Počítač do dlaně

Jak doopravdy funguje procesor

Malý stroj, který se dá pochopit celý — bit po bitu.

Jan Müller

2026

Obsah

ČÁST V — Poskládáme to dohromady

Přílohy

Předmluva

Většina lidí dnes používá počítač, aniž by kdy viděla, co se děje uvnitř. A není divu — moderní procesor je zázrak s miliardami tranzistorů, které se přepínají miliardykrát za sekundu. Podívat se takovému stroji „pod kapotu“ je jako snažit se pochopit velkoměsto tak, že budeš zírat na jednotlivé cihly.

Tahle kniha jde na to opačně. Vezme počítač tak malý, že se vejde do dlaně — a do hlavy — a projde ho kompletně: každý registr, každý drát, každý takt hodin. Stroj se jmenuje **SAP-3/16** a je to schválně zjednodušený osmibitový procesor, přesně na hraně mezi „hračka na tabuli“ a „skutečný počítač“. Až dojdeš na konec, nebude na něm jediná část, kterou bys neuměl vysvětlit — a co je důležitější, budeš rozumět i tomu velkému stroji, na kterém právě čteš tuhle větu. Protože v jádru dělají oba přesně totéž.

Co je SAP-3/16

SAP je zkratka **Simple As Possible** — „co nejjednodušší“. Je to rodina výukových procesorů z klasické učebnice Alberta Malvina, po desetiletí používaná k výuce toho, jak počítače doopravdy fungují. Náš stroj, **SAP-3/16**, je nejbohatší člen řady: osmibitová data, ale **šestnáctibitové adresy**, takže vidí celých 64 kilobajtů paměti. Má registry, aritmetickou jednotku, zásobník, přerušení, a dokonce zvukový čip a obrazovku. To všechno jsou přesně ty prvky, které dělaly domácí počítače přelomu sedmdesátých a osmdesátých let — stroje jako Commodore 64 nebo ZX Spectrum — a všechno se to tu vejde do několika stran vysvětlení.

Nejlepší na tom je, že SAP-3/16 není jen na papíře. Existuje jako **emulátor**, který běží v prohlížeči: vidíš v něm skutečné schéma procesoru, jak po drátech putují data, jak se rozsvěcejí řídicí signály, a můžeš celý stroj krokovat takt po taktu. Tahle kniha a ten emulátor jsou dvě strany téže mince — kniha vysvětluje, co vidíš, a emulátor ukazuje, o čem čteš.

Pro koho je tato kniha

Nepotřebuješ žádné předchozí znalosti o hardwaru. Nemusíš umět programovat v assembleru — naučíš se to tady. Stačí zvědavost a chuť občas si tužkou přepočítat pár bitů. Vysvětlujeme od úplného začátku: kapitola 1 začíná otázkou „co je to vlastně bit“ a odtud stavíme, cihlu po cihle, až k hotovému počítači, který hraje hudbu a běhají na něm hry.

Pokud naopak nějaké základy máš, klidně přeskakuj. Kapitoly na sebe navazují, ale každá stojí i sama o sobě a vzadu najdeš referenční přílohy — kompletní tabulku instrukcí, mapu paměti, seznam řídicích signálů — kam se dá kdykoli nahlédnout.

Jak knihu číst

Kniha je rozdělená do pěti částí. **Část I** položí základy: čísla, paměť, myšlenku uloženého programu. **Část II** otevře procesor a projde jeho vnitřek — registry, sběrnici, aritmetickou jednotku, instrukční cyklus. **Část III** tě naučí stroj programovat v assembleru. **Část IV** připojí svět: paměť, přerušení, klávesnici, zvuk a obraz. A **Část V** všechno poskládá dohromady rozborem opravdové hry a pohledem na to, jak se z tohohle malého stroje stal ten ve tvém telefonu.

V textu potkáš tři druhy zvýrazněných poznámek:

> **Poznámka**
>
> Doplňující souvislost nebo pojem, který se hodí znát, ale hlavní tok výkladu nebrzdí.

> **Tip**
>
> Praktická rada z programátorské praxe — jak něco udělat chytře, rychle nebo elegantně.

> **Pozor**
>
> Zrádné místo, snadný omyl nebo past, do které začátečníci pravidelně spadnou.

Konvence

Čísla zapisujeme třemi způsoby a je dobré si na to hned zvyknout. `42` je běžné desítkové číslo, `0x2A` je totéž šestnáctkově (hexadecimálně) a `0b00101010` binárně. Proč tři soustavy? Protože každá se hodí na něco jiného — a kapitola 1 přesně vysvětlí, kdy kterou použít.

Kód uvidíš ve dvou podobách. Krátké úryvky v textu, jako `LDA #42`, jsou vysázené strojopisem. Delší ukázky stojí v samostatných rámečcích:

```
; Vytiskni písmeno A na terminál
LDA #'A'        ; načti ASCII kód 65 do akumulátoru
```

```
STA 0xFF10      ; zapiš ho do registru terminálu
HLT             ; zastav procesor
```

Všechny ukázky v této knize jsou skutečné programy, které v emulátoru běží — jejich výstup není opsaný z hlavy, ale pořízený spuštěním. Nejlíp se učí tím, že si je sám vyzkoušíš: otevři emulátor, napiš program, stiskni krok a dívej se, jak stroj ožívá.

Tak pojďme na to. Začneme tím nejmenším, co v počítači existuje — jediným bitem.

ČÁST I

Než začneme počítat

KAPITOLA 1

Od bitů k číslům

Všechno, co počítač dělá — každá hra, každá skladba, každá věta na obrazovce — stojí na jediné fyzikální myšlence: na drátu buď napětí je, nebo není. Žádná třetí možnost, žádné „skoro“. Tyto dva stavy nazýváme **1** a **0** a jedné takové ano/ne hodnotě říkáme **bit**. Je to nejmenší kousek informace, jaký existuje — a je fascinující, že z ničeho víc než z milionů takových vypínačů se dá postavit stroj, který počítá, kreslí i hraje.

Jeden bit sám o sobě rozliší jen dvě věci: ano/ne, zapnuto/vypnuto, pravda/nepravda. To je málo. Proto bity sdružujeme do skupin, přesně jako z jednotlivých písmen skládáme slova. Osm bitů tvoří **bajt** (anglicky *byte*) a to je základní stavební jednotka téměř každého počítače na světě.

Kolik různých hodnot pojme jeden bajt? Každý z osmi bitů má dvě možnosti, takže dohromady je jich $2^8 = 256$. Bajt tedy umí uložit číslo od 0 do 255 — dohromady 256 různých hodnot. To je celý jeho svět. Nula je `0b00000000`, dvě stě padesát pět je `0b11111111`, a mezi tím se vejde všechno ostatní.

SAP-3/16 je **osmibitový** počítač: jeho registry (to jsou vnitřní pracovní buňky, ke kterým se dostaneme v kapitole 3), datová sběrnice i paměťové buňky jsou široké přesně jeden bajt. Největší číslo, které se vejde do jednoho registru, je proto 255. Zní to jako svěrací kazajka — a taky že je, přesně v tom je kus kouzla. Naučit počítač počítat s většími čísly, když má k dispozici jen osmibitové kapsy, je jedna z prvních věcí, které budeme muset vyřešit.

Adresy jsou širší

Data v tomhle stroji měří jeden bajt. Ale **adresy** — čísla, kterými říkáme „chci buňku paměti číslo tolik a tolik“ — jsou jiná káva. Ty mají **šestnáct bitů**, tedy dva bajty. Šestnáct bitů dává 2^{16}, tedy 65 536 různých hodnot, takže SAP-3/16 umí adresovat 65 536 paměťových míst. Tomu se říká adresní prostor **64 KB** a zapisuje se od `0x0000` do `0xFFFF`.

Toto rozdělení — **osmibitová data, šestnáctibitové adresy** — není náhoda. Přesně takhle byly postavené slavné procesory 6502 (Apple II, Commodore 64, Nintendo) a Z80 (ZX Spectrum). A vytváří to napětí, které se potáhne celou knihou: šestnáctibitová adresa se do osmibitové cesty na jeden zátah nevejde. Kdykoli bude stroj sahat do paměti, uvidíme, jak se s tímhle rozporem pere.

> **Poznámka**
>
> Číslo za lomítkem v názvu „SAP-3/16“ znamená právě těch šestnáct bitů adresy. Je to hlavní věc, kterou tento stroj přidává k učebnicovému SAP-3: dost velkou paměť na to, aby se v ní daly dělat skutečné programy.

Dvojková a šestnáctková soustava

Bajt zapsaný binárně je řetězec osmi jedniček a nul: `0b00101010`. Jak z toho přečteme číslo? Stejně jako v desítkové soustavě — jen místo mocnin deseti jsou to mocniny dvou. Zprava doleva mají pozice váhy 1, 2, 4, 8, 16, 32, 64, 128. Sečteme váhy tam, kde je jednička:

$$\texttt{0b00101010} = 32 + 8 + 2 = 42$$

Binární zápis vidí hardware, ale pro člověka je nepřehledný — kdo si má pamatovat osm cifer? Proto programátoři používají **šestnáctkovou soustavu** (hexadecimální, zkráceně „hex“). Její kouzlo je v tom, že jedna hex číslice kóduje přesně čtyři bity, tedy půl bajtu. Číslic je šestnáct: 0–9 a pak A–F pro hodnoty 10 až 15.

Stejné číslo 42 je v hexu `0x2A`: horní čtyři bity `0010` jsou 2, dolní `1010` jsou 10, tedy A. Dvě hex číslice tak vždy popisují přesně jeden bajt a čtyři hex číslice celou šestnáctibitovou adresu. Proto se v tomhle emulátoru adresy píšou jako `0x0000` až `0xFFFF` — čtyři cifry, jeden pohled, celá adresa.

> **Tip**
>
> Převod mezi hexem a binárkou se vyplatí umět z hlavy pro šestnáct hodnot 0–F. Pak čteš `0x2A` rovnou jako `0010 1010` a naopak. Je to jako naučit se abecedu — pár minut dřiny a pak už to jde samo.

Záporná čísla: dvojkový doplněk

Bajt nemá kam napsat znaménko minus — je to jen osm bitů. Jak tedy uložit −1? Odpověď, kterou používají opravdu všechny počítače, se jmenuje **dvojkový doplněk**. Trik je jednoduchý: nejvyšší bit (bit 7) prohlásíme za znaménkový a dáme mu váhu **minus** 128 místo plus 128. Těch samých 256 bitových vzorů se pak dá číst jako rozsah od −128 do +127:

Bitový vzor	Bez znaménka	Se znaménkem
`0b00000000`	0	0
`0b00000001`	1	+1
`0b01111111`	127	+127
`0b10000000`	128	−128
`0b11111111`	255	−1

Tabulka 1. Stejných osm bitů, dvě různá čtení. Který výklad platí, rozhoduje programátor.

Krása dvojkového doplňku je v tom, že aritmeticko-logická jednotka se o znaménko **vůbec nemusí starat**. Sčítání i odčítání bitových vzorů funguje úplně stejně, ať je **interpretuješ** jako bezznaménková (0 až 255), nebo znaménková (−128 až +127) čísla. Výsledek jsou tytéž bity; teprve ty se rozhodneš, jak je přečíst.

Procesor ti s tou volbou pomáhá dvěma **příznaky** — malými jednobitovými poznámkami o posledním výsledku. **N** (negative) je prostě kopie bitu 7, tedy „vypadá to jako záporné“, a **V** (overflow) hlásí znaménkové přetečení. K příznakům se dostaneme v kapitole 5 a v kapitole 12 uvidíme, jak se podle každé interpretace správně rozhodovat. Prozatím si zapamatuj jednu větu: **stejné bity mohou znamenat dvě různá čísla a je na tobě, které máš na mysli.**

Text jsou taky jen čísla

Když je bajt jen číslo 0 až 255, jak v něm může být uložené písmeno? Odpovědí je dohoda. Tabulka **ASCII** (vyslovuj „aski“) přiřazuje číslům 0 až 127 znaky: hodnota 65 znamená `A`, hodnota 48 znamená znak `0`, hodnota 32 je mezera a hodnota 10 znamená „přejdi na nový řádek“. Je to prostě číselník, na kterém se svět kdysi domluvil.

Když v assembleru napíšeš `'A'`, překladač za to beze všeho dosadí číslo 65 — je to jen pohodlnější zápis téhož bajtu. A když ten bajt zapíšeš do zvláštní adresy, na kterou je připojený terminál (uvidíme, že je to `0xFF10`), objeví se na obrazovce písmeno A. Žádný „text“ ve stroji doopravdy neexistuje; jsou to pořád jen bajty, kterým jsme dali dohodnutý význam.

> **Poznámka**
>
> Kompletní tabulku ASCII najdeš v příloze D. Nemusíš se ji učit nazpaměť — ale vědět, že `'0'` je 48 a `'A'` je 65, se při programování hodí pořád dokola.

To je celý základ. Počítač nezná čísla, písmena ani barvy — zná jen bity, a všechno ostatní je otázka dohody, jak ty bity číst. V další kapitole se podíváme, jak z paměti

plné bajtů a jednoho neúnavného procesoru vznikne stroj, který umí spustit libovolný program.

KAPITOLA 2

Von Neumannův stroj a rodina SAP

V kapitole 1 jsme viděli, že počítač zná jen bajty. Teď přijde druhá velká myšlenka, na které stojí úplně všechno ostatní: co když jsou **i samotné instrukce programu** jen bajty v téže paměti jako data? Zní to nenápadně, ale je to jeden z nejmocnějších nápadů dvacátého století.

Uložený program

Téměř každý počítač od čtyřicátých let sleduje takzvanou **von Neumannovu architekturu**, pojmenovanou po matematikovi Johnu von Neumannovi. Její recept je prostý: je tu **procesor**, který počítá; jediná **paměť**, která drží jak instrukce programu, tak data, se kterými program pracuje; a **vstupně-výstupní zařízení**, kterými stroj komunikuje se světem. Všechno propojuje **sběrnice** — svazek drátů, po kterých tečou bajty sem a tam.

Klíčem k celému kouzlu je **uložený program**: instrukce nejsou zadrátované napevno, jsou to obyčejné bajty na obyčejných adresách. Díky tomu může počítač nahrát nový program prostě tím, že do paměti zapíše jinou sadu bajtů — bez šroubováku, bez přepojování drátů. Právě proto tvůj telefon umí spustit hru i mailový klient a bankovní aplikaci: jsou to jen různé bajty načtené do téže paměti.

> **Poznámka**
>
> Před von Neumannem se stroje často „programovaly“ fyzickým přepojováním kabelů — přepnout program znamenalo hodiny práce s dráty. Myšlenka uloženého programu tohle celé odstranila a je důvodem, proč je dnešní počítač univerzální stroj, a ne jednoúčelový přístroj.

Kód, nebo data? Stroj to sám nepozná

Uložený program má jeden slavný a trochu strašidelný důsledek: procesor **sám od sebe nerozezná kód od dat**. Bajt `0x8D` je pro něj instrukce, nebo číslo 141 — podle toho, jestli na něj zrovna ukazuje jako na příští instrukci k vykonání, nebo jako na data ke čtení. Rozdíl není v bajtu, ale v tom, **jak se na něj procesor právě dívá.**

Když se program splete a jeho ukazatel na příští instrukci (program counter, potkáme ho v kapitole 3) zabloudí do oblasti dat, procesor je začne vesele „vykonávat“

jako instrukce — s naprosto nesmyslnými následky. Tohle je zdroj celé jedné třídy chyb ve skutečných počítačích.

SAP-3/16 z toho dělá výukový moment. Když se procesor pokusí vykonat bajt `0x00` (tak vypadá prázdná, nevyplněná paměť) nebo jakýkoli jiný nedefinovaný vzor, zastaví se s chybou — takzvaným **trapem** — která přesně řekne, který bajt na které adrese ho zmátl. To je typický příznak programu, který „přeběhl“ svůj konec a spadl do dat za ním.

> **Pozor**
>
> Zapomenout na konci programu na `HLT` (zastav) je klasická začátečnická chyba. Bez něj procesor pokračuje za poslední instrukcí do prázdné paměti — a spadne do trapu. Emulátor ti naštěstí přesně ukáže kde.

Řada SAP: co nejjednodušší

Stroj v této knize patří do rodiny učebnicových procesorů **SAP** — zkratka znamená **Simple As Possible**, tedy „co nejjednodušší“. Vymyslel ji Albert Malvino pro svou klasickou učebnici *Digital Computer Electronics* a po desetiletí se na ní studenti učí, jak počítač funguje, protože je dost malá na to, aby se dala pochopit celá — a přitom dost skutečná, aby to mělo smysl. Řada roste ve třech krocích:

- **SAP-1** — holé minimum: čítač instrukcí, jeden pracovní registr, sčítačka a hrstka instrukcí. Umí sečíst dvě čísla a ukázat výsledek. Víc ne.
- **SAP-2** — přidává skoky, další registry a logické operace. Tím se z kalkulačky stává skutečně *programovatelný* stroj: umí se rozhodovat a opakovat.
- **SAP-3** — „opravdový“ malý počítač: bohatší instrukční sada, indexové registry pro práci s poli, zásobník a podprogramy.

Co přidává SAP-3/16

Emulátor v této knize implementuje **SAP-3/16** — SAP-3 rozšířený o určující prvky skutečných osmibitových strojů přelomu sedmdesátých a osmdesátých let. Právě tyhle přídavky dělají rozdíl mezi „ukázkou na tabuli“ a počítačem, na kterém se dá napsat hra:

- **šestnáctibitová adresní sběrnice** nad plochou pamětí 64 KB (odtud „/16“),
- **nultá stránka** pro rychlý přístup a ukazatele (kapitola 14),
- **startovací a přerušovací vektory** na vrcholu paměti,
- **hardwarová přerušení** od časovače a klávesnice (kapitola 15),
- **paměťově mapované I/O** — klávesnice, terminál, zvuk, obraz (kapitola 16 a dál).

Všechny tyhle prvky jsou vzaté téměř doslova z procesoru 6502, srdce Commodore 64, Apple II i prvního Nintenda. To má příjemný důsledek: skoro všechno, co se tu naučíš, se přímo přenáší na četbu o skutečných strojích jako 6502, Z80 nebo 8080. SAP-3/16 není slepá ulička — je to *brána* ke skutečné počítačové architektuře.

V další kapitole si stroj poprvé prohlédneme jako celek — a hned na něm spustíme první program.

KAPITOLA 3

Seznámení se SAP-3/16

Teorii máme za sebou — pojďme si stroj konečně prohlédnout a hned na něm něco spustit. Tahle kapitola je taková obhlídka: nakoukneme na všechny hlavní části, představíme si je jménem a napíšeme první skutečný program. Detaily každé části přijdou v dalších kapitolách; teď jde o to udělat si celkovou mapu, abys věděl, kam co patří.

Technický list

Kdyby SAP-3/16 měl krabici jako opravdový čip, na boku by stálo tohle:

Vlastnost	Hodnota
Šířka dat	8 bitů (jeden bajt, hodnoty 0–255)
Šířka adres	16 bitů (adresní prostor 64 KB, `0x0000` – `0xFFFF`)
Pracovní registry	A (akumulátor), B, X, Y — každý 8bitový
Adresové registry	PC (čítač instrukcí), SP (ukazatel zásobníku) — 16bitové
Příznaky	C (carry), Z (zero), N (negative), V (overflow), I (přerušení)
Instrukční sada	systematické kódování, 8 adresových režimů, 1–3 bajty na instrukci
Zásobník	v hlavní RAM, roste dolů od `0x0200`
Přerušení	časovač a klávesnice, vektory na vrcholu paměti
Zařízení	terminál, klávesnice, gamepad, zvukový čip (3 hlasy), obrazovka, disk
Hodiny	nastavitelné od jednotek Hz po 1 MHz

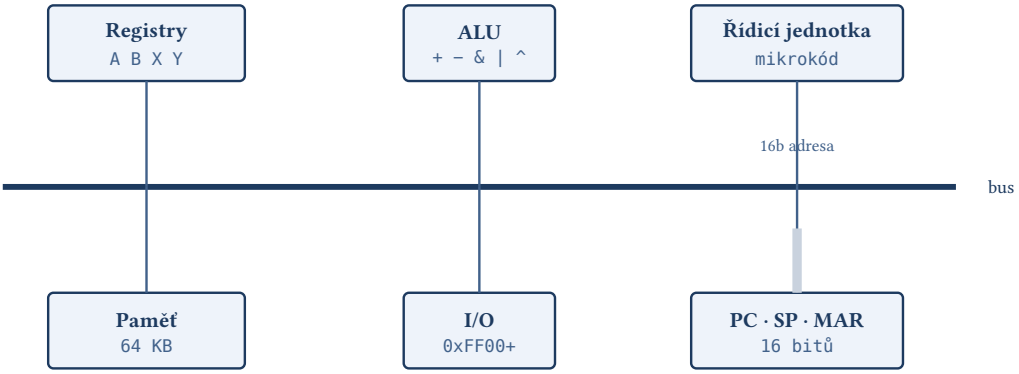
Tabulka 2. Technický list SAP-3/16 — celý počítač na jednom řádku každý.

Pár čísel z té tabulky si zaslouží zdůraznit. **Osm bitů dat, šestnáct bitů adres** už známe z kapitoly 1 a bude se to táhnout celou knihou. **Čtyři pracovní registry** jsou vším, co má procesor „po ruce“ — všechno ostatní musí do paměti a zpět. A **nastavitelné hodiny** jsou tvůj nejlepší učební nástroj: zpomal stroj na pár taktů za sekundu a uvidíš každý bajt, jak putuje po drátech.

Co uvidíš, když otevřeš víko

Emulátor kreslí procesor jako schéma a stojí za to znát jeho hlavní bloky. Všechny se v dalších kapitolách rozebírají dopodrobna; tady je jen představíme:

- **Registry** — hrstka osmi- a šestnáctibitových buněk uvnitř procesoru, se kterými instrukce pracují přímo (kapitola 4).
- **Aritmeticko-logická jednotka (ALU)** — část, která doopravdy počítá: sčítá, odčítá, porovnává, provádí logiku a posuny (kapitola 6).
- **Datová sběrnice** — osm sdílených drátů, po kterých bloky posílají bajty jeden druhému. V každém okamžiku smí „mluvit“ jen jeden (kapitola 5).
- **Paměť** — plochých 64 KB, kde žije program i data (kapitola 14).
- **Řídicí jednotka** — dispečer, který v každém taktu rozhodne, které dráty se rozsvítí. Je to mozek celé choreografie a potkáme ho v kapitole 7.



Obrázek 1. Datová cesta SAP-3/16: jedna 8bitová sběrnice spojuje všechny bloky; v každém taktu na ni „mluví„ jediný z nich. Šestnáctibitové adresy jdou po vyhrazené široké cestě.

Až budeš stroj krokovat, uvidíš, jak se mezi těmito bloky přelévají data a jak řídicí jednotka takt po taktu diriguje provoz. Nejlíp se to pochopí právě pohledem — text popisuje, co se děje, ale skutečné „aha“ přijde, když to uvidíš hýbat se.

První program

Dost řečí, pojďme něco spustit. Nejjednodušší užitečná věc, kterou SAP-3/16 umí, je napsat text na terminál. Vzpomeň si z kapitoly 1, že „text“ jsou jen bajty s ASCII významem — a terminál je zařízení, které libovolný bajt zapsaný na adresu `0xFF10` promění v písmeno na obrazovce.

Program tedy jen postupně nasype ASCII kódy pozdravu do té jediné adresy:

```
; První program: pozdrav na terminálu.
; Každý bajt zapsaný do TERM_DATA (0xFF10) se objeví na obrazovce.
TERM_DATA EQU 0xFF10

main:
    LDA #'A'        ; ASCII kód písmene A (65) do akumulátoru
```

```
    STA TERM_DATA   ; zapiš ho na terminál -> objeví se "A"
    LDA #'h'
    STA TERM_DATA
    LDA #'o'
    STA TERM_DATA
    LDA #'j'
    STA TERM_DATA
    LDA #'!'
    STA TERM_DATA
    LDA #10         ; kód 10 = nový řádek
    STA TERM_DATA
    HLT             ; zastav procesor
```

A jeho výstup:

```
Ahoj!
```

Program běží 91 taktů hodin a zabírá 31 bajtů paměti. Rozeberme si ho, protože se v něm skrývají skoro všechny základní myšlenky, které budeme dál rozvíjet:

- `TERM_DATA EQU 0xFF10` je **konstanta**: říká překladači, že jméno `TERM_DATA` znamená adresu `0xFF10`. Kód se tím čte líp než s holým číslem.
- `main:` je **label** — pojmenování místa v programu. Sem se dostane procesor po startu.
- `LDA #'A'` znamená *load accumulator* — nahraj do registru A hodnotu. Znak `#` je důležitý: říká „ber to jako přímou hodnotu“ (číslo 65), ne jako adresu. K tomuhle rozdílu se ještě vrátíme, je to častý zdroj omylů.
- `STA TERM_DATA` znamená *store accumulator* — zapiš obsah A na danou adresu. A protože je na té adrese připojený terminál, písmeno se objeví na obrazovce.
- `HLT` procesor zastaví. Bez něj by běžel dál do prázdné paměti a spadl do trapu (kapitola 2).

> **Tip**
>
> Všimni si, že mezi dvěma stejnými písmeny nemusíš `LDA` opakovat — akumulátor si hodnotu drží, dokud ji nepřepíšeš. Kdybys tiskl „lll“, stačí jednou `LDA #'l'` a pak třikrát `STA TERM_DATA`.

To je celý první program — pět písmen, nový řádek a stop. Vypadá triviálně, ale právě jsi viděl uložený program, immediate hodnoty, paměťově mapované I/O i zastavení

procesoru. V další části otevřeme víko doopravdy a projdeme procesor registr po registru.

ČÁST II

Uvnitř procesoru

KAPITOLA 4

Registry — pracovní stůl procesoru

Paměť je velká, ale „daleko„. Každý přístup do ní je malá transakce po sběrnici, která stojí čas. Kdyby procesor musel sáhnout do paměti pro každé jednotlivé číslo, se kterým počítá, byl by beznadějně pomalý. Proto má přímo v sobě hrstku pracovních buněk — **registrů** — se kterými instrukce pracují okamžitě.

Nejlepší přirovnání: paměť je **knihovna** a registry jsou **stůl**, u kterého právě sedíš. Na stůl si položíš pár knih, se kterými zrovna pracuješ; zbytek knihovny je k dispozici, ale musíš pro něj vstát. Celé programování osmibitového stroje je z velké části umění chytře hospodařit s tím malým stolem.

Registry viditelné programátorovi

SAP-3/16 má čtyři osmibitové pracovní registry a dva šestnáctibitové adresové. S těmi budeš pracovat přímo:

- **A (akumulátor)** — hlavní pracovní registr. Skoro každá operace ALU bere A jako vstup a výsledek nechává zase v A. Jméno je historické: první počítače v něm „akumulovaly„ průběžné součty.
- **B** — registr druhého operandu ALU. Když sčítáš, odčítáš nebo porovnáváš, druhé číslo jde do B. Má to jeden zrádný důsledek, o kterém za chvíli.
- **X, Y** — indexové registry. Jejich hlavní úloha je **indexované adresování**: `LDA tabulka,X` čte z adresy `tabulka + X`. Smyčka tak projde celé pole prostým zvyšováním X (`INX`) místo přepisování adres. Jsou to počítadla a ukazovátka do dat.
- **PC (Program Counter, čítač instrukcí)** — šestnáctibitový registr s adresou *příští* instrukce. Skok není nic jiného než přepsání PC. Po startu se PC načte z takzvaného RESET vektoru (kapitola 14).
- **SP (Stack Pointer, ukazatel zásobníku)** — šestnáctibitový registr ukazující na vrchol zásobníku v hlavní RAM. Startuje na `0x0200` a roste dolů (kapitola 11).

> **Pozor**
>
> **Registr B se přepisuje sám.** Mikrokód nahraje do B druhý operand každé binární ALU instrukce (ADD, SUB, CMP a dalších) těsně předtím, než ALU vypálí. To znamená, že **každá taková instrukce B přepíše**. Můžeš do B vědomě něco vložit přes `TAB` (třeba pro násobení `MUL`) a přečíst zpět přes `TBA`, ale nečekej,

že tam hodnota přežije jediné `ADD`. Zapomenutí na tohle je jeden z nejčastějších omylů začátečníků na tomhle stroji.

Interní registry: viditelné ve schématu, ne z programu

Procesor má ještě několik buněk, které z programu neovládáš, ale ve schematickém pohledu je uvidíš pracovat. Je dobré je znát jménem, protože v dalších kapitolách se na ně budeme odvolávat:

- **IR (Instruction Register)** — drží právě vykonávaný opcode. Dekodér z jeho bitů vyrábí řídicí signály.
- **OP a OPH (operandové registry)** — podrží druhý a třetí bajt vícebajtové instrukce po dobu jejího vykonávání. Šestnáctibitová adresa přijde jako dva bajty: dolní do OP, horní do OPH.
- **MAR (Memory Address Register)** — šestnáctibitová „adresová záklopka„. Adresu, kterou má paměť právě číst nebo zapisovat, procesor nejdřív uloží sem.
- **TMP a addrCarry** — úplně skrytá dvojice. `TMP` je pomocný registr, kterým indexované a nepřímé adresování počítají efektivní adresy, *aniž by přepsaly* jakýkoli viditelný registr. `addrCarry` je k tomu jednobitová záklopka přenosu, díky které se šestnáctibitová adresa spočítá po bajtech správně. K programu se nedostaneš ani k jednomu — ale právě díky nim zůstávají tvoje A, X a Y při výpočtu adres nedotčené. Skutečné procesory jsou takových neviditelných záklopek plné; 6502 je má také.

> **Poznámka**
>
> Není náhoda, že „viditelných„ registrů je málo a „skrytých“ hodně. Návrhář procesoru dává ven jen to, co programátor potřebuje ovládat, a zbytek schová. Ta skrytá mašinérie ale existuje a v kapitole 7 ji uvidíme pracovat takt po taktu.

Příznaky

Vedle datových registrů sedí **registr příznaků** — pět jednobitových faktů o tom, jak dopadl poslední výpočet:

- **C** (carry) — přenos nebo výpůjčka,
- **Z** (zero) — výsledek byl přesně nula,
- **N** (negative) — výsledek vypadá jako záporný (kopie bitu 7),
- **V** (overflow) — znaménkové přetečení,
- **I** (interrupt) — jsou povolena přerušení?

Příznaky jsou krátkodobá paměť procesoru pro rozhodování. Každý podmíněný skok — „skoč, jen když byl výsledek nula„ — čte právě je. Jsou tak malé (jeden bit!), a přitom bez nich by se program nedokázal rozhodnout vůbec. Celou kapitolu 6 věnujeme tomu, kdy přesně se který příznak mění, a v kapitole 10 uvidíme, jak se podle nich větví.

Teď víme, *co* procesor má. V další kapitole se podíváme, *jak* jsou ty části propojené — a proč je jediná osmibitová sběrnice zdrojem tolika zajímavých kompromisů.

KAPITOLA 5

Sběrnice — 8bitová cesta a 16bitová zkratka

Máme registry, ALU, paměť a zařízení. Jak jsou propojené? Skoro všechno spojuje jediná **osmibitová datová sběrnice**: osm sdílených drátů, na které může kdokoli položit bajt a ze kterých může kdokoli číst. Je to hlavní třída celého města — všechen provoz jede po ní.

Pravidlo „jen jeden mluví„

Sdílené dráty mají jedno železné pravidlo: v každém okamžiku smí na sběrnici **vystavit** hodnotu jen jediný blok. Kdyby dva registry naráz hnaly na tytéž dráty různé bajty, výsledkem by byl elektrický zmatek — ani jedna hodnota, jen zkrat a nesmysl.

Proto má každý registr dva řídicí vstupy: signál **out** („vystav svou hodnotu na sběrnici„) a signál **in** („zachyť, co je právě na sběrnici“). Přenos dat je pak úplně prostý: zvedneš jeden *out* a jeden *in* ve stejný okamžik. Zápis `A0 BI` znamená „A vystavuje na sběrnici (*A out*), B zachytává (*B in*)„ — tedy zkopíruj A do B. Celý procesor je v jádru tahle jedna choreografie: v každém taktu jeden mluvčí a libovolně mnoho posluchačů.

> **Poznámka**
>
> Signály `A0`, `BI`, `RO` a spol. jsou **řídicí signály** — nejnižší úroveň ovládání procesoru. Ve schematickém pohledu emulátoru je uvidíš jako rozsvícené šipky. Kompletní slovníček najdeš v kapitole 7 a v příloze C; teď stačí vědět, že přenos = jeden *out* + jeden *in*.

Problém šestnácti bitů

Jenže tenhle stroj má **šestnáctibitové adresy** a šestnáctibitová hodnota se na osmibitovou sběrnici na jednu cestu prostě nevejde. Co s tím? Odpovědi jsou dvě a SAP-3/16 chytře používá obě.

Za prvé: přenášet adresy po bajtech. Většina instrukcí to dělá právě takhle — absolutní přístup pošle do MAR nejdřív dolní bajt adresy, pak horní, ve dvou po sobě jdoucích krocích. Stojí to takty navíc a — což je na tomhle stroji krásně vidět —

přesně proto je šestnáctibitové adresování na osmibitovém procesoru *dražší* než práce s jedním bajtem. Ten rozdíl v ceně budeš vídat pořád.

Za druhé: vyhrazená šestnáctibitová zkratka. Tři přenosy se dějí neustále a byla by škoda platit za ně vždycky dva takty, takže pro ně stroj má samostatnou širokou cestu, která přenese celou adresu najednou:

- `PCM` — PC ⟶ MAR, při každém načtení instrukce („kde je příští instrukce?„),
- `SPM` — SP ⟶ MAR, při každém přístupu na zásobník,
- `JPW` — operand ⟶ PC, při absolutním skoku („nastav čítač na cílovou adresu„).

Ve schématu je tahle zkratka nakreslená jako zvlášť silná čára vedle osmibitové sběrnice. Je to pěkná ukázka inženýrského kompromisu: obecná cesta je úzká a levná, ale pro pár nejčastějších šestnáctibitových přenosů se vyplatí přidat širokou linku navíc.

Sleduj to při krokování

Nejlíp celou myšlenku pochopíš, když stroj zpomalíš a krokuješ ho v režimu jednotlivých taktů. V každém mikrokroku uvidíš přesně jednoho mluvčího a jeho posluchače na osmibitové cestě, tu a tam doplněné o širokou adresovou zkratku. Celý počítač je choreografie na těchhle sdílených drátech — a řídicí jednotka, kterou potkáme v kapitole 7, je dispečer, který provoz řídí.

V další kapitole se podíváme na blok, který dává sběrnici smysl tím, že s přenášenými čísly konečně něco *dělá* — na aritmeticko-logickou jednotku.

KAPITOLA 6

ALU a příznaky

Registry drží čísla a sběrnice je vozí sem a tam — ale někde se s nimi musí konečně *počítat*. Tou částí je **aritmeticko-logická jednotka**, zkráceně **ALU**. Je to počtář procesoru: vezme akumulátor A a druhý operand (registr B, nebo u immediate režimu rovnou bajt operandu), provede jednu operaci a vydá osmibitový výsledek plus bity příznaků.

První součet

Nejjednodušší operace je sčítání. Vezmi A, přičti k němu číslo, výsledek zůstane v A:

```
; Sečti dvě čísla a ukaž výsledek na výstupním registru.
main:
    LDA #5      ; A = 5
    ADD #3      ; A = A + 3 = 8  (operand 3 jde do B, ALU sečte)
    OUT         ; zobraz A na výstupu
    HLT
```

Výstupní registr po doběhnutí drží `OUT: 8`. Nenápadné, ale právě jsi viděl ALU při práci: `ADD #3` poslalo trojku do registru B, ALU sečetla A + B a výsledek 8 vrátila do A.

Katalog operací

ALU v SAP-3/16 umí tohle:

- **Aritmetika**: `ADD`, `ADC` (sčítání s přenosem), `SUB`, `SBC` (odčítání s výpůjčkou), `INC`, `DEC` (o jedničku nahoru/dolů), `CMP` (porovnání — odečti a výsledek zahoď, zůstanou jen příznaky), `MUL` (násobení), `DIV` (dělení).
- **Logika**: `AND`, `OR`, `XOR`, `NOT` — bit po bitu.
- **Posuny a rotace**: `SHL`, `SHR` (posun vlevo/vpravo, vysunutý bit padá do carry), `ROL`, `ROR` (rotace *přes* carry — bit vypadlý z jednoho konce se vrátí z příznaku C na druhém).
- **Speciál**: `RND` — pseudonáhodný bajt. Generátor lze nastartovat pevným semínkem (*seed*), takže běhy programu mohou být přesně reprodukovatelné.

Počítání přes hranici bajtu

Bajt uloží nanejvýš číslo 255. Jak tedy sečíst dvě šestnáctibitová čísla? Rozdělíš je na dolní a horní bajt, dolní sečteš přes `ADD` a horní přes `ADC` (*add with carry*) — a přenos se mezi nimi propaguje sám, přes příznak C. Přesně takhle osmibitový stroj počítá s velkými čísly:

```
; 16bitový součet: 0x01FF + 0x0001 = 0x0200, uloženo na 0x40/0x41
(LE).
main:
    LDA #0xFF       ; dolní bajt prvního čísla
    ADD #0x01       ; + dolní bajt druhého -> 0x00, carry = 1
    STA 0x40        ; ulož dolní bajt výsledku
    LDA #0x01       ; horní bajt prvního čísla
    ADC #0x00       ; + horní bajt druhého + carry -> 0x02
    STA 0x41        ; ulož horní bajt výsledku
    HLT
```

Po doběhnutí je v paměti `0x40 = 0x00` a `0x41 = 0x02`, tedy dohromady (little-endian) `0x0200` — správný součet. `ADD` dolních bajtů `0xFF + 0x01` přeteklo na `0x00` a nastavilo carry; `ADC` horních bajtů pak ten přenos připočetlo. Stejným postupem sečteš čísla o libovolné šířce — jen přidáváš další `ADC`.

Co příznaky znamenají

Každá operace ALU kromě výsledku nastavuje i příznaky. Jsou to jednobitové poznámky o tom, jak výpočet dopadl:

- **Z (Zero)** — výsledek byl přesně 0. Pracant mezi příznaky: počítadla smyček, testy rovnosti.
- **C (Carry)** — po `ADD` / `ADC` znamená „skutečný výsledek přesáhl 255″; po `SUB` / `SBC` / `CMP` funguje jako **výpůjčka** (borrow): nastavené C znamená, že A bylo (bezznaménkově) *menší* než operand.
- **N (Negative)** — kopie bitu 7 výsledku. Dává smysl, když bajt čteš jako dvojkový doplněk.
- **V (Overflow)** — *znaménkový* výsledek vypadl z rozsahu −128 až +127 (třeba 127 + 1 = „−128″). Mění ho jen `ADD` / `ADC` / `SUB` / `SBC`; logika a posuny ne.

Kdo přesně nastavuje které příznaky

Tohle je jedna z nejdůležitějších tabulek v knize. Pravidla jsou schválně konzistentní a vyplatí se je znát — hodně chyb v assembleru pramení z chybného očekávání, který příznak se zrovna změnil (nebo nezměnil):

Skupina instrukcí	Které příznaky mění
`ADD` / `ADC` / `SUB` / `SBC`	C, Z, N, V
`CMP`	C, Z, N (V se nedotkne)
Logika (`AND` / `OR` / `XOR` / `NOT`) a `BIT`	jen Z, N — C ani V se nemění (`BIT` navíc nechá A)
Posuny a rotace	Z, N; do C padá vysunutý bit
`INC` / `DEC`	jen Z, N — carry zůstává!
Loady a přesuny (`LDA` , `TAX` , …)	Z, N podle přesunuté hodnoty
Zápisy do paměti, push, `INW` / `DEW` , `ADW` / `SBW` , `LXY` / `SXY`	*žádné* příznaky

Tabulka 3. Kdo sahá na příznaky. Nenápadné výjimky (INC/DEC nechává C, ukazatelové INW/DEW nemění nic) jsou schválně — díky nim jde skládat kód, aniž bys rozbil rozdělaný výpočet.

Dvě řádky si zaslouží vysvětlení, protože jsou to dárky pro programátora:

- **Loady nastavují Z a N.** `LDA counter` ti rovnou řekne, jestli je počítadlo nula — žádné `CMP` navíc není potřeba. To je základ úsporných smyček.
- **`INC` / `DEC` nechává C, a `INW` / `DEW` nemění vůbec nic.** Můžeš tak posunout šestnáctibitový ukazatel uprostřed vícebajtové aritmetiky, aniž bys zničil přenos, který zrovna přenášíš. Návrh ALU tuhle „bezpříznakovost„ hlídá schválně.

Teď víme, co procesor umí spočítat a jak si o výsledku dělá poznámky. Zbývá poslední velká otázka: *jak* se jedna instrukce vlastně vykoná — takt po taktu, drát po drátu. To je téma další kapitoly.

KAPITOLA 7

Instrukční cyklus — fetch, decode, execute

Procesor není chytrý. Neplánuje, nepřemýšlí dopředu — opakuje pořád dokola jednu tupou smyčku. V moderních čipech miliardykrát za sekundu, v tomhle emulátoru od jednotek hertzů po pár kilohertzů. Ta smyčka má tři kroky:

1. **Fetch (načti)** — přečti bajt na adrese PC z paměti do registru IR.
2. **Decode (dekóduj)** — řídicí jednotka se podívá na opcode a rozhodne, které mikrokroky spustit.
3. **Execute (vykonej)** — proveď je: přesuň bajty, spusť ALU, zapiš do paměti.

A pak zase od začátku — PC už mezitím ukazuje na další instrukci. To je celý počítač. Všechno ostatní je jen otázka, *co* přesně se v kroku 3 stane.

Bity opcode ti řeknou, co dělat

Právě krok „decode„ dělá ze SAP-3/16 skvělou učební pomůcku. U tohohle stroje totiž **bity opcode přímo prozrazují, co instrukce dělá.** Osmibitový opcode je rozdělený na tři pole:

```
bit  7 6 | 5 4 3 | 2 1 0
     g g | o o o | m m m
     ────┼───────┼─────
    skupina operace  režim
```

- Horní dva bity (`gg`) volí **skupinu**: `00` implied (bez operandu), `01` load/store, `10` ALU, `11` řízení toku.
- Prostřední tři bity (`ooo`) volí **operaci** uvnitř skupiny.
- Dolní tři bity (`mmm`) volí **adresový režim** (jak najít operand — o tom kapitola 8).

Vezmi opcode `0x93` . Binárně je to `10|010|011` : skupina `10` (ALU), operace `010` (SUB), režim `011` (absolutně indexováno přes X). `0x93` tedy znamená „odečti od A hodnotu z adresy `základ + X` „. A to všechno *přečteš z bitů z hlavy*, bez tabulky. Právě tahle pravidelnost je důvod, proč se stroj tak dobře učí — a v kapitole 21 uvidíme, že přesně ona umožňuje moderním procesorům instrukce zřetězit.

> **Poznámka**
>
> Ne každá kombinace bitů je platná instrukce (některé režimy pro některé operace nedávají smysl). Nedefinovaný opcode procesor nevykoná — spadne do trapu a řekne ti, kde. Kompletní mapu opkódů najdeš v příloze A.

T-stavy a mikrokód

Každá instrukce se dělí na takty hodin zvané **T-stavy**. V každém T-stavu řídicí jednotka zvedne konkrétní kombinaci **řídicích signálů** — těch přepínačů *out*/*in* z kapitoly 5 plus ovládání ALU a zásobníku. Tabulce „které signály, ve kterém T-stavu, pro který opcode„ se říká **mikrokód**: je to nejvnitřnější program procesoru, o úroveň níž než assembler.

Délka načítání kopíruje délku instrukce — 2 T-stavy na každý bajt, tedy 2, 4 nebo 6 T-stavů fetche — a vykonání trvá přesně tolik kroků, kolik práce doopravdy vyžaduje. **Žádné doplňování naprázdno:** `NOP` trvá 3 T-stavy, indexované `ADD` dvanáct, a emulátor ukazuje ty poctivé, proměnlivé počty.

Jedna instrukce takt po taktu

Podívejme se na kompletní mikrokód instrukce `LDA 0x50` — „načti A z adresy `0x50` v nulté stránce„. Je to dvoubajtová instrukce (opcode + jeden bajt adresy) a zabere šest T-stavů:

```
T0: PCM        PC → MAR              (kde je opcode?)
T1: RO II CE   RAM[MAR] → IR, PC++   (načti opcode do IR)
T2: PCM        PC → MAR              (kde je operand?)
T3: RO ORI CE  RAM[MAR] → OP, PC++   (načti bajt operandu do OP)
T4: ORO MI     OP → MAR              (0x50 se stává adresou
0x0050)
T5: RO AI FI   RAM[0x0050] → A       (a nastav příznaky Z/N)
```

Rozeber si to: T0–T3 je **fetch** (u každé dvoubajtové instrukce vypadá stejně — načti opcode, načti operand), T4–T5 je **execute** (udělej z operandu adresu a přečti z ní do A). Signál `FI` na konci zachytí příznaky, takže `LDA` rovnou nastaví Z a N podle načtené hodnoty (vzpomeň si na tabulku z kapitoly 6).

A co *absolutní* `LDA 0x1234`? Ta potřebuje o dva T-stavy víc: třetí dvojici fetche pro horní bajt adresy a jeden krok navíc na složení šestnáctibitové adresy v MAR. To je ta daň osmibitové sběrnice z kapitoly 5 — teď ji vidíš vyčíslenou v taktech.

> **Tip**
>
> Přepni emulátor do režimu **µStep** („mikrokrok„) a můžeš tyhle T-stavy krokovat po jednom. Ve schématu přitom sleduješ, jak se jednotlivé řídicí signály rozsvěcují a jak bajt putuje z paměti přes sběrnici do registru A. Je to nejlepší způsob, jak si celý cyklus opravdu „osahat“.

Slovníček signálů (výběr)

Nemusíš se řídicí signály učit nazpaměť — pointa je, že *každá* instrukce, jakkoli abstraktně v assembleru vypadá, končí v malém slovníku akcí na úrovni drátů. Pár nejčastějších:

- `PCM` / `SPM` — MAR := PC / MAR := SP (šestnáctibitová adresová zkratka),
- `CE` — inkrementuj PC,
- `RO` / `RI` — čti z paměti / zapiš do paměti (na adresu v MAR),
- `II` — nahraj opcode do IR,
- `ORI` / `ORO` — operandový registr dovnitř / ven,
- `AI` / `AO` (a `BI` / `BO`, `XI` / `XO`, …) — registr dovnitř / ven,
- `MI` — nahraj adresu nulté stránky do MAR,
- `EO` — vystav výsledek ALU, `FI` — zachyť příznaky,
- `JPW` — absolutní skok (PC := adresa),
- `HLT` — zastav hodiny.

Kompletní seznam všech signálů najdeš v příloze C. Tím máme hotový celý procesor: víme, co obsahuje, jak je to propojené a jak se jedna instrukce vykoná. V další části se naučíme stroji poroučet — jazykem assembler.

ČÁST III

Řeč stroje: assembler

KAPITOLA 8

Kódování instrukcí a adresové režimy

V kapitole 7 jsme viděli, že opcode se skládá ze tří polí — skupina, operace, režim. To poslední pole, **adresový režim**, si teď zaslouží celou kapitolu, protože rozhoduje o jedné klíčové věci: **kde instrukce najde svůj operand**. Táž operace „načti A„ existuje v osmi různých režimech a umět je rozeznat je půlka plynulého čtení assembleru.

Osm způsobů, jak najít operand

Dolní tři bity opcode (`mmm`) kódují jeden z osmi režimů. Tady jsou všechny, s příkladem na instrukci `LDA` („načti do A„):

Režim	Zápis	Kde je operand
immediate	`LDA #5`	operand *je* ta hodnota (číslo 5)
nultá stránka	`LDA 0x42`	hodnota na adrese `0x0042` (jednobajtová adresa)
absolutní	`LDA 0x1234`	hodnota na plné 16bitové adrese
absolutní,X	`LDA tab,X`	hodnota na adrese `tab + X`
absolutní,Y	`LDA tab,Y`	hodnota na adrese `tab + Y`
nepřímý	`LDA (ptr)`	dva bajty na `ptr` *jsou* adresa; čte se odtamtud
nultá stránka,X	`LDA slot,X`	`slot + X` , ale wrapuje uvnitř nulté stránky
nepřímý,Y	`LDA (ptr),Y`	přečti ukazatel na `ptr` , pak přičti Y

Tabulka 4. Osm adresových režimů SAP-3/16. Stejná operace, osm způsobů, jak se dostat k datům.

Projděme si ty důležité:

- **Immediate** (`#`) — operand je přímo hodnota. `LDA #5` dá do A pětku.
- **Nultá stránka** — jednobajtová adresa do prvních 256 bajtů paměti. Kratší a o tři takty rychlejší než absolutní (proč, uvidíme v kapitole 14).
- **Absolutní** — plná šestnáctibitová adresa kdekoli v paměti.
- **Indexované** (`,X` / `,Y`) — k adrese se přičte obsah indexového registru. To je motor práce s poli: smyčka projde tabulku prostým zvyšováním X.
- **Nepřímé** (`(ptr)`) — dva bajty v nulté stránce *jsou samy adresou* a čte se z místa, kam ukazují. To je assemblerová podoba ukazatele.

- **Nepřímé indexované** (`(ptr),Y`) — přečti ukazatel a pak přičti Y. Klasický idiom pro procházení bufferu, jehož adresu znáš až za běhu.

> **Pozor**
>
> Znak `#` u immediate režimu je **povinný a mění význam!** `LDA #42` načte hodnotu 42. `LDA 42` načte to, co leží na adrese 42 v nulté stránce. Vypadají skoro stejně, dělají něco úplně jiného — a záměna je jeden z nejčastějších začátečnických omylů.

Indexování v akci

Nejlíp adresové režimy pochopíš na příkladu. Tenhle program vypíše řetězec znak po znaku pomocí indexovaného adresování `LDA msg,X`. X začíná na nule a v každém průchodu smyčky se zvýší, takže `msg,X` postupně ukazuje na `msg+0`, `msg+1`, `msg+2` a tak dál — projde celé pole:

```
; Výpis řetězce indexovaným adresováním: LDA msg,X projde pole po
znaku.
TERM_DATA EQU 0xFF10

main:
    LDX #0          ; index = 0
.loop:
    LDA msg,X       ; načti znak na pozici msg + X (nastaví Z,
když je 0)
    JEQ .done       ; nula = konec řetězce
    STA TERM_DATA   ; vypiš znak
    INX             ; posuň index
    JMP .loop
.done:
    HLT

msg: DATA "Indexy!", 10, 0    ; text, nový řádek, nulový
zakončovač
```

Výstup:

```
Indexy!
```

Všimni si, jak elegantně to drží pohromadě. Konec řetězce hlídá **nulový zakončovač** (`0` na konci dat) — stejná konvence jako v jazyce C. A protože `LDA` nastavuje příznak

Z (kapitola 6), stačí hned po načtení `JEQ .done` a žádné zvláštní porovnání s nulou není potřeba. Tenhle malý program v sobě skrývá indexování, smyčku, práci s daty i chytré využití příznaků.

Automatická volba šířky

Když napíšeš `LDA neco`, jak assembler pozná, jestli má použít krátký režim nulté stránky, nebo dlouhý absolutní? **Odhadne to podle hodnoty adresy.** Pokud se vejde do jednoho bajtu (≤ `0xFF`), zvolí nultou stránku; jinak absolutní. Když chceš rozhodnout sám, přípony to vynutí: `LDA.B` si vynutí nultou stránku, `LDA.W` absolutní. Detaily téhle „sizing„ mašinérie necháme na kapitolu 9 — teď stačí vědět, že se o šířku obvykle nemusíš starat, assembler to zařídí za tebe.

Teď umíme najít operand osmi způsoby. V další kapitole si pořádně projdeme celý jazyk assembler — syntax, čísla, direktivy — abychom mohli psát pořádné programy.

KAPITOLA 9

Jazyk assembler

Strojový kód jsou bajty. **Assembler** je jejich lidsky čitelný zápis a program, který ten zápis překládá na bajty, se jmenuje také assembler (překladač). Mapování je v podstatě jedna ku jedné — každý řádek s mnemonikou se stane jednou strojovou instrukcí — a právě proto tě assembler naučí, co stroj doopravdy dělá. Nic se neschovává za vrstvy abstrakce; píšeš skoro přímo bajty, jen srozumitelně.

Anatomie řádku

Řádek assembleru SAP-3/16 může obsahovat čtyři druhy věcí:

- **Instrukci** — mnemoniku a volitelný operand: `LDA #42`, `ADD 0x30`, `JMP LOOP`.
- **Label** — symbolické jméno adresy zakončené dvojtečkou: `LOOP:`. Assembler nahradí každé použití `LOOP` skutečnou adresou, takže nikdy nepočítáš bajty ručně.
- **Direktivu** — příkaz pro překladač, ne pro procesor: `.ORG`, `DATA`, `EQU` a další (za chvíli).
- **Komentář** — cokoli za středníkem `;`.

Label i instrukce můžou stát na jednom řádku (`LOOP: LDA #0`) nebo zvlášť. Prázdné řádky a odsazení překladač ignoruje — piš tak, aby se to dobře četlo.

Formáty čísel

Číslo můžeš zapsat několika způsoby a hodí se je znát všechny:

- `42` — desítkově,
- `0x2A` — šestnáctkově,
- `0b00101010` — binárně,
- `'A'` — ASCII znak (překladač dosadí 65),
- `-1` — záporný literál; sestaví se jako dvojkový doplněk, takže `LDA #-1` načte `0xFF`.

Znak `#` a výběr bajtů

Už víme, že `#` znamená immediate (přímou hodnotu). U šestnáctibitových adres se hodí ještě dva příbuzní: `#<` vezme **dolní** bajt a `#>` **horní** bajt symbolu. To je nezbytné, když chceš nahrát adresu do osmibitového registru nebo sestavit ukazatel v nulté stránce:

```
LDA #<msg       ; dolní bajt adresy msg
STA 0x10        ; do nulté stránky
LDA #>msg       ; horní bajt adresy msg
STA 0x11        ; sousední bajt -> spolu tvoří 16bitový ukazatel na
0x10
```

Tenhle vzor — „rozlož adresu na dva bajty a slož ukazatel„ — potkáš při práci s ukazateli pořád (kapitola 14).

Direktivy

Direktivy neprodukují instrukce; řídí, jak překladač skládá výsledný obraz paměti:

Direktiva	Co dělá
`.ORG 0x0300`	pokračuj v sestavování od adresy `0x0300`
`DATA 42, "text", 0`	vlož sem bajty (čísla i řetězce)
`.WORD label`	vlož 16bitovou little-endian adresu (alias `DW`)
`.VECTOR IRQ, isr`	zapiš vektor přerušení (RESET / IRQ / BRK)
`NAME EQU 0xFF10`	definuj konstantu (žádné dvojtečky)

Tabulka 5. Direktivy assembleru SAP-3/16.

`EQU` je tvůj nástroj na pojmenování magických čísel. Místo `STA 0xFF10` roztroušeného po celém programu napíšeš jednou `TERM_DATA EQU 0xFF10` a pak všude `STA TERM_DATA`. Kód se čte líp a případnou změnu adresy uděláš na jednom místě.

`DATA` ukládá bajty rovnou tam, kde stojí — čísla, ASCII znaky i celé řetězce v uvozovkách (uvnitř fungují escapy jako `\n` pro nový řádek). `.WORD` uloží šestnáctibitovou hodnotu ve správném pořadí bajtů (dolní, pak horní), což je přesně to, co potřebuješ pro tabulky adres.

Lokální labely

Když má program deset smyček, nechceš pro každou vymýšlet unikátní jméno jako `loop1`, `loop2`. Proto existují **lokální labely** — začínají tečkou (`.loop`) a platí jen v rámci předchozího „globálního„ labelu. Různé rutiny tak můžou používat stejná krátká jména bez konfliktu:

```
tiskni:
.loop:              ; toto je vlastně "tiskni.loop"
    LDA (ptr),Y
    JEQ .konec
```

```
    INY
    JMP .loop
.konec:
    RET

kopiruj:
.loop:              ; jiný ".loop" - patří k "kopiruj", nekoliduje
    ...
```

Co dostaneš na výstupu

Kromě samotných bajtů ti překladač vygeneruje **tabulku symbolů** (které jméno je na které adrese) a **listing** (řádek zdroje vedle bajtů, které z něj vznikly). Obojí je k nezaplacení při ladění — uvidíš přesně, kam se co uložilo. A pokud v kódu uděláš chybu, assembler tě zastaví se srozumitelnou hláškou a číslem řádku, ne se záhadným pádem.

> **Poznámka**
>
> Assembler tě nepustí umístit kód nebo data do oblasti `0xFF00` – `0xFF7F` — ty adresy patří hardwarovým zařízením (kapitola 16). Když to zkusíš, dostaneš jasnou chybu, ne tajemně nefungující program.

Teď umíme napsat čistý, čitelný program. Zbývá se naučit, jak se stroj **rozhoduje** — a to je téma další kapitoly o větvení.

KAPITOLA 10

Větvení a rozhodování

Program, který jede jen rovně shora dolů, umí málo. Síla přichází s **rozhodováním** — schopností podle výsledku výpočtu skočit sem, nebo tam. Na téhle jediné myšlence stojí každá smyčka, každá podmínka, každá hra. A základním nástrojem je **podmíněný skok**: instrukce, která načte PC (skočí) jen tehdy, když platí určitý příznak.

Podmíněné skoky

Vzpomeň si z kapitoly 6 na příznaky Z, C, N, V. Podmíněné skoky se na ně dívají:

- **JZ / JNZ** — skoč, když je Zero nastavené / smazané. Aliasy **JEQ / JNE** (rovnost, konec smyčky).
- **JC** — skoč při nastaveném Carry. Po `CMP` / `SUB` je carry výpůjčka, takže `JC` znamená „A bylo (bezznaménkově) menší než operand„. Alias **JLT**.
- **JNC** — skoč při smazaném Carry: „A ≥ operand„ (bezznaménkově). Alias **JGE**.
- **JN / JP** — skoč při nastaveném / smazaném Negative (bit 7, tedy znaménko výsledku).
- **JV / JNV** — skoč podle příznaku znaménkového přetečení (pro pokročilé počítání).
- **JGT / JLE** — skoč při „ostře větší„ / „menší nebo rovno“ (bezznaménkově). Jediné skoky, které testují dva příznaky najednou (C a Z).
- **JGES / JLTS** — „větší nebo rovno„ / „menší“ **znaménkově** (testují N ⊕ V; k tomu se vrátíme níže).

Aliasy jako `JEQ` a `JLT` jsou jen čitelnější jména pro tytéž instrukce — píše se ti `JEQ konec` příjemněji než `JZ konec`, ale kód je bajt v bajt stejný.

> **Poznámka**
>
> `JGT` je v sadě až od ISA 3.1. „Větší než„ totiž potřebuje otestovat dva příznaky najednou (není menší **a zároveň** není rovno) — starší kód to proto psal dvěma skoky: nejdřív `JEQ` pryč a pak `JNC` do větve „větší“. Tenhle idiom ve starších programech potkáš pořád; `JGT` ho zkracuje na jedinou instrukci.

Odpočet: smyčka v praxi

Tady je smyčka, která odpočítá od pěti k jedné. Klíč je poslední řádek: `DEX` sníží X o jedna a nastaví příznak Z, když dojde na nulu; `JNZ .loop` opakuje, dokud nula nenastala:

```
; Odpočet 5..1 na terminál pomocí DEX + JNZ.
TERM_DATA EQU 0xFF10
main:
    LDX #5
.loop:
    TXA             ; A = X
    ADD #'0'        ; převeď číslici na ASCII ('0' = 48)
    STA TERM_DATA   ; vypiš
    DEX             ; X = X - 1 (nastaví Z při dosažení nuly)
    JNZ .loop       ; opakuj, dokud X != 0
    LDA #10
    STA TERM_DATA
    HLT
```

Výstup:

```
54321
```

Tohle je **počítání dolů** — a je to schválně. Kdybychom počítali nahoru, museli bychom po každém kroku zvlášť porovnat „už jsem na pěti?„. Ale směrem dolů nám příznak Z spadne do klína zadarmo: `DEX` sám řekne, kdy je hotovo. Tenhle drobný trik uvidíš v efektivním kódu pořád.

Porovnání

Obvyklý vzor je **porovnání následované skokem**. Instrukce `CMP` přijme libovolný adresový režim, interně odečte operand od A, nastaví příznaky — a výsledek zahodí (A zůstane nedotčené):

```
    LDA #10
    CMP #10      ; porovnej A s 10
    JEQ shoda    ; provede se, A je rovno 10
    ; ... sem se to nedostane ...
shoda:
    OUT
```

`CMP #10` porovná A s hodnotou 10, `CMP 0x50` s bajtem na adrese `0x50`. V obou případech se nastaví Z (byly rovné?), C (bylo A menší?) a N — a podle nich se pak rozhodneš skokem.

Jeden bajt, dvě čtení: znaménko versus bez

Tady je nejzrádnější místo celé kapitoly. Vzpomeň si z kapitoly 1, že `0xFF` je bezznaménkově 255, ale znaménkově −1. **Příznaky samy nevědí, které čtení máš na mysli** — musíš zvolit odpovídající skok:

- Porovnáváš **bezznaménkové** hodnoty (velikosti, adresy, počty)? Použij `JC` / `JLT`, `JNC` / `JGE` a pro ostré případy `JGT` / `JLE`.
- Porovnáváš **znaménkové** hodnoty (teploty, rozdíly)? Použij `JLTS` / `JGES` po `SUB` — testují N ⊕ V, což zůstává správné i při přetečení. (`CMP` na V nesahá, proto se znaménkové skoky kombinují s odčítáním.)

Konkrétní příklad: A = −1 (`0xFF`), provedeš `SUB #1`. Znaménková pravda je −1 < 1. `JLTS` správně skočí. Ale `JC` by porovnával 255 < 1, což *neplatí* — bralo by `0xFF` jako 255. Stejné bity, jiný závěr:

> **Pozor**
>
> Volba skoku **je** volbou interpretace. Když použiješ bezznaménkový skok na znaménková data (nebo naopak), program se bude tvářit, že funguje, a přitom bude tiše dělat špatná rozhodnutí. Vždy si ujasni, jestli tvá čísla mají znaménko, a podle toho vyber `JC` / `JNC` / `JGT` / `JLE`, nebo `JLTS` / `JGES`.

Umíme se rozhodovat a opakovat — to je jádro programování. V další kapitole přidáme zásobník a podprogramy, díky kterým půjde kód skládat z pojmenovaných, opakovaně volatelných kousků.

KAPITOLA 11

Zásobník a podprogramy

Představ si, že napíšeš užitečnou rutinu — třeba „vytiskni pozdrav„ — a chceš ji volat z pěti různých míst programu. Rutina musí nějak vědět, *kam se vrátit*, a to místo je pro každé volání jiné. Odpovědí je **zásobník**: paměť typu poslední-dovnitř-první-ven (LIFO, *last in, first out*), kam si stroj odkládá návratové adresy a cokoli dalšího, co potřebuje dočasně schovat.

Volání a návrat

Dvojice `CALL` a `RET` dělá z assembleru skládačku pojmenovaných kousků. `CALL rutina` uloží na zásobník návratovou adresu (kam se má program vrátit) a skočí do rutiny. `RET` si tu adresu vyzvedne zpět a vrátí se. Podívej:

```
; Podprogram volaný dvakrát pomocí CALL/RET.
TERM_DATA EQU 0xFF10
main:
    CALL pozdrav
    CALL pozdrav
    HLT

pozdrav:            ; vytiskne "Hi" a nový řádek, pak se vrátí
    LDA #'H'
    STA TERM_DATA
    LDA #'i'
    STA TERM_DATA
    LDA #10
    STA TERM_DATA
    RET
```

Výstup:

```
Hi
Hi
```

Rutinu `pozdrav` jsme napsali *jednou* a zavolali dvakrát. Po každém `CALL` se `RET` vrátil přesně na správné místo — poprvé na druhý `CALL`, podruhé na `HLT`. To je celé kouzlo podprogramů: napiš jednou, používej kdekoli.

Díky pravidlu LIFO se volání můžou **vnořovat**: rutina může volat další rutinu a návraty se odvinou přesně ve správném pořadí, jako když skládáš talíře na hromádku a bereš je z vrchu. Tenhle mechanismus — od padesátých let skoro nezměněný — je základ strukturovaného programování.

Zásobník žije v hlavní RAM

Jako každý skutečný procesor si i SAP-3/16 vykrajuje zásobník z obyčejné paměti. **SP** (stack pointer) je plnohodnotný šestnáctibitový ukazatel na jeho vrchol:

- SP se po startu nastaví na `0x0200` a roste **dolů**. `PUSH` (vlož) nejdřív sníží SP a pak zapíše, takže SP vždy ukazuje na nejnovější bajt a úplně první `PUSH` přistane na `0x01FF` — na vrcholu **stránky zásobníku** `0x0100`–`0x01FF` (konvence vypůjčená od 6502).
- **PUSH**: SP = SP − 1, pak ulož na `[SP]`. **POP**: přečti z `[SP]`, pak SP = SP + 1.

> **Poznámka**
>
> Že zásobník roste *dolů* a data *nahoru* není náhoda: obojí sdílí tutéž paměť a růstem proti sobě se potkají až v nejhorším případě. Kdyby rostly stejným směrem, srazily by se hned.

Co všechno zásobník umí

Kromě návratových adres se zásobník hodí na spoustu věcí:

- **Záchrana registrů.** Než rutina přepíše A, X nebo Y, může je odložit na zásobník a před `RET` obnovit. Slouží k tomu `PUSH` / `POP` (pro A), `PHX` / `PLX` (pro X), `PHY` / `PLY` (pro Y), `PHB` / `PLB` (pro B — jediný způsob, jak B zachránit bez zničení A) a `PHF` / `PLF` (pro příznaky — `PLF` obnoví všech pět včetně I).
- **Předávání parametrů.** Ulož argumenty, zavolej, rutina si je vyzvedne.
- **Rekurze.** Každá úroveň vnoření dostane vlastní kousek zásobníku, takže rutina může bezpečně volat sama sebe.
- **Rámce přerušení.** Při přerušení uloží PC a příznaky sám hardware — o tom kapitola 15.

> **Tip**
>
> Instrukce `TXYS` a `TSXY` přesouvají šestnáctibitový SP z/do dvojice X:Y. Hodí se, když chceš SP nastavit na jiné místo (třeba vlastní zásobník) nebo si jeho hodnotu prohlédnout.

Past, na kterou nezapomeň

SAP-3/16 nemá **žádnou hardwarovou ochranu proti přetečení zásobníku**. Splašený zásobník — třeba když rutina volá sama sebe donekonečna, nebo když `PUSH` bez odpovídajícího `POP` — se prostě prožírá pamětí dolů a přepisuje, co mu přijde do cesty. Je to přesně ta slavná třída chyb ze skutečných strojů (*stack overflow*).

> **Pozor**
>
> Každý `PUSH` musí mít svůj `POP` a každý `CALL` svůj `RET`. Když v rutině něco uložíš na zásobník a zapomeneš to sundat, `RET` vyzvedne místo návratové adresy tvoje data a skočí bůhvíkam. Nejlepší nástroj na odhalení takových chyb jsou **watchpointy** v debuggeru — nastav si hlídku na SP a uvidíš, kdy uteče, kam nemá.

Umíme skládat program z rutin. V další kapitole zjistíme, jak si samotné psaní zkrátit — pomocí maker.

KAPITOLA 12

Makra a preprocesor

Assembler je upřímný jazyk: jeden řádek, jedna instrukce. To je jeho síla, ale i slabina — opakující se vzory musíš psát pořád znovu. **Makra** tuhle otravu řeší. Makro je pojmenovaná šablona kódu; jednou ji definuješ a pak ji „voláš„ jako zkratku, která se před překladem rozbalí na plný kód. Není to nová instrukce — je to způsob, jak si zkrátit psaní.

O makra a další „textové„ úpravy se stará **preprocesor**, který proběhne *před* vlastním překladem. Jeho styl je převzatý z assembleru NASM, takže pokud znáš makra odjinud, budeš doma.

První makro

Vzpomeň si, jak jsme v předchozích kapitolách pořád psali dvojici „nahraj znak, zapiš na terminál„. Uděláme z ní makro `PUTC`:

```
; Makro PUTC vytiskne jeden znak. Zkrátí opakující se dvojici LDA/
STA.
%macro PUTC 1
    LDA #%1
    STA 0xFF10
%endmacro

main:
    PUTC 'A'
    PUTC 'h'
    PUTC 'o'
    PUTC 'j'
    PUTC 10
    HLT
```

Výstup:

```
Ahoj
```

Rozbor: `%macro PUTC 1` říká „definuj makro `PUTC`, které bere jeden argument„. Uvnitř `%1` znamená „sem dosaď první argument“. Když pak napíšeš `PUTC 'A'`, preprocesor

to nahradí za `LDA #'A'` a `STA 0xFF10`. Pět volání `PUTC` se rozbalí na deset instrukcí — ale zdroj se čte mnohem líp.

> **Poznámka**
>
> Makro **není** podprogram! `CALL` z kapitoly 11 skočí do jednoho sdíleného kusu kódu a vrátí se; makro naproti tomu **vloží kopii** svého těla na každé místo použití. Makro je rychlejší (žádný skok), ale zabere víc paměti. Volba mezi makrem a podprogramem je klasický kompromis rychlost versus velikost.

Unikátní labely uvnitř maker

Co když makro obsahuje smyčku, tedy label? Kdybys makro použil dvakrát, vznikly by dva stejné labely a překladač by protestoval. Řešením je `%%label`: dvojitý procentník vyrobí **unikátní** label pro každé rozbalení makra. Klasická ukázka je čekací smyčka:

```
%macro DELAY 1
    LDX #%1
%%loop:             ; unikátní label pro každé použití DELAY
    DEX
    JNZ %%loop
%endmacro

    DELAY 200        ; první "%%loop"
    DELAY 50         ; druhý, jiný "%%loop" - nekolidují
```

Obě použití mají vlastní `%%loop`, takže se navzájem nepobijí.

Podmínky, opakování a vkládání souborů

Preprocesor umí víc než jen makra:

- `%define JMENO hodnota` — textová konstanta (podobná `EQU`, ale nahrazuje se *před* překladem).
- `%if` / `%elif` / `%else` / `%endif` a `%ifdef` / `%ifndef` — podmíněný překlad: části kódu se přeloží jen podle nějaké podmínky. Hodí se pro varianty programu (třeba ladicí verze).
- `%rep N` … `%endrep` — zopakuj blok N-krát. Skvělé na rozbalení malých smyček nebo generování tabulek.
- `%include "soubor.asm"` — vlož obsah jiného souboru. Takhle se dělají knihovny.

Knihovny a include-guardy

Když si napíšeš užitečná makra, uložíš je do souboru a `%include` neš, kde je potřebuješ. Repozitář má takovou knihovnu připravenou — `examples/include/stdlib.asm`. Jediná past: kdyby se stejný soubor vložil dvakrát, makra by se definovala dvakrát a překladač by si stěžoval. Řeší to **include-guard** — vzor, který zajistí, že se obsah zpracuje jen jednou:

```
%ifndef STDLIB_INCLUDED       ; ještě nevloženo?
%define STDLIB_INCLUDED 1     ; označ jako vloženo
    ; ... zde jsou definice maker ...
%endif
```

Pokud `%include` neš `stdlib.asm` desetkrát, tělo se zpracuje jen jednou — podruhé už je `STDLIB_INCLUDED` definované a `%ifndef` obsah přeskočí. Je to stejný trik jako hlavičkové guardy v jazyce C.

> **Tip**
>
> Makra drž malá a s jasným účelem. Velké makro, které dělá deset věcí, se čte hůř než podprogram. Dobré makro pojmenuje jeden opakující se vzor — `PUTC`, `DELAY`, `PUSH_ALL` — a tím kód zpřehlední, ne zamlží.

Umíme psát čitelný, dobře strukturovaný a úsporný kód. V poslední kapitole této části posbíráme osvědčené řemeslné fígle, kterými se z fungujícího programu stane program *rychlý a úsporný*.

KAPITOLA 13

Řemeslné fígle a optimalizace

Práce v těsných mezích — jeden akumulátor, osmibitová ALU, paměť za cenu taktů — je klasická škola efektivního programování. Když máš málo, naučíš se s tím zacházet chytře. Tahle kapitola sbírá osvědčené fígle, kterými z fungujícího programu uděláš program *rychlý a úsporný*. Nejsou to triky pro efekt; jsou to způsoby myšlení, které se hodí i o padesát let a mnoho abstrakcí výš.

Drž horké hodnoty co nejblíž

Přístup do nulté stránky je o bajt kratší a o tři takty rychlejší než absolutní (kapitola 14). Proto **horké proměnné patří do nulté stránky** — je to nejbližší věc „registrům navíc„, jakou tenhle stroj má. A cokoli, s čím pracuješ opravdu intenzivně, drž rovnou v registru A, X nebo Y; do paměti sahej, až když musíš.

Využívej příznaky, které dostáváš zadarmo

Loady a přesuny nastavují Z a N podle přenášené hodnoty (kapitola 6). To znamená, že `LDA counter` následované `JZ hotovo` nepotřebuje žádné `CMP` — počítadlo se testuje samo. Zvykni si číst tabulku „kdo nastavuje příznaky„ jako seznam *dárků*: pokaždé, když ti nějaká instrukce nastaví příznak, který jsi stejně potřeboval, ušetříš instrukci navíc.

Počítej dolů, ne nahoru

Už jsme to viděli v kapitole 10, ale stojí to za zopakování: `DEC` (nebo `DEX` / `DEY`) plus `JNZ` uzavře smyčku dvěma instrukcemi, protože `DEC` sám nastaví Z při dosažení nuly. Počítání nahoru potřebuje porovnání navíc („jsem už na konci?„). Kdykoli nezáleží na *pořadí* průchodů, počítej dolů.

Posouvej místo násobení

`SHL` hodnotu zdvojnásobí, `SHR` půlí — a obojí je mnohem levnější než `MUL`. Násobení mocninou dvou je prostě posun. A i „ošklivé„ násobky se dají poskládat: násobení deseti je `×2`, pak `×4`, přičti původní (`×5`) a nakonec `×2` (`×10`). Trocha přemýšlení ušetří drahou operaci.

Choď s ukazateli

Na cokoli delšího než 256 bajtů si drž šestnáctibitový **ukazatel** v nulté stránce, čti přes něj pomocí `(ptr)` a posouvej ho instrukcí `INW` — nebo o celý krok najednou přes `ADW ptr, #n` (třeba o šířku řádku obrazovky). Klíčové je, že obě **schválně nesahají na příznaky** (kapitola 6), takže ti nikdy nerozbijí podmínku smyčky, kterou zrovna testuješ. Ukazatel na začátku naplníš dvojicí `LXY #adresa` + `SXY ptr`. Průchod libovolně velkou datovou strukturou je pak čistá, krátká smyčka.

Pamatuj na registr B

Každá binární ALU instrukce přepíše B (kapitola 4). Operandy pro `MUL` / `DIV` proto chystej přes `TAB` *těsně* před samotné násobení nebo dělení — a když B potřebuješ zachránit přes cizí kód, ulož ho `PHB` / `PLB`. Počítat s tím, že B něco přežije přes `ADD`, je spolehlivá cesta k záhadné chybě.

> **Pozor**
>
> „Záhadné chyby„ na tomhle stroji mají překvapivě často jednu ze dvou příčin: zapomenutý `#` u immediate (kapitola 8), nebo přepsaný registr B. Když program dělá nesmysl a kód *vypadá* správně, zkontroluj nejdřív tyhle dvě věci.

Spi, netoč se

Čekat na událost pollovací smyčkou znamená spálit 100 % času otázkou „už? už?„. Elegantnější je nechat se vzbudit přerušením: `EI` (povol přerušení) následované `HLT` uspí procesor, dokud něco nepřijde (kapitola 15). Šetří to nejen elegancí, ale i „energií“ — přesně tak zahálejí skutečné operační systémy.

Rozbaluj malé smyčky

Když máš v paměti místo, čtyřnásobné zopakování těla smyčky srazí režii (`DEC` + `JNZ` za každý průchod) na čtvrtinu. Je to věčný obchod **rychlost versus velikost** — a preprocesorové `%rep` z kapitoly 12 ti rozbalení napíše za tebe.

Sebemodifikující kód

A na závěr ten nejzazší fígl, přímo z ducha uloženého programu: kód *jsou* bajty v RAM, takže program může **přepisovat vlastní instrukce** — třeba záplatovat adresu uvnitř `LDA` těsně před jejím vykonáním. Dnes se to považuje za nebezpečný styl (těžko se ladí, moderní procesory ho nemají rády), ale je to autentická osmibitová technika a nejčistší možná demonstrace myšlenky z kapitoly 2: mezi kódem a daty *není* rozdíl.

> **Poznámka**
>
> Optimalizuj až to, co měříš. Emulátor ti u každého programu ukáže počet taktů — nejdřív napiš kód *správně a čitelně*, pak se podívej, kde doopravdy tráví čas, a teprve tam sáhni po těchhle fíglech. Předčasná optimalizace zamlží kód a málokdy se vyplatí.

Tím končí část o programování. Umíš stroji poroučet. V další části ho propojíme se světem — s pamětí, přerušeními, klávesnicí, zvukem a obrazem.

ČÁST IV

Paměť, přerušení a svět kolem

KAPITOLA 14

Paměťová mapa a nultá stránka

Se šestnáctibitovými adresami vidí procesor plochých **64 KB** paměti — žádné bankování, žádné triky, jen jedno velké pole bajtů od `0x0000` do `0xFFFF`. Jednotlivé oblasti se neliší hardwarem (je to pořád táž RAM), ale **konvencí** a tím, co je na ně případně zapojeno. Znát tuhle mapu je jako znát rozvržení domu — víš, kam co patří.

Mapa paměti

Rozsah	Oblast	K čemu
`0x0000` – `0x00FF`	**Nultá stránka**	rychlá data + sloty pro ukazatele
`0x0100` – `0x01FF`	Stránka zásobníku	kde konvencí žije SP (kapitola 11)
`0x0200` – `0x7FFF`	Program + data	zhruba 31,5 KB; sem se dává program
`0x8000` – `0xBFFF`	VRAM	obraz: znaky, glyfy, barvy (kapitola 18)
`0xC000` – `0xFEFF`	Datová RAM	zhruba 16 KB pro velká pole
`0xFF00` – `0xFF7F`	I/O registry	paměťově mapovaná zařízení (kapitola 16)
`0xFF80` – `0xFFF9`	Systémová RAM	scratch pro obsluhy, boot ROM
`0xFFFA` – `0xFFFB`	**BRK vektor**	obsluha softwarového přerušení
`0xFFFC` – `0xFFFD`	**RESET vektor**	kde vykonávání začíná
`0xFFFE` – `0xFFFF`	**IRQ vektor**	kam jdou přerušení (kapitola 15)

Tabulka 6. Mapa 64 KB paměti SAP-3/16. Táž RAM, různé role dané konvencí a zapojením.

Nultá stránka: rychlý pruh a kartotéka ukazatelů

Prvních 256 bajtů — **nultá stránka** — má výsadní postavení. Adresa do ní se vejde do jediného bajtu, takže instrukce je kratší a o tři takty rychlejší než plný absolutní přístup. Z toho plynou dvě věci:

- **Horké proměnné patří sem.** Je to nejbližší náhrada za „registry navíc„, přesně jako na 6502.
- **Je to kartotéka ukazatelů.** Libovolné dva sousední bajty nulté stránky mohou nést šestnáctibitovou little-endian adresu a nepřímý režim `(zp)` čte *skrz* tento ukazatel.

Tady je to naživo — ukazatel `PTR` v nulté stránce projde řetězec kdekoli v paměti:

```
; Výpis řetězce přes ukazatel v nulté stránce: LDA (PTR) + INW
PTR.
TERM_DATA EQU 0xFF10
PTR       EQU 0x10        ; slot pro 16bitový ukazatel (0x10 a
0x11)

main:
    LDA #<msg             ; dolní bajt adresy msg
    STA PTR
    LDA #>msg             ; horní bajt adresy msg
    STA PTR+1
.loop:
    LDA (PTR)             ; načti znak z místa, kam PTR ukazuje
    JEQ .done             ; 0 = konec
    STA TERM_DATA
    INW PTR               ; posuň ukazatel o jeden bajt (nemění
příznaky)
    JMP .loop
.done:
    HLT

msg: DATA "Ukazatel!", 10, 0
```

Výstup:

```
Ukazatel!
```

Nejdřív sestavíme ukazatel: `#<msg` a `#>msg` uloží dolní a horní bajt adresy do `PTR` a `PTR+1`. Pak smyčka čte `(PTR)`, tiskne a posouvá ukazatel `INW`em. Protože `INW` nemění příznaky, `LDA (PTR)` hned nato nastaví Z podle znaku a `JEQ` bezpečně pozná konec. Takhle osmibitový stroj prochází data kdekoli v celých 64 KB — ne přes indexový registr (ten umí jen 0–255), ale přes ukazatel.

Tip

Rozdíl mezi `LDA tab,X` (kapitola 8) a `LDA (PTR)` je zásadní. Indexování `,X` je skvělé pro krátká pole (X je jen osmibitový). Ukazatel `(PTR)` dosáhne kamkoli v paměti a `INW` ho posouvá bez omezení. Krátká data indexuj, dlouhá procházej ukazatelem.

Vektory a start

Jak procesor po zapnutí ví, kde má začít? Na samém vrcholu paměti sedí dva šestnáctibitové **vektory** — ukazatele, které zapíše assembler a přečte hardware:

- **RESET vektor (`0xFFFC` / `0xFFFD`)** — při startu procesor tyhle dva bajty přečte do PC a začne tam vykonávat. Assembler ho naplní vstupním bodem tvého programu automaticky.
- **IRQ vektor (`0xFFFE` / `0xFFFF`)** — totéž pro přerušení (kapitola 15), nastavíš přes `.VECTOR IRQ, …`.

Na startovací adrese tedy **není nic magického** — jsou to jen data na dobře známém místě. Přesně takhle startuje 6502.

Paměťově mapované I/O v kostce

SAP-3/16 nemá zvláštní instrukce pro vstup a výstup. Zařízení místo toho reagují na konkrétní **adresy**: zápis na `0xFF10` vytiskne znak, čtení z `0xFF00` vyzvedne klávesu. Tomu se říká **paměťově mapované I/O** a je to převládající návrh ve skutečných systémech. Každé zařízení vlastní šestnáctibajtový slot v bloku `0xFF00` – `0xFF7F`. Celou galerii zařízení projdeme v následujících kapitolách.

Máme mapu paměti a víme, jak stroj startuje. V další kapitole se podíváme, jak reaguje na *neočekávané* události — přes přerušení.

KAPITOLA 15

Přerušení a start počítače

Jak si má procesor všimnout, že uživatel stiskl klávesu, nebo že uplynul časový úsek? Jsou dvě cesty a rozdíl mezi nimi je jedním z důležitých témat celé počítačové architektury.

Polling versus přerušení

Naivní způsob je **polling** — smyčka, která pořád dokola čte stavový registr zařízení: „už něco? už něco?„. Funguje to, ale procesor přitom spálí 100 % času ptaním se — a když se ptá moc pomalu, může událost úplně prošvihnout.

Elegantní způsob je **přerušení** (interrupt): zařízení procesoru samo „poklepe na rameno„. Procesor dokončí právě běžící instrukci, automaticky si uloží, kde skončil, a skočí do zvláštní rutiny — **obsluhy přerušení** (ISR). Když obsluha doběhne, přerušený program pokračuje, jako by se nic nestalo. Přerušeními řeší každý skutečný počítač vstup a výstup, měření času i multitasking.

Tři zdroje, dvě pojistky

SAP-3/16 má tři zdroje přerušení: **časovač** (perioda se nastavuje registrem `TMR_DIV` na `0xFF30`), **klávesnici** a **sériovou linku** (příjem bajtu — kapitola 19). Které z nich smějí přerušovat, řídí dvě vrstvy:

- **IRQ_ENABLE (`0xFF40`)** — bitová maska: bit 0 povoluje časovač, bit 1 klávesnici, bit 2 sériový příjem.
- **Příznak I** — hlavní vypínač, ovládaný instrukcemi `EI` (povol) a `DI` (zakaž). Po resetu jsou přerušení zamaskovaná, takže nic nevystřelí, dokud se program sám nepřihlásí.

Musíš tedy odemknout obě pojistky: odmaskovat konkrétní zdroj v `IRQ_ENABLE` *a* zapnout hlavní vypínač přes `EI`.

Vstupní sekvence

Když čeká povolené přerušení, procesor dokončí právě běžící instrukci a vloží hardwarovou sekvenci (ve schématu ji uvidíš jako pseudo-opcode):

1. uloží PC (horní bajt, pak dolní) na zásobník,
2. uloží příznaky sbalené do jednoho bajtu — C/Z/N/V *i příznak I* — a smaže I (aby se přerušení nevnořovalo do sebe),

3. přečte **IRQ vektor na** `0xFFFE` / `0xFFFF` do PC — stejný vzor dvou čtení jako při startu.

Obsluha bydlí, kam ji umístíš; nasměruješ na ni vektor přes `.VECTOR IRQ, obsluha`. Končí instrukcí **RTI** (*return from interrupt*), která vyzvedne bajt příznaků zpět (obnoví všech pět včetně I) a pak návratové PC.

Časovač v akci

Tady je celý mechanismus pohromadě. Hlavní program jen **spí** (`HLT`); veškerou práci dělá obsluha, kterou budí časovač. Po pěti hvězdičkách si obsluha časovač zamaskuje sama, takže poslední `HLT` už zůstane konečný:

```
; Časovač vyvolá přerušení; obsluha vytiskne hvězdičku a po pěti
se zastaví.
TERM_DATA  EQU 0xFF10
TMR_DIV    EQU 0xFF30
IRQ_ENABLE EQU 0xFF40
IRQ_STATUS EQU 0xFF41

    .VECTOR IRQ, isr        ; nasměruj IRQ vektor na obsluhu

main:
    LDA #200
    STA TMR_DIV             ; časovač střílí každých 200 taktů
    LDA #1
    STA IRQ_ENABLE          ; odmaskuj časovač (bit 0)
    EI                      ; hlavní vypínač přerušení
sleep:
    HLT                     ; spi a čekej na přerušení
    JMP sleep

; --- obsluha přerušení ---
isr:
    LDA IRQ_STATUS          ; potvrď časovač (čtení maže záchytku)
    LDA #'*'
    STA TERM_DATA           ; vytiskni hvězdičku
    INC counter
    LDA counter
    CMP #5
    JZ .finish
    RTI                     ; zpátky spát
```

```
.finish:
    LDA #0
    STA IRQ_ENABLE          ; zamaskuj časovač -> HLT bude konečný
    RTI

counter: DATA 0
```

Výstup:

```
*****
```

HLT znamená „počkej na přerušení„

Všimni si `HLT` v hlavní smyčce. S **povolenými** přerušeními neznamená `HLT` „konec navždy„, ale „*uspi procesor, dokud nepřijde přerušení*“. Časovač dál tiká, každý tik procesor probudí, proběhne obsluha, `RTI` se vrátí — a `HLT` uspí procesor znovu. Přesně takhle zahálejí skutečné operační systémy: místo aby se vysilovaly pollingem, spí a nechají se budit.

Nezapomeň potvrdit

Obsluha **musí smazat zdroj, který ji probudil.** Čtení `IRQ_STATUS` (`0xFF41`) potvrdí (a smaže) záchytku časovače, čtení `KBD_DATA` (`0xFF00`) potvrdí klávesnici. Když na to zapomeneš, totéž přerušení vystřelí hned po `RTI` znovu — a znovu, donekonečna:

Pozor

IRQ storm. Obsluha, která nepotvrdí svůj zdroj, se spustí okamžitě po návratu znovu a program zamrzne v nekonečné obsluze přerušení. U časovače navíc dbej, ať je perioda (`TMR_DIV`) **delší** než doba běhu obsluhy — jinak se přerušení hromadí rychleji, než je stíháš odbavit, a hlavní program „vyhladovíš„.

Poznámka

Hardware ti automaticky uloží jen **příznaky** a PC. Záchrana **registrů** (A, X, Y), na které obsluha sáhne, je její vlastní zodpovědnost — na začátku je odlož (`PUSH` / `PHX` / `PHY`), před `RTI` obnov. V naší ukázce to nebylo potřeba, protože hlavní smyčka registry stejně nepoužívá.

Existuje ještě **softwarové** přerušení `BRK` — skutečná instrukce, která provede tentýž rámec jako IRQ, ale spouští se z kódu a vektoruje přes vlastní vektor `0xFFFA`. Slouží pro systémová volání a breakpointy.

Stroj umí reagovat na svět. V další kapitole projdeme konkrétní zařízení — klávesnici, terminál a čas.

KAPITOLA 16

Klávesnice, terminál a čas

Všechna zařízení jsou dostupná přes paměťově mapované registry z kapitoly 14 — čteš z nich a zapisuješ do nich jako do obyčejné paměti, jen na druhé straně sedí hardware. Projděme si ta „lidská„ zařízení: vstup z klávesnice, výstup na terminál a měření času.

Klávesnice

Klávesnice má dva registry: `KBD_STATUS` (`0xFF01`) říká, jestli čeká klávesa, a `KBD_DATA` (`0xFF00`) vydá její ASCII kód. Čtení `KBD_DATA` zároveň slouží jako potvrzení — přečtenou klávesu tím „spotřebuješ„. Tady je echo, které opisuje stisknuté klávesy na terminál, dokud nepřijde Enter:

```
; Echo klávesnice: čti klávesy a piš je na terminál, dokud
nepřijde Enter.
KBD_STATUS EQU 0xFF01
KBD_DATA   EQU 0xFF00
TERM_DATA  EQU 0xFF10

main:
.wait:
    LDA KBD_STATUS      ; čeká klávesa?
    JZ .wait            ; ne -> polluj dál
    LDA KBD_DATA        ; ano -> přečti ji (a potvrď)
    STA TERM_DATA       ; ozvi ji na terminál
    CMP #10             ; byl to nový řádek (Enter)?
    JNZ .wait           ; ne -> další klávesa
    HLT
```

Se vstupem `ahoj` a Enter vypíše:

```
ahoj
```

Smyčka `.wait` je čítankový **polling**: pořád dokola se ptá `KBD_STATUS`, dokud není klávesa připravená. Funguje to, ale procesor u toho nic jiného nedělá. Elegantnější je nechat se vzbudit přerušením od klávesnice (kapitola 15) — přesně jak jsme to udělali s časovačem.

Gamepad

Klávesnice hlásí jednu klávesu po druhé a po přečtení ji zapomene — skvělé pro psaní, k ničemu pro hru, která potřebuje vědět, že je tlačítko *stále držené*. Tuhle mezeru vyplňuje **gamepad**: `PAD1` (`0xFF70`) a `PAD2` (`0xFF71`) jsou bitmapy právě stisknutých tlačítek, jeden osmibitový port na hráče. Bity jsou 0 Nahoru, 1 Dolů, 2 Vlevo, 3 Vpravo, 4 Palba, 5 B, 6 Start, 7 Select. Čtení je nemaže, takže se stačí podívat jednou za snímek:

```
    LDA 0xFF70   ; tlačítka hráče 1
    AND #0x04    ; drží se Vlevo? (bit 2)
    JNZ MOVE_LEFT
```

> **Tip**
>
> Pro „výstřel jednou za stisk„ místo autopalby si pamatuj bitmapu z minulého snímku a reaguj jen na *nově* nastavené bity: `nyní AND NOT dříve` . Tím oddělíš „drží se“ od „právě stiskl„.

Terminál

Terminál je nejjednodušší výstup: zapiš ASCII bajt do `TERM_DATA` (`0xFF10`) a objeví se na obrazovce. Viděli jsme to už v prvním programu. Kromě tisknutelných znaků reaguje i na několik **řídicích znaků** — třeba 10 (nový řádek) nebo 8 (backspace). Tisk celého řetězce je smyčka: čti bajty přes ukazatel (`LDA (ptr)`), zastav na nule a každý bajt pošli na `0xFF10` — přesně jak jsme to dělali v kapitole 14.

Jedna poctivost navíc: za `TERM_DATA` sedí sériová linka s pevnou rychlostí a **64bajtovou frontou** — krátké výpisy odejdou „hned„, ale kdo chrlí tisíce znaků, musí se před každým zápisem zeptat `TERM_STATUS` (bit 0 = ve frontě je místo), jinak se přebytečné bajty ztratí. Je to stejný vzor „než zapíšeš, zeptej se“, který podrobně rozebereme u sériové linky v kapitole 19.

Časovač

Časovač jsme potkali v kapitole 15. `TMR_DIV` (`0xFF30`) určuje, jak často střílí přerušení (v T-stavech), `TMR_CNT` (`0xFF31`) zpřístupňuje běžící počet. Časovač je srdeční rytmus všeho periodického — snímky animace, tempo hudby, timeouty. Povol jeho přerušení a práci dej do obsluhy.

Kolik je hodin? Čítače a reálný čas

Kromě časovače má stroj sadu jen pro čtení, kterou program zjistí, „kolik toho spočítal„ a „jak dlouho běží“:

- **Čítač taktů** (`0xFF32` – `0xFF35` , 32 bitů) — kolik T-stavů uplynulo od startu. Odečteš ho na začátku a na konci úseku a máš přesnou míru, jak dlouho úsek trval — obdoba profilování časovým razítkem.
- **Uptime** (`0xFF36` – `0xFF37`) — kolik snímků (period obrazu) uběhlo; hrubší, ale levné hodiny.
- **Takt za milisekundu** — `CLK_TPMS` (`0xFF38` – `0xFF39`) — kolik T-stavů odpovídá jedné reálné milisekundě (tedy rychlost hodin dělená tisícem). To je klíč k tomu, aby hra běžela **stejně rychle** bez ohledu na to, jak je stroj přetaktovaný.

> **Poznámka**
>
> Vícebajtové čítače se čtou zvláštním trikem: přečtení *dolního* bajtu zamkne celou hodnotu do stínové kopie, ze které pak přečteš vyšší bajty. Tím máš jistotu, že se ti čtyři bajty čítače „nerozjedou„ uprostřed čtení, i kdyby zrovna tikl. Skutečné procesory to řeší úplně stejně.

Proč tolik hodin? Když necháš herní smyčku běžet „jak nejrychleji to jde„, poběží na pomalém stroji jinak než na rychlém. Klíčem k plynulé hře je čtvrtý čítač — `CLK_MS` (`0xFF3A` – `0xFF3B`), který tiká jednou za **reálnou milisekundu** bez ohledu na takt (uvnitř není nic záhadného: počítá T-stavy a po každých `CLK_TPMS` z nich udělá jednu milisekundu). Herní smyčka si zapamatuje čas posledního snímku a počká, až od něj uplyne šestnáct milisekund — a hra běží stejně plynule při 100 kHz i při 2 MHz. Je to přesně ten druh detailu, který odlišuje hračku od skutečného stroje: skutečné konzole také drží pevnou obnovovací frekvenci, ne pevný počet instrukcí.

Umíme číst vstup, psát text a měřit čas. V dalších dvou kapitolách přijdou dvě nejzábavnější zařízení — zvukový čip a obrazovka.

KAPITOLA 17

Zvukový čip — tři hlasy

Teď přijde jedno z nejzábavnějších zařízení. SAP-3/16 má malý **tříhlasý zvukový čip** — tři nezávislé **kanály** (hlasy), které umí hrát současně, takže složíš akord nebo melodii s doprovodem. A co je hezké: procesor jen zapíše pár bajtů do registrů; **veškerý zvuk pak dělá hardware sám.** Program nekreslí zvukovou vlnu vzorek po vzorku — jen řekne „hraj tenhle tón takhle„ a čip se postará o zbytek.

Frekvence přímo v hertzích

Každý kanál má šestnáctibitovou frekvenci, a to **rovnou v hertzích**. Chceš komorní A? Zapiš 440. Žádné přepočty, žádné tabulky dělitelů — číslo, které uložíš, je frekvence tónu. Protože je šestnáctibitová, rozloží se na dolní a horní bajt (`AUD_FREQ_LO` / `AUD_FREQ_HI`), přesně jako každá jiná šestnáctibitová hodnota.

Hudební noty odpovídají konkrétním frekvencím. Tady je jedna oktáva C-dur, kterou stačí zapsat:

C4	D4	E4	F4	G4	A4	H4	C5
262	294	330	349	392	440	494	523

Tabulka 7. Nota ⟶ frekvence v Hz. Zapiš číslo do frekvenčních registrů a kanál zahraje tón.

Registry jednoho kanálu

Kanál 0 má pět registrů hned za sebou:

Adresa	Registr	Význam
`0xFF21` / `22`	`AUD_FREQ_LO` / `HI`	frekvence v Hz (16 bitů)
`0xFF23`	`AUD_WAVE`	tvar vlny (viz níže)
`0xFF24`	`AUD_AD`	obálka: horní půlbajt attack, dolní decay
`0xFF25`	`AUD_SR`	obálka: horní půlbajt sustain, dolní release

Tabulka 8. Pětice registrů zvukového kanálu.

Kanály 1 a 2 mají tutéž pětici, jen posunutou: základ kanálu N je `0xFF21 + N × 5`, tedy `0xFF21`, `0xFF26` a `0xFF2B`. Toho se dá chytře využít — s indexovým registrem `X` nastaveným na 0, 5 nebo 10 adresuje `STA AUD_FREQ_LO,X` kterýkoli kanál toutéž instrukcí.

Gate: hrana spouští tón

Nad všemi třemi kanály sedí jeden řídicí registr `AUD_CTRL` (`0xFF20`) — **bitmapa gate**, jeden bit na kanál. Reaguje na **hranu**, ne na úroveň: přechod bitu z 0 na 1 tón *spustí* (rozezní obálku od začátku), přechod z 1 na 0 spustí *doznění*. To má dva příjemné důsledky: jedním zápisem spustíš akord na všech třech hlasech naráz, a opakovaný zápis *stejné* hodnoty tón neretrigeruje (dokud bit nesklopíš a zase nezvedneš, hraje dál).

Tady je nejjednodušší pípnutí — komorní A, obdélníkový průběh:

```
; Pípnutí: komorní A (440 Hz), obdélníkový průběh, krátký tón.
AUD_CTRL    EQU 0xFF20        ; bitmapa gate (bit N = kanál N)
AUD_FREQ_LO EQU 0xFF21
AUD_FREQ_HI EQU 0xFF22
AUD_WAVE    EQU 0xFF23

main:
    LDA #1
    STA AUD_WAVE        ; 1 = obdélník
    LDA #<440
    STA AUD_FREQ_LO     ; frekvence přímo v Hz (dolní bajt)
    LDA #>440
    STA AUD_FREQ_HI     ; horní bajt
    LDA #1
    STA AUD_CTRL        ; gate kanálu 0: hrana 0->1 spustí tón
.delay:
    LDX #200
.d:
    DEX
    JNZ .d
    LDA #0
    STA AUD_CTRL        ; gate 0: hrana 1->0 spustí doznění
(release)
    HLT
```

V příkazové řádce program jen tiše proběhne — ale spusť ho v emulátoru a uslyšíš krátké pípnutí.

Tvary vlny

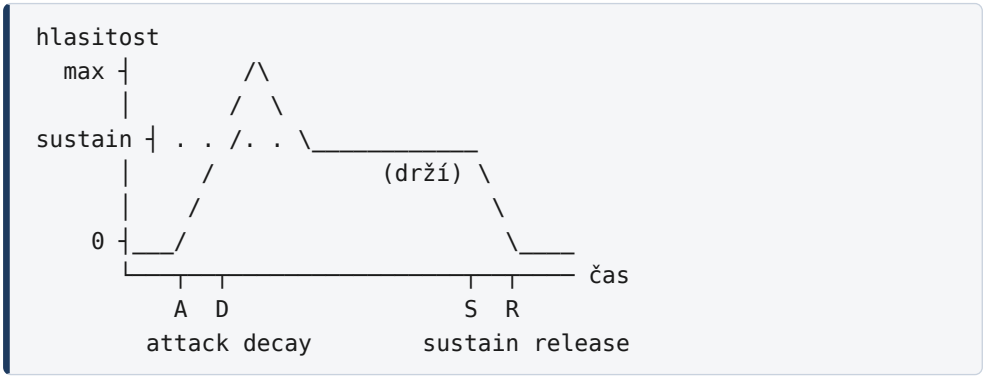
Registr `AUD_WAVE` volí barvu zvuku:

Hodnota	Tvar vlny a charakter
0	sinus — čistý, měkký tón
1	obdélník — dutý, „herní„ zvuk
2	trojúhelník — jemný, flétnový
3	pila — ostrý, bzučivý
4	šum — bez výšky; z něj se dělají bicí a exploze

Tabulka 9. Pět tvarů vlny. Šum nemá tón — hodí se na perkuse a zvukové efekty.

Obálka ADSR: tvar tónu v čase

Skutečný nástroj nezní pořád stejně hlasitě — tón *nastoupí*, chvíli *dozní* na ustálenou hlasitost, tu *drží* a po puštění klávesy *vyhasne*. Tomuhle průběhu hlasitosti se říká **obálka ADSR** podle jejích čtyř fází:



- **Attack** — jak rychle tón naskočí na maximum,
- **Decay** — jak rychle klesne na úroveň sustain,
- **Sustain** — na jaké hlasitosti se drží, dokud je gate zapnuté,
- **Release** — jak rychle vyhasne po vypnutí gate.

Časové fáze (A, D, R) se zadávají po půlbajtech v registrech `AUD_AD` a `AUD_SR`. Půlbajt 0 znamená *okamžitě*, jinak čas roste zhruba od 8 ms (hodnota 1) po zhruba 1024 ms (hodnota 15). Sustain je *úroveň*, ne čas: 0 je ticho, 15 plná hlasitost. Zapamatuj si to takhle:

- `AUD_AD` = attack (horní půlbajt) · decay (dolní),
- `AUD_SR` = sustain (horní půlbajt) · release (dolní).

Tip

Pár osvědčených nastavení, od kterých se dá odrazit: **drnknutí** (pluck) — attack okamžitý, decay střední, sustain nízký; **varhany** — attack rychlý, sustain plný, release krátký; **úder bicích** — šumový průběh, attack okamžitý, decay krátký, sustain nula. Výchozí stav čipu (sustain plný, ostatní okamžité) je prostý vypínač „hraje / nehraje„.

Akord na tři hlasy

Protože kanály jsou tři a gate spouští hranou, zahraješ akord jedním zápisem. Nastav frekvence, tvary a obálky všech tří kanálů (klidně přes `,X` s X = 0, 5, 10) a pak jediné `STA AUD_CTRL` s hodnotou `0b00000111` rozezní všechny tři naráz — dokonale synchronně. Kompletní tříhlasý „tracker„ najdeš mezi ukázkami (`examples/sound/mini-tracker.asm`).

Zvuk máme. Zbývá poslední velké zařízení — obrazovka. To je téma další kapitoly.

KAPITOLA 18

Obraz, barvy a blitter

Poslední a nejnápadnější zařízení je obrazovka. SAP-3/16 má **textový režim 40×25 znaků** — čtyřicet sloupců, dvacet pět řádků — a k tomu barvy, vlastní znakovou sadu a hardwarovou výpomoc pro rychlé kreslení; ke konci kapitoly uvidíš, že umí i **bitmapovou grafiku a sprity**. Zajímavé je, že obraz **není zvláštní hardware „někde jinde„**: je to obyčejná paměť, do které zapisuješ, a video zařízení ji jen průběžně čte a zobrazuje.

Obrazová paměť

Znakový buffer začíná na `0x8000`. Každý bajt je jedna buňka obrazovky a její adresa se počítá jako `0x8000 + řádek × 40 + sloupec`. Zapiš do buňky ASCII kód a objeví se odpovídající znak. Nejdřív ale musíš obraz zapnout — bit 0 registru `VID_CTRL` (`0xFF50`). Tady je „ahoj na obrazovku„:

```
; Zápis textu do obrazové paměti (VRAM). Znaky se objeví na
obrazovce.
VID_CTRL EQU 0xFF50        ; bit 0 = zapni obraz
SCREEN   EQU 0x8000        ; znakový buffer 40x25, adresa = SCREEN
+ y*40 + x

main:
    LDA #1
    STA VID_CTRL           ; zapni textový režim
    LDX #0
.copy:
    LDA msg,X              ; znaky řetězce (0 = konec)
    JZ .done
    STA SCREEN,X          ; zapiš do VRAM na řádek 0
    INX
    JMP .copy
.done:
    HLT

msg: DATA "AHOJ", 0
```

Po doběhnutí drží paměť od `0x8000` bajty `41 48 4F 4A` — ASCII kódy `A H O J`. Spusť program v emulátoru a uvidíš je v levém horním rohu. Je to táž smyčka jako výpis na terminál z kapitoly 8, jen cíl není terminál, ale buňka obrazovky.

Barvy

Vedle znakového bufferu leží **barevná RAM** (od `0x8C00`, tedy o `0x0C00` výš než odpovídající znak). Každá buňka má jeden bajt barvy: dolní půlbajt volí barvu **písma**, horní půlbajt barvu **pozadí**. Barev je **šestnáct**, ve stylu palety Commodore 64 — od černé a bílé přes odstíny modré, zelené, červené až po šedé. Hodnota `0x00` znamená „výchozí barvy„. Kompletní paletu najdeš v příloze D.

Kromě znaků má obraz **okraj** (`VID_BORDER`, `0xFF54`) — jednobarevný rám kolem plochy, oblíbený u osmibitových her na signalizaci stavu (blikne, když se něco stane).

Kurzor a synchronizace snímků

Hardwarový kurzor umístíš registry `VID_CURX` (`0xFF52`) a `VID_CURY` (`0xFF53`) — určíš sloupec a řádek, kde má blikat. A registr `VID_STATUS` (`0xFF51`) hlásí **vsync** — tik na začátku každého snímku (čtení ho smaže). Tenhle tik je zásadní pro plynulou animaci: počkáš na něj, teprve pak překreslíš obraz, a pohyb je hladký místo trhaného.

> **Poznámka**
>
> Čekání na vsync je stejný princip jako čekání na časovač z kapitoly 15 — jen místo `TMR` sleduješ `VID_STATUS`. Animační smyčka pak běží tempem obrazu: jednou za snímek zjisti vstup, posuň objekty, překresli. Přesně tak fungovaly osmibitové hry.

Blitter: hardwarový stěhovák

Tady je problém. Představ si scrollovací hru, kde se celá plocha každý snímek posune o řádek. To je kopie skoro devíti set bajtů — a na holém procesoru je to smyčka o tisících instrukcí, která se do rozpočtu jednoho snímku (řádově tisíc taktů) nevejde. Hra by se zadrhávala.

Řešením je **blitter** — malý specializovaný stěhovák, který umí přesouvat a vyplňovat bloky paměti **přímo, bez procesoru**. Nastavíš mu zdroj (`BLT_SRC`), cíl (`BLT_DST`), délku (`BLT_LEN`) a spustíš příkaz zápisem do `BLT_CTRL` (`0xFF79`) — kopírování (`1`), nebo výplň (`2`). Blitter to provede během jediného zápisu, za nulové takty navíc. Dvě věci ho dělají obzvlášť užitečným:

- **Kopíruje jako `memmove`.** Sám pozná, jestli se zdroj a cíl překrývají, a zvolí správný směr, takže posunutí celé plochy o řádek (překrývající se kopie) vyjde vždy správně. To je přesně ten **hardwarový scroll**, který by na procesoru byl neúnosně drahý.
- **Umí rychlou výplň** — vysype blok jedním bajtem (třeba vymaže obrazovku mezerami).

Blitter je důvod, proč na tomhle malém stroji plynule běží scrollovací hry. Nejlepší ukázkou je `examples/games/cave-runner.asm`, kterou rozebereme v kapitole 20.

Bitmapové režimy

Text je rychlý, ale kreslit se s ním dá jen po celých znacích. Bity 1–2 registru `VID_CTRL` proto přepínají **režim obrazu**: režim 1 je bitmapa **160×100 v šestnácti barvách**, režim 2 bitmapa **320×200 dvoubarevná**. Oba čtou tentýž **framebuffer** — 8000 bajtů od `0x9000`. Přepnutí režimu paměť nemaže, jen ji video zařízení začne jinak vykládat.

V režimu 1 nese jeden bajt **dva pixely** vedle sebe: horní půlbajt je levý pixel, dolní pravý, a půlbajt je rovnou číslo barvy z palety (adresa bajtu je `0x9000 + y·80 + x/2`). Nakreslit pixel tedy znamená bajt přečíst, vyměnit správný půlbajt a zapsat ho zpět — druhý pixel v bajtu přitom zůstane nedotčený. V režimu 2 nese bajt **osm pixelů** (nejvyšší bit je vlevo, adresa `0x9000 + y·40 + x/8`): rozsvícený bit má barvu „inkoustu„, zhasnutý barvu „papíru“ a obě volí registr `VID_MCOLOR` (`0xFF57`) stejným formátem jako barevná RAM. Z toho plyne pěkný trik: přepsáním jediného registru přebarvíš celou obrazovku, aniž se framebufferu dotkneš.

Vyzkoušej `examples/graphics/16-bitmap.asm` (oba režimy za sebou) a `19-starfield.asm` — letící hvězdné pole, kde se každá hvězda každý snímek smaže, posune a nakreslí znovu právě takovou výměnou půlbajtu.

Jemný scroll

Blitter posouvá obraz po celých buňkách — skokem o osm pixelů. Registry `VID_SCROLLX` a `VID_SCROLLY` (`0xFF55` / `56`) doplňují zbytek: posunou **vyčítání** obrazu o 0–7 pixelů (dokola, co vypadne vlevo, objeví se vpravo), aniž se v paměti pohne jediný bajt. Klasický recept na plynulý pohyb zní: sedmkrát zvyš `VID_SCROLLX` o jedničku, a v osmém snímku registr vynuluj a **současně** posuň data blitterem o celou buňku. Oba pohyby se přesně vyruší a oko vidí souvislý posun po jednom pixelu — přesně tak scrollovaly titulky osmibitových her. Ukázka: `examples/graphics/17-fine-scroll.asm`.

Sprity

Poslední kousek skládačky jsou **sprity** — malé obrázky, které plují **nad** obrazem, aniž by ho ničily. Stroj jich má osm, každý 16×16 pixelů (v souřadnicích režimu 1, na

obrazovce tedy 32×32 bodů), s barvami z palety; barva 0 je **průhledná**, takže sprite nemusí být čtvercový. Kde se sprity překrývají, vyhrává ten s nižším číslem.

Jako všechno u obrazu i sprity žijí v paměti: **vzory** (obrázky, 32 × 128 bajtů, dva pixely na bajt jako v režimu 1) od `0xB000` a **atributová tabulka** od `0xAFE0` — čtyři bajty na sprite: souřadnice X a Y, číslo vzoru a příznaky pro zrcadlení. Jediný registr je `SPR_ENABLE` (`0xFF58`): bit i sprite i zobrazí. Pohyb spritu je pak obyčejný zápis dvou bajtů do atributů — žádné mazání a překreslování, o složení obrazu se stará hardware. Ukázkou je `examples/graphics/18-sprites.asm`: osm skákajících míčů nad textovou plochou, jeden z nich řízený klávesami.

> **Poznámka**
>
> Souřadnice spritů jsou posunuté o +16: hodnota 16 znamená „přilepený k okraji„, menší hodnoty sprite postupně vysouvají z obrazu ven. Díky tomu jeden bajt pokryje i částečné vyjetí za všechny čtyři okraje — bez záporných čísel, která by se do bajtu nevešla.

A ještě disk

Pro úplnost: stroj má i **disk** (zařízení SDC) — 256 bloků po 256 bajtech, dohromady 64 KB trvalého úložiště. Přes několik registrů mu zadáš „načti blok N do paměti„ nebo „ulož“, a data se přesunou sama (opět DMA, jako u blitteru). Na disku dokonce může být **zaváděcí sektor**, ze kterého stroj nastartuje malý operační systém „SAP-DOS„ — a vestavěný BASIC pak umí `SAVE` a `LOAD` programů. To už je ale nad rámec téhle knihy; kdo chce, najde detaily v dokumentaci.

Procesor, paměť, přerušení, vstup, zvuk i obraz — vypadá to jako kompletní počítač. Jedno zařízení nám ale ještě zbývá, a je jiné než všechna dosavadní: nemluví s tebou, ale s **jiným strojem**.

KAPITOLA 19

Sériová linka — dva stroje na kabelu

Všechna zařízení, která jsme dosud potkali, mluví **s tebou**: klávesnice čte tvoje prsty, terminál a obrazovka píšou tvým očím, reproduktor hraje tvým uším. Sériová linka je první zařízení, které mluví **s jiným strojem**. Za registrem `SER_DATA` nesedí člověk, ale druhý SAP-3 — a najednou jde psát chat, dvouhráčovou hru nebo přenos dat. Ze samotáře je počítač v síti. Sítí je zde kabel se dvěma konci, ale přesně takhle — po dvou drátech mezi dvěma stroji — začínal i internet.

V emulátoru je „kabelem„ druhý panel prohlížeče: otevři aplikaci ve dvou tabech, v obou přepni paměťový panel na záložku **Serial**, zadej stejné jméno kanálu a klikni na **Connect**. Rozsvítí se kontrolka LINK a od té chvíle jsou oba stroje spojené, jako bys mezi nimi natáhl null-modemový kabel.

Tři registry a dvě fronty

Zařízení má nejmenší registrovou sadu ze všech — dva pracovní registry a jeden příkazový:

Adresa	Registr	Význam
`0xFF65`	`SER_DATA`	čtení vydá nejstarší přijatý bajt; zápis zařadí bajt k odeslání
`0xFF66`	`SER_STATUS`	bit 0 přijato · bit 1 lze vysílat · bit 2/3 ztráta dat · bit 4 kabel
`0xFF67`	`SER_CTRL`	zápis bitu 0 vyprázdní příjem, bitu 1 vysílání

Nový je tu jeden pojem: **FIFO** (first in, first out — fronta). Na každém směru sedí šestnáctibajtová schránka. Co druhá strana pošle, čeká v **přijímací** frontě, dokud si to program nevyzvedne čtením `SER_DATA`; co program zapíše, čeká ve **vysílací** frontě, než to linka odešle. Fronty oba stroje oddělují časově — nemusí běžet stejně rychle ani číst přesně ve chvíli, kdy druhý píše.

Vysílač, který nestíhá

Krátké výpisy na terminál vypadají okamžité; sériová linka tuhle iluzi nedovolí vůbec — odeslání jednoho bajtu trvá **nejméně 128 T-stavů** (na vysokých taktech víc, aby linka nikdy nepředběhla reálnou přenosovou rychlost). Proč? Skutečný UART (čip, který sériovou linku obsluhuje) posílá bajt po jednotlivých bitech pevným tempem,

třeba 115 200 bitů za sekundu — a bajt tak na drátě stráví skoro desetinu milisekundy. Náš stroj tohle tempo věrně napodobuje. Zapsat můžeš rychleji, než linka odesílá; proto je tu fronta — a proto se před **každým** zápisem ptej bitu 1:

```
SER_DATA   EQU 0xFF65
SER_STATUS EQU 0xFF66

posli:                      ; A = bajt k odeslání
    STA 0x10                ; odlož si ho - polling přepíše A
.cekej:
    LDA SER_STATUS
    AND #0x02               ; je ve vysílací frontě místo?
    JZ .cekej               ; ne -> počkej, až linka odešle
    LDA 0x10
    STA SER_DATA            ; ano -> zařaď k odeslání
    RET
```

Co se stane, když polling vynecháš a napereš do plné fronty sedmnáctý bajt? Bajt se **ztratí** — a jediným svědkem je „lepkavý„ bit 3 v `SER_STATUS`, který zůstane nastavený, dokud si status nepřečteš. Stejný strážce (bit 2) hlídá i příjem: když druhá strana chrlí a ty nečteš, přeteče přijímací fronta. Ztráta dat u linky není chyba programu, ale **vlastnost světa** — dráty jsou pomalejší než procesory. Dobrý program s ní počítá.

> **Tip**
>
> Tenhle vzor — „než zapíšeš, zeptej se, jestli smíš„ — je nejstarší idiom systémového programování. Stejnou smyčku nad stavovým bitem vysílače psali programátoři sálových počítačů, osmibitů i dnešních mikrokontrolérů. I terminál našeho stroje má stejnou frontu a stejný bit — jen je díky své hloubce vidět až u opravdu dlouhých výpisů (kapitola 16).

Příjem: polling, nebo přerušení

Čtení je zrcadlem klávesnice: bit 0 v `SER_STATUS` říká „něco přišlo„, čtení `SER_DATA` vydá nejstarší bajt. A stejně jako klávesnice má i linka svůj drát do řadiče přerušení — v masce `IRQ_ENABLE` jí patří **bit 2**. Je to úrovňový zdroj: hlásí se, dokud je v přijímací frontě aspoň bajt, a zhasne, až frontu vyprázdníš. Obsluha proto čte v cyklu, dokud bit 0 nezhasne — jedno přerušení klidně obslouží několik bajtů najednou.

Přesně tak funguje ukázkový chat (`examples/serial/01-serial-chat.asm`): hlavní smyčka polluje klávesnici a každou klávesu pošle výše uvedenou rutinou `posli`,

zatímco příjem obstarává obsluha přerušení, která frontu vysype na terminál. Napiš písmeno v jednom tabu — a vyskočí v druhém.

Bajty nestačí: potřebuješ protokol

Chat je snadný, protože proud bajtů **je** rovnou zpráva. Jakmile ale posíláš víc druhů údajů, narazíš na otázku, kterou linka za tebe nevyřeší: přišel bajt `0x0B` — je to pozice pálky, kus skóre, nebo půlka souřadnice míče? Odpovědí je **protokol**: dohoda obou stran o tom, co bajty znamenají a v jakém pořadí jdou. Je to totéž, co dělá kódování instrukcí z kapitoly 8 — dát holým bajtům význam — jen tentokrát pravidla píšeš ty.

Ukázková hra `examples/serial/02-serial-pong.asm` předvádí klasické řešení z dvouhráčových her. Jeden stroj je **hostitel** a vlastní pravdu: simuluje míč, počítá skóre a každý snímek odvysílá pětibajtový rámec — synchronizační bajt `0xFF`, pozici míče, svou pálku a skóre. Druhý stroj je **klient**: posílá jediný bajt (řádek své pálky) a kreslí, co mu přijde. Kdyby se kterýkoli bajt ztratil, klient prostě zahodí vše až do dalšího `0xFF` a chytí se znovu — rámec s hlavičkou dělá z křehkého proudu bajtů samoopravný přenos. Hostitel je zároveň metronom: klient nečeká na vsync, ale na další rámec.

> **Poznámka**
>
> Když stejný kanál připojíš ve **třech** tabech, vznikne „párty linka„: každý slyší bajty všech ostatních. Dvoubodové spojení je jen dohoda, ne vlastnost hardwaru — skutečné sběrnice jako RS-485 nebo CAN takhle schválně fungují. Pro dvě oddělené hry prostě zvol dvě jména kanálů.

Kabel a stroj času

Emulátor umí krokovat i zpět (tlačítko *Step Back*) — a fronty, rozesílaný bajt i lepkavé bity se vracejí přesně, protože jsou součástí stavu stroje. **Druhý stroj** se ale s tebou nevrací: co jednou odešlo drátem, nejde odposlat, a když si úsek po návratu přehraješ znovu, protistrana dostane bajty podruhé. Zní to jako drobnost, ale je to hluboká vlastnost všech distribuovaných systémů: každý stroj vládne jen svému času.

Tím je zvěřinec kompletní: procesor, paměť, přerušení, klávesnice, čas, zvuk, obraz — a kabel ven. V poslední části knihy všechno poskládáme dohromady.

ČÁST V

Poskládáme to dohromady

KAPITOLA 20

Případová studie: rozebíráme hru

Dosud jsme každou část procesoru probírali zvlášť. Teď je čas vidět, jak *spolupracují* — na skutečné, hratelné hře. Vezmeme **Cave Runner** (`examples/games/cave-runner.asm`): loď letí vzhůru klikatou jeskyní, stěny se sunou dolů, hráč uhýbá doleva a doprava. Rozebereme, jak se v jednom programu potká všechno, co jsme se naučili — obraz, blitter, gamepad, zvuk i časování.

Hra v jedné větě

Každý snímek se celá plocha posune o řádek dolů, nahoře přibude nový kousek stěny s mezerou, hráč se posune podle gamepadu, zkontroluje se náraz a celé to běží v rytmu obrazu s tříhlasým doprovodem. Zní to jako spousta práce — a je. Kouzlo je v tom, jak se ta práce rozdělí mezi procesor a specializovaný hardware.

Herní smyčka

Srdcem každé hry je **herní smyčka** — jeden snímek, pořád dokola:

```
opakuj každý snímek:
    1. počkej na vsync (VID_STATUS)      -- srovnej tempo na
rytmus obrazu
    2. přečti gamepad (PAD1)             -- kam chce hráč
    3. posuň plochu o řádek (blitter)    -- hardwarový scroll
    4. vykresli nový horní řádek         -- stěna + mezera
    5. posuň loď, zkontroluj náraz
    6. odbav hudbu (jeden krok patternu)
    7. aktualizuj skóre
```

Každý z těch kroků je přímé použití nějaké kapitoly z této knihy. Projděme si ty nejzajímavější.

Krok 3: proč vůbec blitter existuje

Tady je jádro celé hry — a nejlepší ukázka, proč stroj potřebuje blitter. Plocha, která se scrolluje, má 22 řádků po 40 znacích, tedy **880 bajtů, které se každý snímek musí posunout o řádek dolů.**

Zkusme to spočítat na holém procesoru. Kopie 880 bajtů znamená smyčku o zhruba 880 průchodech, každý o několika instrukcích — dohromady kolem pěti tisíc instrukcí.

Jenže rozpočet jednoho snímku je řádově *tisíc taktů*. Kopie by ho překročila mnohonásobně a hra by se plazila.

Blitter to udělá jako jediné překrývající se kopírování — za nula taktů navíc. Nastaví se zdroj (`SCREEN`), cíl (`SCREEN+40`, o řádek níž), délka (880) a spustí příkaz COPY. A protože blitter kopíruje jako `memmove` (kapitola 18), sám pozná, že se cíl s zdrojem překrývají, a zvolí správný směr, aby se data nepřepsala. Nový horní řádek se pak „vyřízne„ dvěma rychlými výplněmi (FILL): jednou stěnou, jednou mezerou. **Veškerá hromadná práce s obrazem je tak hardwarové DMA** — procesoru zbude čas na logiku hry.

> **Poznámka**
>
> Tohle je obecný vzor, ne trik jedné hry: kdykoli potřebuješ přesunout hodně dat rychle, hledej specializovaný hardware. Blitter na osmibitovém stroji, grafická karta v dnešním počítači — je to táž myšlenka „nenech to dělat procesor„ jen v jiném měřítku.

Krok 2 a 5: vstup a pohyb

Vstup je jednoduchý: `LDA PAD1` přečte bitmapu držených tlačítek (kapitola 16), maskou `AND` se otestuje „drží se doleva?„ a podle toho se sloupec lodi zvýší nebo sníží. Protože je gamepad **úrovňový** registr (čtení ho nemaže), reaguje hra plynule na držené tlačítko — přesně to, co plošinovka potřebuje, a co by klávesnice (jedna klávesa, pak zapomenout) nezvládla.

Náraz se pozná pohledem do obrazové paměti: na pozici lodi se přečte, co tam je, a jestli je to stěna, je konec. Data hry *jsou* obraz — žádná zvláštní kolizní mapa se nevede.

Krok 1 a 6: čas a hudba na jednom rytmu

Aby hra běžela stejně rychle na pomalém i rychlém stroji, opírá se o **vsync** (kapitola 16 a 18): na začátku každého snímku počká na tik `VID_STATUS`. Tím je tempo pevně svázané s obrazem, ne s taktovací frekvencí.

A na *stejném* rytmu jede i hudba. Všechny tři zvukové kanály (kapitola 17) hrají opakující se pattern — obdélníkový vedoucí hlas, trojúhelníkový bas, šumové bicí — a v každém snímku se posunou o krok. Když hráč narazí, dva kanály se přepnou na klesající „zavrčení„ a šumové zadunění exploze. Hudba i hra tak tikají z jednoho zdroje času a nikdy se nerozejdou.

Krok 4 a 7: náhoda a skóre

Klikatost jeskyně řídí **generátor náhodných čísel** (`RND`, kapitola 6): mezera se snímek co snímek lehce posouvá a pomalu zužuje. A protože generátor umí **pevné semínko** (`--seed`), je průběh hry *reprodukovatelný* — se stejným semínkem projde jeskyně pokaždé stejně, což je k nezaplacení při ladění. Skóre je vzdálenost, kterou loď uletí, počítaná po snímcích a vypisovaná do řádku HUD dole na obrazovce.

Co si z toho odnést

Cave Runner není nic jiného než všechno z této knihy najednou. Adresové režimy a smyčky procházejí obrazovou paměť; zásobník drží podprogramy; přerušení a čítače dávají rytmus; paměťově mapovaná zařízení tvoří vstup, zvuk i obraz; a specializovaný hardware (blitter) uklidí těžkou práci, na kterou by procesor sám nestačil. Právě tahle **dělba práce** mezi obecný procesor a specializované obvody je princip, na kterém stojí každý počítač — od osmibitové jeskyně až po tvůj telefon. A přesně k tomu srovnání se dostaneme v poslední kapitole.

KAPITOLA 21

Od SAP-3 k moderním procesorům

Došli jsme na konec prohlídky. Máš v hlavě celý počítač — od jediného bitu po hru s hudbou. Poslední otázka zní: jak daleko je odsud k procesoru, na kterém běží tvůj prohlížeč? Odpověď je překvapivá: **mnohem blíž, než by ses čekal.**

Co zůstává stejné

Všechno, co jsi v této knize viděl, existuje — rozpoznatelně — i v tom velkém stroji. Moderní procesor pořád dělá **fetch, decode, execute** (kapitola 7). Pořád má **registry, ALU, příznaky, zásobník** i **paměťově mapovaná zařízení a přerušení**. Pořád v základu mluví jazykem **řídicích signálů** — v každém taktu se rozsvítí kombinace přepínačů a bajty putují po sběrnicích. Kdybys uměl zpomalit procesor v mobilu na pár taktů za sekundu, poznal bys v něm skoro všechno z těchhle stránek.

Co přibylo: dělat totéž rychleji

Rozdíl není v *čem*, ale v *kolika* a *jak rychle*. Skoro každé zlepšení za posledních padesát let je způsob, jak provést *tentýž* cyklus rychleji:

- **Pipelining (zřetězení)** — načítat příští instrukci už během vykonávání té současné, jako na montážní lince. Zárodek téhle myšlenky už znáš: každá instrukce SAP-3/16 začíná stejnými kroky fetche (kapitola 7), a právě tahle pravidelnost dovoluje o překrývání vůbec uvažovat.
- **Cache** — malé rychlé paměti, které skrývají pomalost velké RAM. Je to kompromis „nultá stránka versus vzdálená paměť„ z kapitoly 14, jen zautomatizovaný a navrstvený do několika úrovní.
- **Predikce větvení** — hádat, kudy podmíněný skok půjde, ještě než jsou známé příznaky, aby pipeline zůstala plná. (Vzpomeň si na kapitolu 10 — právě větvení je to, co pipeline nejvíc komplikuje.)
- **Superskalární a mimopořadové vykonávání** — několik ALU zpracovává víc instrukcí najednou, kdykoli jsou jejich operandy nezávislé.
- **Všechno širší** — 64bitové registry místo osmi, gigabajty adresního prostoru místo 64 KB a virtuální paměť, která překládá každou adresu za běhu (nepřímost vektorů a ukazatelů z kapitol 14 a 8, zobecněná do hardwaru).

Nic z toho ale základy *nenahrazuje* — všechno na nich staví. Když krokuješ program v SAP-3/16 a sleduješ bajty putovat po sběrnici pod taktovkou mikrokódu, díváš se

na skutečnou věc. Zbytek počítačové architektury je tenhle obrázek, jen ve větším měřítku.

Od emulátoru ke křemíku

A ještě jedna věc na závěr, která ukazuje, že SAP-3/16 není jen simulace na obrazovce. Vzpomeň si na **mikrokód** z kapitoly 7 — tabulku „které signály v kterém T-stavu„. Tahle tabulka se dá zabalit do **paměti ROM**, kde každé slovo je jeden takt řídicích signálů. A takovou ROM lze vypálit do skutečného programovatelného hradlového pole — **FPGA**.

To má hezký důsledek: *tentýž* mikrokód, který v emulátoru řídí animované schéma, může řídit i reálný obvod na desce velikosti krabičky od sirek (třeba populární FPGA Tang Nano). Emulátor a hardware pak prokazatelně vykonávají stejný stroj — ne dvě podobné věci, ale doslova tutéž definici procesoru, jednou v prohlížeči a jednou v křemíku. Osmibitový počítač z této knihy tak není slepá ulička ani hračka na jedno použití; je to úplný, funkční návrh procesoru, který můžeš držet v ruce.

Kam dál

Skoro všechno v SAP-3/16 je vzaté z procesoru 6502 — srdce Commodore 64, Apple II i prvního Nintenda. To znamená, že tahle kniha byla i přípravou: cokoli si teď přečteš o 6502, Z80 nebo 8080, ti bude povědomé. Registry, adresové režimy, nultá stránka, přerušení, vektory — to všechno tam najdeš, jen pod trochu jinými jmény.

A hlavně — teď už víš, co se děje, když počítač počítá. Není to magie. Je to hrstka registrů, jedna sběrnice, aritmetická jednotka a neúnavná smyčka fetch-decode-execute, dirigovaná mikrokódem. Zbytek je jen měřítko. Když příště stiskneš klávesu a na obrazovce se objeví písmeno, budeš přesně vědět, jaká cesta bajtů se právě odehrála — protože jsi ji viděl celou, takt po taktu, na počítači, který se vejde do dlaně.

Přílohy

PŘÍLOHA A

Kompletní tabulka instrukcí

Opcode je jeden bajt rozdělený na tři pole `gg|ooo|mmm` (kapitola 7): `gg` = skupina, `ooo` = operace, `mmm` = adresový režim. U skupin 01 a 10 se opcode skládá jako `základ_skupiny + ooo × 8 + mmm`.

Adresové režimy (pole `mmm`)

`mmm`	Zápis	Význam
`000`	`#n`	immediate — operand je hodnota
`001`	`addr8`	nultá stránka
`010`	`addr16`	absolutní
`011`	`addr16,X`	absolutní indexováno X
`100`	`addr16,Y`	absolutní indexováno Y
`101`	`(addr8)`	nepřímý (ukazatel v nulté stránce)
`110`	`addr8,X`	nultá stránka indexováno X (wrapuje)
`111`	`(addr8),Y`	nepřímý indexovaný Y

Skupina 00 — implied (1 bajt)

Op	Mn.	Příznaky	Popis
`0x01`	`NOP`	—	nedělá nic
`0x02`	`HLT`	—	zastav (probudí přerušení)
`0x03`	`OUT`	—	OUT := A
`0x04`	`CLC`	C	C := 0
`0x05`	`SEC`	C	C := 1
`0x06`	`EI`	I	povol přerušení
`0x07`	`DI`	I	zakaž přerušení
`0x08`	`INC`	Z N	A := A + 1
`0x09`	`DEC`	Z N	A := A − 1
`0x0A`	`SHL`	C Z N	posun vlevo, C := bit 7

Op	Mn.	Příznaky	Popis
0x0B	SHR	C Z N	posun vpravo, C := bit 0
0x0C	ROL	C Z N	rotace vlevo přes C
0x0D	ROR	C Z N	rotace vpravo přes C
0x0E	NOT	Z N	A := ¬A
0x0F	RND	—	A := náhodný bajt
0x10 – 0x13	INX DEX INY DEY	Z N	±1 na X / Y
0x18 – 0x1B	TAX TXA TAY TYA	Z N	přesuny A↔X, A↔Y
0x1C 0x1D	TAB TBA	Z N	přesuny A↔B
0x1E 0x1F	TXYS TSXY	—	SP ↔ X:Y
0x20 – 0x25	PUSH POP PHX PLX PHY PLY	pop: Z N	zásobník A / X / Y
0x26 0x27	PHF PLF	PLF: vše	zásobník příznaků
0x28	MUL	C Z N	A×B ⟶ A (dolní), B (horní)
0x29	DIV	C Z N	A÷B ⟶ A (podíl), B (zbytek)
0x2A 0x2B	PHB PLB	PLB: Z N	zásobník B (záchrana bez zničení A)

Skupina 01 — load/store · Skupina 10 — ALU

Pole `ooo` volí operaci; opcode = základ + `ooo` × 8 + `mmm`. Load/store začíná na `0x40`, ALU na `0x80`.

ooo	Load/store	Popis
000	LDA	A := paměť
001	LDX	X := paměť
010	LDY	Y := paměť
011	CPX	porovnej X
100	STA	paměť := A
101	STX	paměť := X
110	STY	paměť := Y
111	CPY	porovnej Y

ooo	ALU	Popis
000	ADD	A := A + M
001	ADC	A := A + M + C
010	SUB	A := A − M
011	SBC	A := A − M − výpůjčka
100	AND	A := A & M
101	OR	A := A | M
110	XOR	A := A ^ M
111	CMP	porovnej A s M

Ne každý režim je legální pro každou instrukci (`STA` nemá immediate, `LDX` nemá `,X` atd.). ALU operace a `LDA` / `STA` nastavují příznaky podle kapitoly 6.

Skupina 11 — řízení toku a RMW

Op	Mn.	Popis
`0xC0` – `0xC7`	`JNZ JZ JNC JC JP JN JNV JV`	podmíněné skoky (aliasy `JNE JEQ JGE JLT`)
`0xC8`	`JMP addr16`	PC := adresa
`0xC9`	`JMP (zp)`	skok přes ukazatel (skokové tabulky)
`0xCA`	`CALL addr16`	ulož PC, skoč
`0xCB`	`RET`	návrat z podprogramu
`0xCC`	`RTI`	návrat z přerušení (obnoví i příznaky)
`0xCD`	`BRK`	softwarové přerušení (vektor `0xFFFA`)
`0xCE`	`CALL (zp)`	nepřímé volání přes ukazatel (dispatch tabulky)
`0xD0` – `0xD3`	`INC` / `DEC` zp/abs	±1 v paměti (Z N)
`0xD4` – `0xD7`	`INW` / `DEW` zp/abs	±1 na 16bitovém ukazateli (bez příznaků)
`0xD8` – `0xDF`	`SHL SHR ROL ROR` zp/abs	posuny/rotace přímo v paměti (C Z N) — vícebajtové posuny přes C
`0xE0` – `0xE3`	`JGT JLE JGES JLTS`	složené skoky: C∨Z (bezznaménkové >, ≤) a N⊕V (znaménkové ≥, <)
`0xE4` – `0xE6`	`BIT` #/zp/abs	test A & M — jen Z N, A přežije (B := M)
`0xE8` `0xE9`	`ADW` / `SBW` zp,#imm8	16bit word ± konstanta (bez příznaků, B := imm)
`0xEA` `0xEB`	`LXY` #imm16 · `SXY` zp	16bit konstanta do X:Y (X horní) · uložení X:Y jako LE word

Opcody `0xFD` (RESET) a `0xFE` (IRQ) jsou interní pseudo-opcody — nedají se z paměti vykonat a assembler je odmítne. `0x00` a `0xFF` jsou schválně pasti (trap).

PŘÍLOHA B

Mapa paměti a registry I/O

Mapa 64 KB paměti

Rozsah	Oblast	Popis
`0x0000` – `0x00FF`	Nultá stránka	rychlá data a sloty pro 16bitové ukazatele
`0x0100` – `0x01FF`	Stránka zásobníku	SP startuje na `0x0200`, roste dolů
`0x0200` – `0x7FFF`	Program + data	výchozí origin assembleru `0x0200`
`0x8000` – `0x83E7`	Znakový buffer	40×25, adresa = `0x8000 + y·40 + x`
`0x8400` – `0x8BFF`	Glyph RAM	znaková sada, 8 bajtů na glyf
`0x8C00` – `0x8FE7`	Barevná RAM	atribut na buňku: dolní půlbajt FG, horní BG
`0x9000` – `0xAF3F`	Bitmapový framebuffer	režimy 1 a 2, 8000 bajtů (řádek 80, resp. 40 bajtů)
`0xAFE0` – `0xAFFF`	Atributy spritů	8 spritů × 4 bajty: X+16, Y+16, vzor, příznaky
`0xB000` – `0xBFFF`	Vzory spritů	32 vzorů × 128 bajtů, 2 pixely na bajt
`0xC000` – `0xFEFF`	Datová RAM	velká pole
`0xFF00` – `0xFF7F`	I/O registry	paměťově mapovaná zařízení (níže)
`0xFF80` – `0xFFF9`	Systémová RAM	scratch obsluh, boot ROM
`0xFFFA` – `0xFFFB`	BRK vektor	obsluha softwarového přerušení
`0xFFFC` – `0xFFFD`	RESET vektor	start vykonávání
`0xFFFE` – `0xFFFF`	IRQ vektor	obsluha hardwarového přerušení

Paměťově mapovaná zařízení (`0xFF00` – `0xFF7F`)

Šestnáctibajtový krok na zařízení; číslo zařízení jsou adresní bity [6:4].

Adresa	Registr	Význam
`0xFF00`	`KBD_DATA`	přečti klávesu (čtení ji spotřebuje)
`0xFF01`	`KBD_STATUS`	bit 0 = čeká klávesa
`0xFF10`	`TERM_DATA`	zápis vypíše znak na terminál

Adresa	Registr	Význam
`0xFF11`	`TERM_STATUS`	bit 0 = lze vysílat (fronta není plná) · bit 1 = přetečení (čtení maže)
`0xFF20`	`AUD_CTRL`	bitmapa gate (bit N = kanál N)
`0xFF21` + `N·5`	`AUD_FREQ_LO/HI`	frekvence kanálu N v Hz (16 bitů)
`0xFF23` + `N·5`	`AUD_WAVE`	tvar vlny: 0 sinus … 4 šum
`0xFF24` + `N·5`	`AUD_AD`	attack (horní půlbajt) · decay (dolní)
`0xFF25` + `N·5`	`AUD_SR`	sustain (horní) · release (dolní)
`0xFF30`	`TMR_DIV`	perioda časovače v T-stavech
`0xFF31`	`TMR_CNT`	běžící čítač časovače
`0xFF32` – `0xFF35`	`CYC0` – `CYC3`	32bitový čítač taktů (čtení `CYC0` zamkne)
`0xFF36` – `0xFF37`	`UPT_LO/HI`	16bitový čítač snímků (uptime)
`0xFF38` – `0xFF39`	`CLK_TPMS`	taktů na milisekundu
`0xFF3A` – `0xFF3B`	`CLK_MS`	volně běžící čítač reálných milisekund (čtení LO zamkne oba) — základ reálného tempa
`0xFF40`	`IRQ_ENABLE`	bit 0 časovač, bit 1 klávesnice, bit 2 sériový příjem
`0xFF41`	`IRQ_STATUS`	čekající přerušení (čtení maže)
`0xFF50`	`VID_CTRL`	bit 0 zapni; bity 1–2 režim: 0 text, 1 bitmapa 160×100, 2 bitmapa 320×200
`0xFF51`	`VID_STATUS`	bit 0 = vsync tik (čtení maže)
`0xFF52` / `53`	`VID_CURX/Y`	pozice hardwarového kurzoru (jen text)
`0xFF54`	`VID_BORDER`	barva okraje (paleta &0x0F)
`0xFF55` / `56`	`VID_SCROLLX/Y`	jemný posun obrazu 0–7 pixelů (dokola)
`0xFF57`	`VID_MCOLOR`	barvy režimu 2: papír (horní půlbajt) · inkoust (dolní)
`0xFF58`	`SPR_ENABLE`	bit i zobrazí sprite i
`0xFF60`	`DSK_CMD`	1 = čti blok, 2 = zapiš blok
`0xFF61`	`DSK_STATUS`	bit 0 připraven (0 během přenosu, 2 ms), bit 1 chyba (čtení maže)
`0xFF62` – `0xFF64`	`DSK_BLK` / `ADDR`	číslo bloku a cílová adresa v RAM

Adresa	Registr	Význam
`0xFF65`	`SER_DATA`	sériová linka: čtení vydá přijatý bajt, zápis zařadí k odeslání
`0xFF66`	`SER_STATUS`	bit 0 přijato · 1 lze vysílat · 2/3 přetečení · 4 kabel
`0xFF67`	`SER_CTRL`	bit 0 vyprázdni příjem, bit 1 vysílání
`0xFF70` / `71`	`PAD1` / `PAD2`	gamepad: bitmapa držených tlačítek
`0xFF72` – `0xFF7A`	`BLT_*`	blitter: zdroj, cíl, délka, výplň, `BLT_CTRL` (1 kopie, 2 výplň)

Nenamapované I/O adresy čtou 0 a ignorují zápis. Assembler odmítne umístit kód nebo data do bloku `0xFF00` – `0xFF7F`.

PŘÍLOHA C

Řídicí signály

Mikrokód (kapitola 7) skládá každou instrukci z T-stavů a v každém T-stavu zvedne kombinaci **řídicích signálů**. Tady je jejich přehled roztříděný podle role. Ve schematickém pohledu emulátoru je uvidíš jako rozsvícené šipky.

Zdroje na sběrnici (kdo „mluví„)

V každém taktu smí sběrnici řídit nanejvýš jeden zdroj.

Signál	Vystaví na sběrnici
`RO`	bajt z paměti (z adresy v MAR)
`AO` `BO` `XO` `YO` `TO`	registr A / B / X / Y / TMP
`ORO` `OHO`	operandový registr (dolní / horní)
`CLO` `CHO`	dolní / horní bajt PC (push u CALL/IRQ)
`SLO` `SHO`	dolní / horní bajt SP
`FLO`	příznaky sbalené do stavového bajtu
`EO` `EXO`	výsledek ALU / druhotný výsledek (horní bajt MUL, zbytek DIV)

Cíle na sběrnici (kdo „poslouchá„)

Signál	Zachytí ze sběrnice
`RI`	zapiš do paměti (na adresu v MAR)
`II`	instrukční registr (opcode)
`AI` `BI` `XI` `YI` `TI`	registr A / B / X / Y / TMP
`ORI` `OHI`	operandový registr (dolní / horní)
`OI`	výstupní registr (instrukce OUT)
`MI` `MIL` `MIH`	MAR: nultá stránka / dolní / horní půlka
`JPL` `JPH`	dolní / horní půlka PC (pop u RET/RTI)
`SLI` `SHI`	dolní / horní půlka SP
`FLI`	obnov příznaky ze stavového bajtu

Adresová cesta a čítače

Signál	Akce
`PCM` `SPM`	MAR := PC / MAR := SP (16bitová zkratka)
`JPW`	PC := operand (absolutní skok)
`MRI`	MAR := MAR + 1 (druhý bajt ukazatele/vektoru)
`VRS` `VIR` `VBR`	MAR := báze vektoru RESET / IRQ / BRK
`CE`	inkrementuj PC
`SD` `SE`	dekrementuj / inkrementuj SP

Ovládání ALU a příznaků

Signál	Význam
`SU AND OR XOR NOT INC …`	volba operace ALU
`CI`	carry-in z příznaku C (pro ADC/SBC)
`ATM` `AOP`	vstupní muxy ALU: TMP místo A, operand místo B
`ACO` `AHC` `AHB`	záklopka mezibajtového přenosu adresy (skryté 16bitové sčítání)
`FI`	zachyť příznaky (z ALU, nebo Z/N ze sběrnice)
`CFC` `CFS`	smaž / nastav Carry
`EIF` `DIF`	povol / zakaž přerušení
`RND`	náhodný bajt do A
`HLT`	zastav hodiny

Pointa není se signály učit nazpaměť — je to, že *každá* instrukce, jakkoli abstraktně v assembleru vypadá, se rozloží do tohohle malého slovníku akcí na úrovni drátů.

PŘÍLOHA D

ASCII a paleta barev

Tabulka ASCII (výběr)

Text jsou čísla (kapitola 1). Tady jsou nejužitečnější kódy — řídicí znaky a tisknutelné rozsahy. Písmena a číslice jdou po sobě, takže `'0' + n` dá číslici n a `'A' + n` n-té písmeno.

Dec	Hex	Význam
8	`0x08`	backspace
9	`0x09`	tabulátor
10	`0x0A`	nový řádek (LF)
13	`0x0D`	návrat vozíku (CR)
32	`0x20`	mezera
48	`0x30`	číslice `0`
57	`0x39`	číslice `9`

Dec	Hex	Význam
65	`0x41`	`A`
90	`0x5A`	`Z`
97	`0x61`	`a`
122	`0x7A`	`z`
33	`0x21`	`!`
46	`0x2E`	`.`
63	`0x3F`	`?`

Tisknutelné znaky jsou v rozsahu 32–126. Hodnoty 0–31 jsou řídicí znaky; terminál reaguje na některé z nich (třeba 10 a 13). Vše nad 127 je mimo standardní ASCII.

Paleta barev

Šestnáct barev inspirovaných Commodore 64, sdílených barevnou RAM (dolní půlbajt = písmo, horní = pozadí) a registrem `VID_BORDER`. Hodnota `0x00` v barevné RAM znamená „výchozí vzhled„.

Č.	Barva	Č.	Barva
0	černá	8	oranžová
1	bílá	9	hnědá
2	červená	10	světle červená
3	azurová	11	tmavě šedá
4	fialová	12	středně šedá
5	zelená	13	světle zelená
6	modrá	14	světle modrá
7	žlutá	15	světle šedá

Barevný bajt buňky sestavíš jako `pozadí × 16 + písmo`. Například světlé písmo (1) na modrém pozadí (6) je `6 × 16 + 1 = 0x61`.

PŘÍLOHA E

Nástroje a další čtení

Spouštění programů z příkazové řádky

Programy z této knihy spustíš i mimo prohlížeč, nástrojem `npm run asm`:

```
npm run asm <soubor.asm> [-- volby]
```

Užitečné volby:

Volba	Význam
`--show-output`	ukaž jen výstup programu (terminál + OUT)
`--show-memory`	vypiš nenulové oblasti paměti (hex)
`--debug`	podrobný výpis registrů po každém taktu
`--max-cycles N`	strop počtu taktů (výchozí 10000)
`--seed N`	pevné semínko generátoru `RND` (reprodukovatelné běhy)
`--input "text\n"`	skriptovaný vstup z klávesnice

Semínko (`--seed`) je klíč k reprodukovatelnosti: se stejným semínkem dá `RND` pokaždé stejnou posloupnost, takže program s náhodou proběhne identicky — což jsme využili u všech ukázek v knize.

Ladicí nástroje v emulátoru

Webový emulátor přidává nástroje, které z něj dělají skvělou učební pomůcku:

- **Krokování** — spusť program po jedné instrukci a sleduj, jak se mění registry.
- **µStep** — krokuj po jednotlivých T-stavech a ve schématu pozoruj, které řídicí signály se rozsvěcují (kapitola 7).
- **Breakpointy** — zastav běh na konkrétní adrese.
- **Watchpointy** — hlídej registr nebo adresu v paměti a zastav, když se změní (nebo nabude dané hodnoty). Nejlepší nástroj na chytání záludných chyb, třeba přetečení zásobníku.
- **Cestování časem** — přehraj program dozadu a dopředu. Vidíš přesně, jak se stroj dostal do daného stavu — luxus, který skutečný hardware nenabízí.
- **Nastavitelné hodiny** — zpomal stroj na pár taktů za sekundu, ať vidíš každý bajt putovat po sběrnici.

Kam pokračovat

- **Specifikace ISA 3.0** (`docs/ISA3.md`) — autoritativní referenční popis architektury do posledního bitu; kdykoli je pochyba, platí ona.
- **Technický popis procesoru** (`docs/CPU.md`) — lidsky čtená reference s dalšími propočítanými příklady.
- **Ukázkové programy** (`examples/`) — desítky hotových programů: základy, matematika, algoritmy, grafika, zvuk, přerušení, terminál a hry. Nejlepší způsob, jak se učit, je číst a upravovat cizí kód.
- **Skutečné procesory** — skoro všechno v SAP-3/16 pochází z procesoru 6502. Cokoli si teď přečteš o 6502, Z80 nebo 8080, ti bude povědomé.

A hlavně — otevři emulátor, napiš vlastní program a dívej se, jak stroj ožívá. Teorie je mapa; teprve za volantem se z ní stane cesta.