

0x07  DI  (3 T)
[PCM] [RO,II,CE] [DIF]

0x08  INC  (3 T)
[PCM] [RO,II,CE] [INC,EO,AI,FI]

0x09  DEC  (3 T)
[PCM] [RO,II,CE] [DEC,EO,AI,FI]

0x0A  SHL  (3 T)
[PCM] [RO,II,CE] [SHL,EO,AI,FI]

0x0B  SHR  (3 T)
[PCM] [RO,II,CE] [SHR,EO,AI,FI]

0x0C  ROL  (3 T)
[PCM] [RO,II,CE] [ROL,EO,AI,FI]

0x0D  ROR  (3 T)
[PCM] [RO,II,CE] [ROR,EO,AI,FI]

0x0E  NOT  (3 T)
[PCM] [RO,II,CE] [NOT,EO,AI,FI]

0x0F  RND  (3 T)
```

```
[PCM] [RO,II,CE] [RND]
```

0x10 INX (4 T)
```
[PCM] [RO,II,CE] [XO,TI] [ATM,INC,EO,XI,FI]
```

0x11 DEX (4 T)
```
[PCM] [RO,II,CE] [XO,TI] [ATM,DEC,EO,XI,FI]
```

0x12 INY (4 T)
```
[PCM] [RO,II,CE] [YO,TI] [ATM,INC,EO,YI,FI]
```

0x13 DEY (4 T)
```
[PCM] [RO,II,CE] [YO,TI] [ATM,DEC,EO,YI,FI]
```

0x18 TAX (3 T)
```
[PCM] [RO,II,CE] [AO,XI,FI]
```

0x19 TXA (3 T)
```
[PCM] [RO,II,CE] [XO,AI,FI]
```

0x1A TAY (3 T)
```
[PCM] [RO,II,CE] [AO,YI,FI]
```

0x1B TYA (3 T)
```
[PCM] [RO,II,CE] [YO,AI,FI]
```

0x1C TAB (3 T)
```
[PCM] [RO,II,CE] [AO,BI,FI]
```

0x1D TBA (3 T)
```
[PCM] [RO,II,CE] [BO,AI,FI]
```

0x1E TXYS (4 T)
```
[PCM] [RO,II,CE] [XO,SHI] [YO,SLI]
```

0x1F TSXY (4 T)
```
[PCM] [RO,II,CE] [SHO,XI] [SLO,YI]
```

0x20 PUSH (4 T)
```
[PCM] [RO,II,CE] [SD,SPM] [AO,RI]
```

0x21 POP (4 T)
```
[PCM] [RO,II,CE] [SPM] [RO,AI,FI,SE]
```

0x22 PHX (4 T)
```
[PCM] [RO,II,CE] [SD,SPM] [XO,RI]
```

0x23 PLX (4 T)
```
[PCM] [RO,II,CE] [SPM] [RO,XI,FI,SE]
```

0x24 PHY (4 T)
```
[PCM] [RO,II,CE] [SD,SPM] [YO,RI]
```

0x25 PLY (4 T)
```
[PCM] [RO,II,CE] [SPM] [RO,YI,FI,SE]
```

0x26 PHF (4 T)
```
[PCM] [RO,II,CE] [SD,SPM] [FLO,RI]
```

0x27 PLF (4 T)
```
[PCM] [RO,II,CE] [SPM] [RO,FLI,SE]
```

0x28 MUL (4 T)
```
[PCM] [RO,II,CE] [MUL,EO,AI,FI] [EXO,BI]
```

0x29 DIV (4 T)
```
[PCM] [RO,II,CE] [DIV,EO,AI,FI] [EXO,BI]
```

0x2A PHB (4 T)
```
[PCM] [RO,II,CE] [SD,SPM] [BO,RI]
```

0x2B PLB (4 T)
```
[PCM] [RO,II,CE] [SPM] [RO,BI,FI,SE]
```

0x40 LDA #n (5 T)
```
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [ORO,AI,FI]
```

0x41 LDA addr8 (6 T)

```
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [ORO,MI] [RO,AI,FI]
```

0x42 LDA addr16 (9 T)
```
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [PCM] [RO,OHI,CE] [ORO,MI] [OHO,MIH] [RO,AI,FI]
```

0x43 LDA addr16,X (11 T)
```
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [PCM] [RO,OHI,CE] [XO,TI] [ATM,AOP,EO,MI,ACO]
[OHO,TI] [ATM,AHC,EO,MIH] [RO,AI,FI]
```

0x44 LDA addr16,Y (11 T)
```
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [PCM] [RO,OHI,CE] [YO,TI] [ATM,AOP,EO,MI,ACO]
[OHO,TI] [ATM,AHC,EO,MIH] [RO,AI,FI]
```

0x45 LDA (addr8) (10 T)
```
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [ORO,MI] [RO,TI] [MRI] [RO,MIH] [TO,MIL] [RO,AI,FI]
```

0x46 LDA addr8,X (7 T)
```
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [XO,TI] [ATM,AOP,EO,MI] [RO,AI,FI]
```

0x47 LDA (addr8),Y (13 T)
```
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [ORO,MI] [RO,ORI] [YO,TI] [MRI] [RO,OHI]
[ATM,AOP,EO,MI,ACO] [OHO,TI] [ATM,AHC,EO,MIH] [RO,AI,FI]
```

0x48 LDX #n (5 T)
```
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [ORO,XI,FI]
```

0x49 LDX addr8 (6 T)
```
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [ORO,MI] [RO,XI,FI]
```

0x4A LDX addr16 (9 T)
```
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [PCM] [RO,OHI,CE] [ORO,MI] [OHO,MIH] [RO,XI,FI]
```

0x4C LDX addr16,Y (11 T)
```
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [PCM] [RO,OHI,CE] [YO,TI] [ATM,AOP,EO,MI,ACO]
[OHO,TI] [ATM,AHC,EO,MIH] [RO,XI,FI]
```

0x50 LDY #n (5 T)
```
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [ORO,YI,FI]
```

0x51 LDY addr8 (6 T)
```
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [ORO,MI] [RO,YI,FI]
```

0x52 LDY addr16 (9 T)
```
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [PCM] [RO,OHI,CE] [ORO,MI] [OHO,MIH] [RO,YI,FI]
```

0x53 LDY addr16,X (11 T)
```
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [PCM] [RO,OHI,CE] [XO,TI] [ATM,AOP,EO,MI,ACO]
[OHO,TI] [ATM,AHC,EO,MIH] [RO,YI,FI]
```

0x56 LDY addr8,X (7 T)
```
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [XO,TI] [ATM,AOP,EO,MI] [RO,YI,FI]
```

0x58 CPX #n (7 T)
```
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [XO,TI] [ORO,BI] [ATM,CMP,EO,FI]
```

0x59 CPX addr8 (8 T)
```
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [XO,TI] [ORO,MI] [RO,BI] [ATM,CMP,EO,FI]
```

0x5A CPX addr16 (11 T)
```
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [PCM] [RO,OHI,CE] [XO,TI] [ORO,MI] [OHO,MIH] [RO,BI]
[ATM,CMP,EO,FI]
```

0x61 STA addr8 (6 T)
```
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [ORO,MI] [AO,RI]
```

0x62 STA addr16 (9 T)
```
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [PCM] [RO,OHI,CE] [ORO,MI] [OHO,MIH] [AO,RI]
```

0x63 STA addr16,X (11 T)
```
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [PCM] [RO,OHI,CE] [XO,TI] [ATM,AOP,EO,MI,ACO]
[OHO,TI] [ATM,AHC,EO,MIH] [AO,RI]
```

0x64 STA addr16,Y (11 T)
```
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [PCM] [RO,OHI,CE] [YO,TI] [ATM,AOP,EO,MI,ACO]
[OHO,TI] [ATM,AHC,EO,MIH] [AO,RI]
```

0x65 `STA (addr8)` (10 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [ORO,MI] [RO,TI] [MRI] [RO,MIH] [TO,MIL] [AO,RI]

0x66 `STA addr8,X` (7 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [XO,TI] [ATM,AOP,EO,MI] [AO,RI]

0x67 `STA (addr8),Y` (13 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [ORO,MI] [RO,ORI] [YO,TI] [MRI] [RO,OHI] [ATM,AOP,EO,MI,ACO] [OHO,TI] [ATM,AHC,EO,MIH] [AO,RI]

0x69 `STX addr8` (6 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [ORO,MI] [XO,RI]

0x6A `STX addr16` (9 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [PCM] [RO,OHI,CE] [ORO,MI] [OHO,MIH] [XO,RI]

0x71 `STY addr8` (6 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [ORO,MI] [YO,RI]

0x72 `STY addr16` (9 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [PCM] [RO,OHI,CE] [ORO,MI] [OHO,MIH] [YO,RI]

0x78 `CPY #n` (7 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [YO,TI] [ORO,BI] [ATM,CMP,EO,FI]

0x79 `CPY addr8` (8 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [YO,TI] [ORO,MI] [RO,BI] [ATM,CMP,EO,FI]

0x7A `CPY addr16` (11 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [PCM] [RO,OHI,CE] [YO,TI] [ORO,MI] [OHO,MIH] [RO,BI] [ATM,CMP,EO,FI]

0x80 `ADD #n` (6 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [ORO,BI] [EO,AI,FI]

0x81 `ADD addr8` (7 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [ORO,MI] [RO,BI] [EO,AI,FI]

0x82 `ADD addr16` (10 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [PCM] [RO,OHI,CE] [ORO,MI] [OHO,MIH] [RO,BI] [EO,AI,FI]

0x83 `ADD addr16,X` (12 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [PCM] [RO,OHI,CE] [XO,TI] [ATM,AOP,EO,MI,ACO] [OHO,TI] [ATM,AHC,EO,MIH] [RO,BI] [EO,AI,FI]

0x84 `ADD addr16,Y` (12 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [PCM] [RO,OHI,CE] [YO,TI] [ATM,AOP,EO,MI,ACO] [OHO,TI] [ATM,AHC,EO,MIH] [RO,BI] [EO,AI,FI]

0x85 `ADD (addr8)` (11 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [ORO,MI] [RO,TI] [MRI] [RO,MIH] [TO,MIL] [RO,BI] [EO,AI,FI]

0x86 `ADD addr8,X` (8 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [XO,TI] [ATM,AOP,EO,MI] [RO,BI] [EO,AI,FI]

0x87 `ADD (addr8),Y` (14 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [ORO,MI] [RO,ORI] [YO,TI] [MRI] [RO,OHI] [ATM,AOP,EO,MI,ACO] [OHO,TI] [ATM,AHC,EO,MIH] [RO,BI] [EO,AI,FI]

0x88 `ADC #n` (6 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [ORO,BI] [CI,EO,AI,FI]

0x89 `ADC addr8` (7 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [ORO,MI] [RO,BI] [CI,EO,AI,FI]

0x8A `ADC addr16` (10 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [PCM] [RO,OHI,CE] [ORO,MI] [OHO,MIH] [RO,BI] [CI,EO,AI,FI]

0x8B `ADC addr16,X` (12 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [PCM] [RO,OHI,CE] [XO,TI] [ATM,AOP,EO,MI,ACO] [OHO,TI] [ATM,AHC,EO,MIH] [RO,BI] [CI,EO,AI,FI]

0x8C `ADC addr16,Y` (12 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [PCM] [RO,OHI,CE] [YO,TI] [ATM,AOP,EO,MI,ACO] [OHO,TI] [ATM,AHC,EO,MIH] [RO,BI] [CI,EO,AI,FI]

0x8D `ADC (addr8)` (11 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [ORO,MI] [RO,TI] [MRI] [RO,MIH] [TO,MIL] [RO,BI] [CI,EO,AI,FI]

0x8E `ADC addr8,X` (8 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [XO,TI] [ATM,AOP,EO,MI] [RO,BI] [CI,EO,AI,FI]

0x8F `ADC (addr8),Y` (14 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [ORO,MI] [RO,ORI] [YO,TI] [MRI] [RO,OHI] [ATM,AOP,EO,MI,ACO] [OHO,TI] [ATM,AHC,EO,MIH] [RO,BI] [CI,EO,AI,FI]

0x90 `SUB #n` (6 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [ORO,BI] [SU,EO,AI,FI]

0x91 `SUB addr8` (7 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [ORO,MI] [RO,BI] [SU,EO,AI,FI]

0x92 `SUB addr16` (10 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [PCM] [RO,OHI,CE] [ORO,MI] [OHO,MIH] [RO,BI] [SU,EO,AI,FI]

0x93 `SUB addr16,X` (12 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [PCM] [RO,OHI,CE] [XO,TI] [ATM,AOP,EO,MI,ACO] [OHO,TI] [ATM,AHC,EO,MIH] [RO,BI] [SU,EO,AI,FI]

0x94 `SUB addr16,Y` (12 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [PCM] [RO,OHI,CE] [YO,TI] [ATM,AOP,EO,MI,ACO] [OHO,TI] [ATM,AHC,EO,MIH] [RO,BI] [SU,EO,AI,FI]

0x95 `SUB (addr8)` (11 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [ORO,MI] [RO,TI] [MRI] [RO,MIH] [TO,MIL] [RO,BI] [SU,EO,AI,FI]

0x96 `SUB addr8,X` (8 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [XO,TI] [ATM,AOP,EO,MI] [RO,BI] [SU,EO,AI,FI]

0x97 `SUB (addr8),Y` (14 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [ORO,MI] [RO,ORI] [YO,TI] [MRI] [RO,OHI] [ATM,AOP,EO,MI,ACO] [OHO,TI] [ATM,AHC,EO,MIH] [RO,BI] [SU,EO,AI,FI]

0x98 `SBC #n` (6 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [ORO,BI] [SU,CI,EO,AI,FI]

0x99 `SBC addr8` (7 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [ORO,MI] [RO,BI] [SU,CI,EO,AI,FI]

0x9A `SBC addr16` (10 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [PCM] [RO,OHI,CE] [ORO,MI] [OHO,MIH] [RO,BI] [SU,CI,EO,AI,FI]

0x9B `SBC addr16,X` (12 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [PCM] [RO,OHI,CE] [XO,TI] [ATM,AOP,EO,MI,ACO] [OHO,TI] [ATM,AHC,EO,MIH] [RO,BI] [SU,CI,EO,AI,FI]

0x9C `SBC addr16,Y` (12 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [PCM] [RO,OHI,CE] [YO,TI] [ATM,AOP,EO,MI,ACO] [OHO,TI] [ATM,AHC,EO,MIH] [RO,BI] [SU,CI,EO,AI,FI]

0x9D `SBC (addr8)` (11 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [ORO,MI] [RO,TI] [MRI] [RO,MIH] [TO,MIL] [RO,BI] [SU,CI,EO,AI,FI]

0x9E `SBC addr8,X` (8 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [XO,TI] [ATM,AOP,EO,MI] [RO,BI] [SU,CI,EO,AI,FI]

0x9F `SBC (addr8),Y` (14 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [ORO,MI] [RO,ORI] [YO,TI] [MRI] [RO,OHI] [ATM,AOP,EO,MI,ACO] [OHO,TI] [ATM,AHC,EO,MIH] [RO,BI] [SU,CI,EO,AI,FI]

0xA0 `AND #n` (6 T)

[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [ORO,BI] [AND,EO,AI,FI]

0xA1 AND addr8 (7 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [ORO,MI] [RO,BI] [AND,EO,AI,FI]

0xA2 AND addr16 (10 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [PCM] [RO,OHI,CE] [ORO,MI] [OHO,MIH] [RO,BI] [AND,EO,AI,FI]

0xA3 AND addr16,X (12 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [PCM] [RO,OHI,CE] [XO,TI] [ATM,AOP,EO,MI,ACO] [OHO,TI] [ATM,AHC,EO,MIH] [RO,BI] [AND,EO,AI,FI]

0xA4 AND addr16,Y (12 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [PCM] [RO,OHI,CE] [YO,TI] [ATM,AOP,EO,MI,ACO] [OHO,TI] [ATM,AHC,EO,MIH] [RO,BI] [AND,EO,AI,FI]

0xA5 AND (addr8) (11 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [ORO,MI] [RO,TI] [MRI] [RO,MIH] [TO,MIL] [RO,BI] [AND,EO,AI,FI]

0xA6 AND addr8,X (8 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [XO,TI] [ATM,AOP,EO,MI] [RO,BI] [AND,EO,AI,FI]

0xA7 AND (addr8),Y (14 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [ORO,MI] [RO,ORI] [YO,TI] [MRI] [RO,OHI] [ATM,AOP,EO,MI,ACO] [OHO,TI] [ATM,AHC,EO,MIH] [RO,BI] [AND,EO,AI,FI]

0xA8 OR #n (6 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [ORO,BI] [OR,EO,AI,FI]

0xA9 OR addr8 (7 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [ORO,MI] [RO,BI] [OR,EO,AI,FI]

0xAA OR addr16 (10 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [PCM] [RO,OHI,CE] [ORO,MI] [OHO,MIH] [RO,BI] [OR,EO,AI,FI]

0xAB OR addr16,X (12 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [PCM] [RO,OHI,CE] [XO,TI] [ATM,AOP,EO,MI,ACO] [OHO,TI] [ATM,AHC,EO,MIH] [RO,BI] [OR,EO,AI,FI]

0xAC OR addr16,Y (12 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [PCM] [RO,OHI,CE] [YO,TI] [ATM,AOP,EO,MI,ACO] [OHO,TI] [ATM,AHC,EO,MIH] [RO,BI] [OR,EO,AI,FI]

0xAD OR (addr8) (11 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [ORO,MI] [RO,TI] [MRI] [RO,MIH] [TO,MIL] [RO,BI] [OR,EO,AI,FI]

0xAE OR addr8,X (8 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [XO,TI] [ATM,AOP,EO,MI] [RO,BI] [OR,EO,AI,FI]

0xAF OR (addr8),Y (14 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [ORO,MI] [RO,ORI] [YO,TI] [MRI] [RO,OHI] [ATM,AOP,EO,MI,ACO] [OHO,TI] [ATM,AHC,EO,MIH] [RO,BI] [OR,EO,AI,FI]

0xB0 XOR #n (6 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [ORO,BI] [XOR,EO,AI,FI]

0xB1 XOR addr8 (7 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [ORO,MI] [RO,BI] [XOR,EO,AI,FI]

0xB2 XOR addr16 (10 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [PCM] [RO,OHI,CE] [ORO,MI] [OHO,MIH] [RO,BI] [XOR,EO,AI,FI]

0xB3 XOR addr16,X (12 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [PCM] [RO,OHI,CE] [XO,TI] [ATM,AOP,EO,MI,ACO] [OHO,TI] [ATM,AHC,EO,MIH] [RO,BI] [XOR,EO,AI,FI]

0xB4 XOR addr16,Y (12 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [PCM] [RO,OHI,CE] [YO,TI] [ATM,AOP,EO,MI,ACO] [OHO,TI] [ATM,AHC,EO,MIH] [RO,BI] [XOR,EO,AI,FI]

0xB5 `XOR (addr8)` (11 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [ORO,MI] [RO,TI] [MRI] [RO,MIH] [TO,MIL] [RO,BI] [XOR,EO,AI,FI]

0xB6 `XOR addr8,X` (8 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [XO,TI] [ATM,AOP,EO,MI] [RO,BI] [XOR,EO,AI,FI]

0xB7 `XOR (addr8),Y` (14 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [ORO,MI] [RO,ORI] [YO,TI] [MRI] [RO,OHI] [ATM,AOP,EO,MI,ACO] [OHO,TI] [ATM,AHC,EO,MIH] [RO,BI] [XOR,EO,AI,FI]

0xB8 `CMP #n` (6 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [ORO,BI] [CMP,EO,FI]

0xB9 `CMP addr8` (7 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [ORO,MI] [RO,BI] [CMP,EO,FI]

0xBA `CMP addr16` (10 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [PCM] [RO,OHI,CE] [ORO,MI] [OHO,MIH] [RO,BI] [CMP,EO,FI]

0xBB `CMP addr16,X` (12 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [PCM] [RO,OHI,CE] [XO,TI] [ATM,AOP,EO,MI,ACO] [OHO,TI] [ATM,AHC,EO,MIH] [RO,BI] [CMP,EO,FI]

0xBC `CMP addr16,Y` (12 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [PCM] [RO,OHI,CE] [YO,TI] [ATM,AOP,EO,MI,ACO] [OHO,TI] [ATM,AHC,EO,MIH] [RO,BI] [CMP,EO,FI]

0xBD `CMP (addr8)` (11 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [ORO,MI] [RO,TI] [MRI] [RO,MIH] [TO,MIL] [RO,BI] [CMP,EO,FI]

0xBE `CMP addr8,X` (8 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [XO,TI] [ATM,AOP,EO,MI] [RO,BI] [CMP,EO,FI]

0xBF `CMP (addr8),Y` (14 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [ORO,MI] [RO,ORI] [YO,TI] [MRI] [RO,OHI] [ATM,AOP,EO,MI,ACO] [OHO,TI] [ATM,AHC,EO,MIH] [RO,BI] [CMP,EO,FI]

0xC0 `JNZ addr16` (7 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [PCM] [RO,OHI,CE] [JPW]?

0xC1 `JZ addr16` (7 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [PCM] [RO,OHI,CE] [JPW]?

0xC2 `JNC addr16` (7 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [PCM] [RO,OHI,CE] [JPW]?

0xC3 `JC addr16` (7 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [PCM] [RO,OHI,CE] [JPW]?

0xC4 `JP addr16` (7 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [PCM] [RO,OHI,CE] [JPW]?

0xC5 `JN addr16` (7 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [PCM] [RO,OHI,CE] [JPW]?

0xC6 `JNV addr16` (7 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [PCM] [RO,OHI,CE] [JPW]?

0xC7 `JV addr16` (7 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [PCM] [RO,OHI,CE] [JPW]?

0xC8 `JMP addr16` (7 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [PCM] [RO,OHI,CE] [JPW]

0xC9 `JMP (addr8)` (9 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [ORO,MI] [RO,TI] [MRI] [RO,JPH] [TO,JPL]

0xCA `CALL addr16` (11 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [PCM] [RO,OHI,CE] [SD,SPM] [CHO,RI] [SD,SPM] [CLO,RI] [JPW]

0xCB `RET` (6 T)

[PCM] [RO,II,CE] [SPM] [RO,JPL,SE] [SPM] [RO,JPH,SE]

0xCC RTI (8 T)
[PCM] [RO,II,CE] [SPM] [RO,FLI,SE] [SPM] [RO,JPL,SE] [SPM] [RO,JPH,SE]

0xCD BRK (12 T)
[PCM] [RO,II,CE] [SD,SPM] [CHO,RI] [SD,SPM] [CLO,RI] [SD,SPM] [FLO,RI,DIF] [VBR] [RO,JPL] [MRI] [RO,JPH]

0xCE CALL (addr8) (13 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [SD,SPM] [CHO,RI] [SD,SPM] [CLO,RI] [ORO,MI] [RO,TI] [MRI] [RO,JPH] [TO,JPL]

0xD0 INC addr8 (7 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [ORO,MI] [RO,TI] [ATM,INC,EO,RI,FI]

0xD1 INC addr16 (10 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [PCM] [RO,OHI,CE] [ORO,MI] [OHO,MIH] [RO,TI] [ATM,INC,EO,RI,FI]

0xD2 DEC addr8 (7 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [ORO,MI] [RO,TI] [ATM,DEC,EO,RI,FI]

0xD3 DEC addr16 (10 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [PCM] [RO,OHI,CE] [ORO,MI] [OHO,MIH] [RO,TI] [ATM,DEC,EO,RI,FI]

0xD4 INW addr8 (10 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [ORO,MI] [RO,TI] [ATM,INC,EO,RI,ACO] [MRI] [RO,TI] [ATM,AHC,EO,RI]

0xD5 INW addr16 (13 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [PCM] [RO,OHI,CE] [ORO,MI] [OHO,MIH] [RO,TI] [ATM,INC,EO,RI,ACO] [MRI] [RO,TI] [ATM,AHC,EO,RI]

0xD6 DEW addr8 (10 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [ORO,MI] [RO,TI] [ATM,DEC,EO,RI,ACO] [MRI] [RO,TI] [ATM,AHB,EO,RI]

0xD7 DEW addr16 (13 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [PCM] [RO,OHI,CE] [ORO,MI] [OHO,MIH] [RO,TI] [ATM,DEC,EO,RI,ACO] [MRI] [RO,TI] [ATM,AHB,EO,RI]

0xD8 SHL addr8 (7 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [ORO,MI] [RO,TI] [ATM,SHL,EO,RI,FI]

0xD9 SHL addr16 (10 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [PCM] [RO,OHI,CE] [ORO,MI] [OHO,MIH] [RO,TI] [ATM,SHL,EO,RI,FI]

0xDA SHR addr8 (7 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [ORO,MI] [RO,TI] [ATM,SHR,EO,RI,FI]

0xDB SHR addr16 (10 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [PCM] [RO,OHI,CE] [ORO,MI] [OHO,MIH] [RO,TI] [ATM,SHR,EO,RI,FI]

0xDC ROL addr8 (7 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [ORO,MI] [RO,TI] [ATM,ROL,EO,RI,FI]

0xDD ROL addr16 (10 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [PCM] [RO,OHI,CE] [ORO,MI] [OHO,MIH] [RO,TI] [ATM,ROL,EO,RI,FI]

0xDE ROR addr8 (7 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [ORO,MI] [RO,TI] [ATM,ROR,EO,RI,FI]

0xDF ROR addr16 (10 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [PCM] [RO,OHI,CE] [ORO,MI] [OHO,MIH] [RO,TI] [ATM,ROR,EO,RI,FI]

0xE0 JGT addr16 (7 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [PCM] [RO,OHI,CE] [JPW]?

0xE1 JLE addr16 (7 T)

```
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [PCM] [RO,OHI,CE] [JPW]?
```

0xE2 `JGES addr16` (7 T)
```
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [PCM] [RO,OHI,CE] [JPW]?
```

0xE3 `JLTS addr16` (7 T)
```
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [PCM] [RO,OHI,CE] [JPW]?
```

0xE4 `BIT #n` (6 T)
```
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [ORO,BI] [AND,EO,FI]
```

0xE5 `BIT addr8` (7 T)
```
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [ORO,MI] [RO,BI] [AND,EO,FI]
```

0xE6 `BIT addr16` (10 T)
```
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [PCM] [RO,OHI,CE] [ORO,MI] [OHO,MIH] [RO,BI]
[AND,EO,FI]
```

0xE8 `ADW addr8,#n` (13 T)
```
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [PCM] [RO,OHI,CE] [ORO,MI] [OHO,BI] [RO,TI]
[ATM,EO,RI,ACO] [MRI] [RO,TI] [ATM,AHC,EO,RI]
```

0xE9 `SBW addr8,#n` (13 T)
```
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [PCM] [RO,OHI,CE] [ORO,MI] [OHO,BI] [RO,TI]
[ATM,SU,EO,RI,ACO] [MRI] [RO,TI] [ATM,AHB,EO,RI]
```

0xEA `LXY #nn` (8 T)
```
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [PCM] [RO,OHI,CE] [ORO,YI] [OHO,XI]
```

0xEB `SXY addr8` (8 T)
```
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [ORO,MI] [YO,RI] [MRI] [XO,RI]
```

0xFD `RESET` (4 T)
```
[VRS] [RO,JPL] [MRI] [RO,JPH]
```

0xFE `IRQ` (10 T)
```
[SD,SPM] [CHO,RI] [SD,SPM] [CLO,RI] [SD,SPM] [FLO,RI,DIF] [VIR] [RO,JPL] [MRI] [RO,JPH]
```

APPENDIX D

Control Signals

All 70 control signals, split into six mutually exclusive groups. That split is not just documentation — it is the microword's field layout, from which the FPGA's ROM is generated, and **the order of members inside a group is part of the encoding**.

Naming convention: a signal ending in `O` usually drives the data bus, one ending in `I` latches from it. At most one bus source may be asserted per T-state; there may be several sinks.

Data-bus sources

Signal	Description
`RO`	RAM Out - read memory at MAR onto the data bus
`AO`	A Register Out
`BO`	B Register Out
`XO`	X Register Out
`YO`	Y Register Out
`TO`	TMP Register Out
`ORO`	Operand (low) Register Out
`OHO`	Operand-high Register Out
`CLO`	PC low half onto the data bus (CALL/IRQ push)
`CHO`	PC high half onto the data bus (CALL/IRQ push)
`SLO`	SP low half onto the data bus (TSXY)
`SHO`	SP high half onto the data bus (TSXY)
`FLO`	Flags Out - pack C/Z/N/V/I into a status byte on the bus
`EO`	ALU Out - ALU result onto the data bus
`EXO`	ALU secondary result out (MUL high byte / DIV remainder)

Data-bus sinks

Signal	Description
`MI`	MAR In, zero-extended - MAR := 0x00:bus (zero-page address)

Signal	Description
MIL	MAR low half := bus
MIH	MAR high half := bus
RI	RAM In - write the data bus to memory at MAR
II	Instruction Register In
AI	A Register In
BI	B Register In
XI	X Register In
YI	Y Register In
TI	TMP Register In
ORI	Operand (low) Register In
OHI	Operand-high Register In
OI	Output Register In
JPL	PC low half := bus (RET/RTI pop)
JPH	PC high half := bus (RET/RTI pop)
SLI	SP low half := bus (TXYS)
SHI	SP high half := bus (TXYS)
FLI	Flags In - restore C/Z/N/V/I from a status byte on the bus

Address path (MAR)

Signal	Description
PCM	MAR := PC (16-bit address path)
SPM	MAR := SP (16-bit address path)
MRI	MAR := MAR + 1 (second byte of a pointer or vector)
VRS	MAR := 0xFFFC (RESET vector base)
VIR	MAR := 0xFFFE (IRQ vector base)
VBR	MAR := 0xFFFA (BRK vector base)

ALU modes

Signal	Description
SU	ALU Subtract mode
AND	ALU AND mode

Signal	Description
`OR`	ALU OR mode
`XOR`	ALU XOR mode
`NOT`	ALU NOT mode
`INC`	ALU Increment mode
`DEC`	ALU Decrement mode
`SHL`	ALU Shift Left mode
`SHR`	ALU Shift Right mode
`ROL`	ALU Rotate Left through carry
`ROR`	ALU Rotate Right through carry
`CMP`	ALU Compare mode (flags only, result discarded)
`MUL`	ALU Multiply mode - A×B, low byte out, high byte to EXO
`DIV`	ALU Divide mode - A÷B, quotient out, remainder to EXO
`AHC`	ALU := TMP + addrCarry (high-byte address adjust)
`AHB`	ALU := TMP - addrCarry (high-byte borrow adjust)

Side effects

Signal	Description
`CFC`	Clear Carry flag (CLC)
`CFS`	Set Carry flag (SEC)
`EIF`	Enable Interrupt flag (EI)
`DIF`	Disable Interrupt flag (DI)
`HLT`	Halt - stop until an enabled interrupt arrives
`RND`	Random byte into A (seedable xorshift32)

Standalone microword bits

Signal	Description
`JPW`	PC := OPH:OP (16-bit address path - absolute jump)
`CE`	Program Counter Enable - PC := PC + 1
`SD`	Stack Pointer Decrement (16-bit)
`SE`	Stack Pointer Increment (16-bit)
`CI`	ALU carry-in from the C flag (ADC/SBC)

Signal	Description
`ATM`	ALU input mux - TMP instead of A
`AOP`	ALU input mux - operand register instead of B
`ACO`	Latch ALU carry/borrow-out into addrCarry (flags untouched)
`FI`	Flags In - latch flags (ALU flags with EO, Z/N of the bus otherwise)

APPENDIX E

Memory Map and Peripheral Registers

Address space

Address	Size	Region
`0x0000–0x00FF`	256 B	Zero page — fast access and the pointer file
`0x0100–0x01FF`	256 B	Stack
`0x0200–0x7FFF`	32256 B	Program and data
`0x8000–0x83E7`	1000 B	Character buffer (40×25)
`0x83E8–0x83FF`	24 B	reserved
`0x8400–0x8BFF`	2048 B	Glyph RAM (8 bytes per character)
`0x8C00–0x8FE7`	1000 B	Color RAM (one attribute per cell)
`0x8FE8–0x8FFF`	24 B	reserved
`0x9000–0xAF3F`	8000 B	Bitmap framebuffer
`0xAF40–0xAFDF`	160 B	reserved
`0xAFE0–0xAFFF`	32 B	Sprite attributes (8 × 4 bytes)
`0xB000–0xBFFF`	4096 B	Sprite patterns (32 × 128 bytes)
`0xC000–0xFEFF`	16128 B	Data RAM — large arrays, heap
`0xFF00–0xFF7F`	128 B	Memory-mapped I/O
`0xFF80–0xFFF9`	122 B	System RAM — ISR scratch, boot-ROM home
`0xFFFA–0xFFFF`	6 B	BRK / RESET / IRQ vectors

The rows add up to exactly 64 KB — the ones marked reserved are the holes nothing uses today.

Vectors

Address	Vector	Read
0xFFFA	BRK	when the `BRK` instruction executes
0xFFFC	RESET	at power-up and on reset
0xFFFE	IRQ	when a hardware interrupt is taken

All three are two bytes, low byte first.

Peripheral registers

The block 0xFF00–0xFF7F, eight devices of sixteen bytes each. The device number is address bits [6:4].

Keyboard

Address	Register
0xFF00	KBD_DATA
0xFF01	KBD_STATUS

Terminal

Address	Register
0xFF10	TERM_DATA
0xFF11	TERM_STATUS

Audio

Address	Register
0xFF20	AUD_CTRL
0xFF21	AUD_FREQ_LO
0xFF22	AUD_FREQ_HI
0xFF23	AUD_WAVE
0xFF24	AUD_AD
0xFF25	AUD_SR

Timer and counters

Address	Register
0xFF30	TMR_DIV
0xFF31	TMR_CNT
0xFF32	CYC0
0xFF33	CYC1
0xFF34	CYC2
0xFF35	CYC3
0xFF36	UPT_LO

Address	Register
0xFF37	UPT_HI
0xFF38	CLK_TPMS_LO
0xFF39	CLK_TPMS_HI
0xFF3A	CLK_MS_LO
0xFF3B	CLK_MS_HI

Interrupt control

Address	Register
0xFF40	IRQ_ENABLE
0xFF41	IRQ_STATUS

Video

Address	Register
0xFF50	VID_CTRL
0xFF51	VID_STATUS
0xFF52	VID_CURX
0xFF53	VID_CURY
0xFF54	VID_BORDER
0xFF55	VID_SCROLLX
0xFF56	VID_SCROLLY
0xFF57	VID_MCOLOR
0xFF58	SPR_ENABLE

Storage and serial

Address	Register
0xFF60	DSK_CMD
0xFF61	DSK_STATUS
0xFF62	DSK_BLK
0xFF63	DSK_ADDR_LO
0xFF64	DSK_ADDR_HI
0xFF65	SER_DATA
0xFF66	SER_STATUS
0xFF67	SER_CTRL

Address	Register
0xFF68	DSK_DISK

Gamepads and blitter

Address	Register
0xFF70	PAD1
0xFF71	PAD2
0xFF72	BLT_SRC_LO
0xFF73	BLT_SRC_HI
0xFF74	BLT_DST_LO
0xFF75	BLT_DST_HI
0xFF76	BLT_LEN_LO
0xFF77	BLT_LEN_HI
0xFF78	BLT_FILL
0xFF79	BLT_CTRL
0xFF7A	BLT_STATUS

Audio channels

The three channels repeat the same five registers:

Address	Register
0xFF21	AUD0_FREQ_LO
0xFF22	AUD0_FREQ_HI
0xFF23	AUD0_WAVE
0xFF24	AUD0_AD
0xFF25	AUD0_SR
0xFF26	AUD1_FREQ_LO
0xFF27	AUD1_FREQ_HI
0xFF28	AUD1_WAVE
0xFF29	AUD1_AD
0xFF2A	AUD1_SR
0xFF2B	AUD2_FREQ_LO
0xFF2C	AUD2_FREQ_HI
0xFF2D	AUD2_WAVE
0xFF2E	AUD2_AD

Address	Register
0xFF2F	AUD2_SR

APPENDIX F

Character Set and Palette

The built-in font

Glyph memory has 256 slots. The built-in font fills 97 of them — the printable part of ASCII (0x20–0x7E) plus three symbols at 0x01–0x03. The row is the code's high nibble, the column its low.

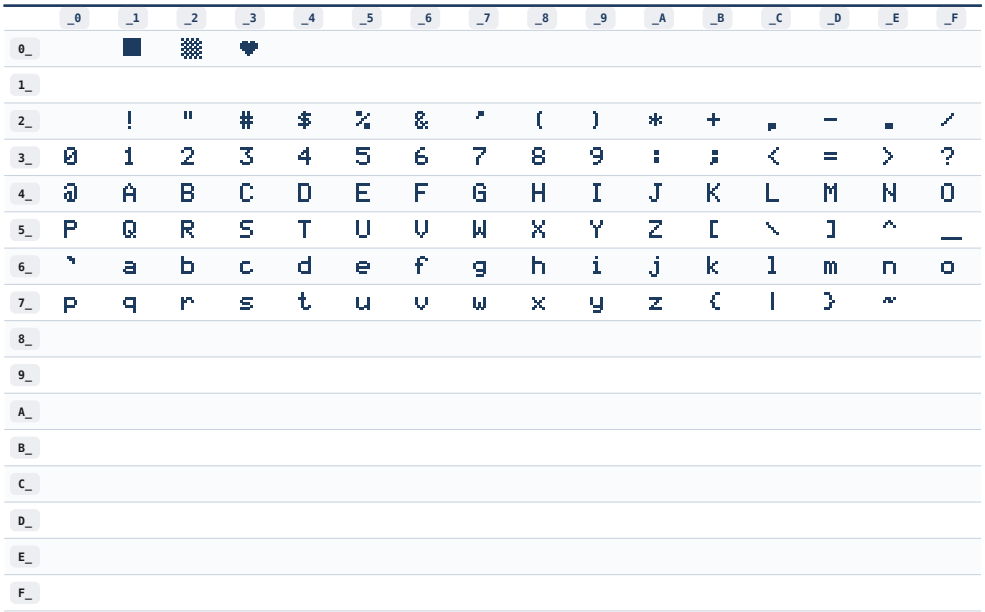

Figure 14. The built-in font. Blank cells are free slots, not spaces.

The whole upper half (0x80–0xFF) is **empty and available**. It is the easiest place for a program to keep its own symbols without giving up the font — tiles, icons, pieces of a playfield.

Glyphs are not in ROM: they live in memory at 0x8400–0x8BFF, eight bytes per character, one byte per row, most significant bit leftmost. A program may overwrite them at any time.

Printable characters

The range 0x20–0x7F matches ASCII.

Opcode	Char	Opcode	Char	Opcode	Char	Opcode	Char
0x20		0x38	8	0x50	P	0x68	h
0x21	!	0x39	9	0x51	Q	0x69	i
0x22	"	0x3A	:	0x52	R	0x6A	j
0x23	#	0x3B	;	0x53	S	0x6B	k
0x24	$	0x3C	<	0x54	T	0x6C	l
0x25	%	0x3D	=	0x55	U	0x6D	m
0x26	&	0x3E	>	0x56	V	0x6E	n
0x27	'	0x3F	?	0x57	W	0x6F	o
0x28	(	0x40	@	0x58	X	0x70	p
0x29	)	0x41	A	0x59	Y	0x71	q
0x2A	*	0x42	B	0x5A	Z	0x72	r
0x2B	+	0x43	C	0x5B	[	0x73	s
0x2C	,	0x44	D	0x5C	\	0x74	t
0x2D	-	0x45	E	0x5D	]	0x75	u
0x2E	.	0x46	F	0x5E	^	0x76	v
0x2F	/	0x47	G	0x5F	_	0x77	w
0x30	0	0x48	H	0x60	`	0x78	x
0x31	1	0x49	I	0x61	a	0x79	y
0x32	2	0x4A	J	0x62	b	0x7A	z
0x33	3	0x4B	K	0x63	c	0x7B	{
0x34	4	0x4C	L	0x64	d	0x7C	|
0x35	5	0x4D	M	0x65	e	0x7D	}
0x36	6	0x4E	N	0x66	f	0x7E	~
0x37	7	0x4F	O	0x67	g	0x7F	

Figure 15. The printable part of the character set with its codes.

Terminal control codes

Code	Escape	Effect
0x08	`\b`	erase the previous character
0x09	`\t`	tab to the next 8-column stop
0x0A	`\n`	new line
0x0C	`\f`	clear the terminal screen
0x0D	`\r`	return to the start of the line
0x00	`\0`	string terminator

Palette

Index	Color	Index	Color
0	#000000	8	#8b5429
1	#ffffff	9	#574200
2	#883932	10	#b86962
3	#67b6bd	11	#505050
4	#8b3f96	12	#787878
5	#55a049	13	#94e089
6	#40318d	14	#7869c4
7	#bfce72	15	#9f9f9f

Figure 16. The 16-color palette, shared by color RAM, `VID_MCOLOR` and `VID_BORDER`.

Indices go into color RAM as nibbles: the low one is the foreground, the high one the background. 0x00 means the default look, not black on black.

APPENDIX G

Instruction Index

Alphabetical index of every mnemonic, jump aliases included, with the page its dictionary entry starts on.