SAP-3/16

Referenční příručka procesoru

Architektura, instrukční sada a periferie — každý bit, každý T-stav, každý registr na jednom místě.

Jan Müller

2026

Obsah

Přílohy

Předmluva

Tahle kniha vznikla z otázek, na které se špatně odpovídá zpaměti. Kolik T-stavů stojí `LDA (zp),Y`? Nastavuje `CMP` příznak přetečení? Co přesně znamená, že `DIV` dělením nulou „nespadne“? Který bit v `TERM_STATUS` je sticky a kdy se maže?

Odpovědi na ně existují — jsou v mikrokódu, v ALU, v obsluze periferií — ale dokud je člověk musí lovit ve zdrojácích, není to reference. Tahle kniha je sbírá na jedno místo.

Co to je za stroj

SAP-3/16 je osmibitový procesor se šestnáctibitovou adresní sběrnicí: osm bitů dat, plochý adresní prostor 0x0000–0xFFFF bez bankování, 179 legálních opkódů složených z 79 mnemonik a osmi adresních režimů. Vektorovaná přerušení, zásobník v hlavní paměti, indexované a nepřímé adresování.

Není to historický stroj, který by šlo koupit. Je to architektura navržená tak, aby na ní bylo **všechno vidět**: každý opkód se dá dekódovat z jeho bitů, každý přenos po sběrnici je krok, který jde odkrokovat, a boot je posloupnost obyčejných čtení z paměti. Existuje ve třech implementacích — emulátor v prohlížeči, jádro AGATA-1 v FPGA a port pro ESP32 — a všechny tři sdílejí jeden zdroj pravdy o mikrokódu.

Jak se tahle kniha liší od ostatních

V knihovně jsou tři knihy a každá dělá něco jiného:

Počítač do dlaně vypráví, jak počítač funguje. Začíná u jednoho bitu a končí u programů se zvukem a obrazem. Když je tohle tvůj první procesor, začni tam.

Postavíme si Arkanoid staví jednu konkrétní hru etapu po etapě. Učí, jak se v assembleru dělá **projekt** — ne jak funguje instrukce.

Referenční příručka tahle kniha. Nevypráví příběh; odpovídá na otázky. Je psaná tak, aby se v ní dalo listovat od prostředka.

Kdo umí programovat a chce znát tenhle stroj, může začít rovnou tady. Části I a II se dají číst popořadě jako výklad architektury; část III je slovník, do kterého se chodí pro jednu instrukci; část IV je popis periferií.

Konvence zápisu

Čísla se píšou tak, jak je bere assembler: `0x2A` šestnáctkově, `0b101010` dvojkově, `42` desítkově. V textu o kódování se bity píšou zleva doprava od nejvyššího, tedy `bit 7` je nejvyšší.

Adresní režimy mají v celé knize stejné zkratky:

Zápis	Význam
`#n`	přímý operand (immediate) — hodnota je v instrukci
`zp`	nultá stránka — jednobajtová adresa 0x0000–0x00FF
`abs`	absolutní — dvoubajtová adresa kdekoli v paměti
`abs,X` / `abs,Y`	absolutní indexovaná registrem X, resp. Y
`(zp)`	nepřímá — v nulté stránce je uložen ukazatel
`zp,X`	indexovaná v nulté stránce, sčítání se zabaluje uvnitř stránky
`(zp),Y`	nepřímá indexovaná — k ukazateli z nulté stránky se přičte Y

Příznaky se uvádějí jen ty, které instrukce **definuje**. Prázdné políčko znamená, že příznak zůstává beze změny — ne že je vynulovaný. Tenhle rozdíl je v SAP-3/16 zásadní: logické operace nechávají C na pokoji právě proto, aby šlo mezi porovnáním a skokem maskovat bity.

Mikrokód se zapisuje v hranatých závorkách, jeden pár na jeden T-stav, signály uvnitř oddělené čárkou; číslo v kulaté závorce na konci je celkový počet T-stavů včetně fetche:

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

Otazník za závorkou označuje krok, který se přeskočí, když neplatí podmínka (používají ho podmíněné skoky).

Odkud se berou čísla

Každá tabulka opkódů, každý počet T-stavů, každá množina příznaků a každá adresa registru v této knize je **generovaná z implementace procesoru**, ne opsaná. Sazba je ruční, čísla ne. Kontrola, že se generovaná data shodují se zdrojem, běží při každém sestavení projektu — kniha se s procesorem nemůže rozejít, aniž by to shodilo build.

To má jeden praktický důsledek pro čtenáře: pokud v knize najdeš číslo, které neodpovídá tomu, co dělá tvůj stroj, není to překlep v knize. Je to buď jiná verze ISA, nebo chyba, kterou stojí za to nahlásit.

Kniha popisuje **ISA 3.1**.

Co kniha nepokrývá

Záměrně tu nenajdeš: Tiny BASIC, standardní knihovnu, SAP-DOS jako operační systém (jen boot a formát sektoru, protože to je součást hardwarového kontraktu),

návody na psaní her a detaily HDL nebo desek plošných spojů. Poslední kapitola o železe je přehledová — ne servisní manuál.

ČÁST I

Architektura

KAPITOLA 1

Přehled architektury

SAP-3/16 je osmibitový procesor se šestnáctibitovou adresní sběrnicí. Ta jediná věta určuje skoro všechno ostatní: data se hýbou po bajtech, adresy mají dva bajty, a protože po datové sběrnici projde vždycky jen jeden bajt naráz, každá šestnáctibitová adresa stojí procesor T-stavy navíc. Na téhle asymetrii stojí celá mikroarchitektura — a je vidět v tabulce délek instrukcí v každém hesle slovníku.

Klíčové parametry

Parametr	Hodnota
Šířka dat	8 bitů
Šířka adresy	16 bitů, plochých 64 KB bez bankování
Registry	A, B, X, Y (8 b) + SP, PC (16 b) + OUT
Příznaky	C, Z, N, V, I
Legálních opkódů	179 z 256 (75 volných slotů)
Mnemonik	79
Adresních režimů	8
Délka instrukce	1 až 3 bajty
Zásobník	v hlavní paměti, stránka 0x0100–0x01FF
Přerušení	vektorovaná, tři zdroje + softwarové `BRK`
Vektory	0xFFFA BRK, 0xFFFC RESET, 0xFFFE IRQ
Periferie	0xFF00–0xFF7F, stride 16 bajtů na zařízení
Videopaměť	0x8000–0xBFFF

Tabulka 1. Souhrn architektury SAP-3/16, ISA 3.1.

Tři designové cíle

Architektura má tři cíle a jsou seřazené — když si odporují, vyhrává ten dřívější. Většina rozhodnutí, která se dál v knize můžou zdát zvláštní, jde vysvětlit tímto pořadím.

1. **Srozumitelnost** každý opkód musí být dekódovatelný ze svých bitů, každý přenos adresy musí být vidět na schématu a boot musí jít odkrokovat po jednotlivých

čteních z paměti. Proto neexistují prefixové bajty, proto se reset chová jako obyčejná instrukce a proto má i `NOP` v mikrokódu jeden prázdný krok navíc.

2. **Pravidelnost** adresní režimy jsou ortogonální, kódování je systematické, výjimky jsou vzácné. Instrukce se neskládá z tabulky, ale ze vzorce `gg|ooo|mmm` — skupina, operace, režim.
3. **Dostatečnost** programy o velikosti jednotek kilobajtů se musí psát příjemně. Odtud nultá stránka jako soubor ukazatelů, `MUL` / `DIV` v hardwaru, šestnáctibitové operace nad slovem v nulté stránce a blitter.

> **Poznámka**
>
> Cíle jsou seřazené, ne vyvážené. Kdykoli by šlo ušetřit T-stavy za cenu toho, že by přestalo být vidět, co se v procesoru děje, rozhodlo se ve prospěch viditelnosti. Rychlost je až čtvrtá v pořadí a je to vědomá cena.

Odkud se to vzalo

Jméno je poděkování: SAP — **Simple As Possible** — je řada školních procesorů z Malvinovy učebnice *Digital Computer Electronics*. SAP-1 umí sečíst dvě čísla, SAP-2 přidá skoky a vstup/výstup, SAP-3 registry a zásobník. Jsou to stroje navržené k tomu, aby se daly nakreslit na tabuli.

SAP-3/16 z téhle tradice bere přístup, ne konkrétní zapojení. Přidává druhý bajt k adresám (odtud „/16“), skutečnou paměťovou mapu s periferiemi, vektorovaná přerušení a systematické kódování instrukcí. Z osmibitové klasiky si půjčuje to, co se osvědčilo: nultou stránku a post-indexované nepřímé adresování od 6502, konvenci `TXS` / `TSX` pro práci se zásobníkovým ukazatelem, plnou sestupnou strukturu zásobníku.

Co si nepůjčuje, je nepravidelnost. U 6502 se člověk musí naučit, které instrukce umí které režimy, protože pro to neexistuje pravidlo. Tady pravidlo existuje a je vidět v opkódu.

Programovací model v kostce

Podrobnosti jsou v kapitole o registrech a příznacích; tady je jen mapa terénu, aby dávaly smysl příklady v následujících kapitolách.

Programátor vidí čtyři osmibitové registry. **A** je akumulátor — cíl skoro každé aritmetiky. **B** je druhý operand ALU. **X** a **Y** jsou indexy. K tomu dva šestnáctibitové: **PC** a **SP**.

> **Pozor**
>
> **B** není registr k odkládání hodnot. Každá binární operace ALU si do něj natáhne svůj operand, takže cokoli, co si tam program uložil, zmizí při nejbližším `ADD`, `CMP` nebo `AND`. Je to nejčastější zdroj překvapení u lidí, kteří přicházejí od jiných osmibitů — a důvod, proč existuje dvojice `PHB` / `PLB`.

Uvnitř procesoru je ještě několik registrů, které nemá žádná instrukce jak adresovat, ale bez kterých by nešlo pochopit mikrokód: **IR** drží právě prováděný opkód, **OP** a **OPH** operandové bajty, **MAR** adresu, na kterou se právě sahá do paměti, **TMP** je pracovní vstup ALU a **addrCarry** jednobitová záklopka pro přenos mezi bajty adresy. Ta poslední dvojice je důvod, proč indexované adresování nezničí žádný uživatelský registr.

Tři implementace, jedno chování

Stejný binární program běží na třech různých strojích:

Emulátor TypeScript, běží v prohlížeči na sap-3.com. Umí krokovat po T-stavech, vracet čas, ukazovat řídicí signály a obsah sběrnic.

AGATA-1 jádro v jazyce Verilog na FPGA (Sipeed Tang Nano 20K), srdce konzole MAXIK. HDMI výstup, SD karta, porty pro NES ovladače, turbo režim.

Port pro ESP32 tentýž procesor v C++ na mikrokontroléru s malým displejem.

Nejsou to tři nezávislé implementace téhož popisu — to by se rozešly. Mikrokód má jediný zdroj a HDL i firmware ho dostávají **vygenerovaný**; ALU se v simulaci ověřuje proti vyčerpávající sadě vektorů spočítané z emulátoru a celé programy se přehrávají proti referenčním stopám T-stav po T-stavu. Rovnost „prohlížeč se chová jako deska“ je tím pádem artefakt sestavení, ne příslib v dokumentaci.

> **Poznámka**
>
> Tenhle detail má důsledek pro celou knihu: počty T-stavů, které tu najdeš, nejsou orientační. Jsou to tytéž počty, které naměříš na desce.

Právě proto v architektuře chybí věci, které by se u procesoru s takovou frekvencí jinak čekaly — cache, predikce skoků, přeuspořádání instrukcí. Žádná z nich není vynechaná kvůli složitosti; jsou vynechané proto, že by rozbily determinismus, na kterém stojí krokování, návrat v čase i referenční stopy.

Kudy dál

Zbytek části I jde po vrstvách: paměť (co je kde), kódování (jak vypadá instrukce), datová cesta (kudy tečou bajty), instrukční cyklus (v jakém pořadí) a přerušení (co se stane, když do toho někdo skočí). Kdo chce jen programovat, může skočit rovnou na část II; kdo hledá jednu instrukci, patří do slovníku v části III.

KAPITOLA 2

Paměťový model

SAP-3/16 vidí jediný plochý adresní prostor o velikosti 64 KB. Žádné bankování, žádné okna, žádné oddělené prostory pro kód a data. Adresa je šestnáctibitové číslo a znamená vždycky totéž místo.

Celý prostor je obyčejná RAM s jedinou výjimkou: blok 0xFF00–0xFF7F, který procesor zachytává dřív, než se čtení nebo zápis dostane do paměti, a obsluhuje ho jako periferii. Všechno ostatní — včetně videopaměti — se chová jako paměť. Zápis do VRAM je normální zápis do RAM: je vidět v žurnálu, dá se na něj dát watchpoint a vrátit se před něj v čase. Video si obsah jen **čte**.

Mapa paměti

Adresa	Velikost	Oblast
`0x0000–0x00FF`	256 B	Nultá stránka — rychlý přístup a soubor ukazatelů
`0x0100–0x01FF`	256 B	Zásobník
`0x0200–0x7FFF`	32256 B	Program a data
`0x8000–0x83E7`	1000 B	Znakový buffer (40×25)
`0x83E8–0x83FF`	24 B	rezervováno
`0x8400–0x8BFF`	2048 B	Glyph RAM (8 bajtů na znak)
`0x8C00–0x8FE7`	1000 B	Barevná RAM (atribut na buňku)
`0x8FE8–0x8FFF`	24 B	rezervováno
`0x9000–0xAF3F`	8000 B	Bitmapový framebuffer
`0xAF40–0xAFDF`	160 B	rezervováno
`0xAFE0–0xAFFF`	32 B	Atributy spritů (8 × 4 bajty)
`0xB000–0xBFFF`	4096 B	Vzory spritů (32 × 128 bajtů)
`0xC000–0xFEFF`	16128 B	Datová RAM — velká pole, halda
`0xFF00–0xFF7F`	128 B	Registry periferií (MMIO)
`0xFF80–0xFFF9`	122 B	Systémová RAM — scratch pro ISR, domov boot ROM
`0xFFFA–0xFFFF`	6 B	Vektory BRK / RESET / IRQ

Obrázek 1. Rozdělení 64 KB adresního prostoru. Součet všech oblastí je celý prostor — řádky označené jako rezervované jsou díry, které dnes nikdo nepoužívá.

Nultá stránka

Prvních 256 bajtů má dvě zvláštní vlastnosti a obě stojí za to znát, protože ovlivňují, jak se na tomhle stroji píšou programy.

Je levnější. Instrukce s jednobajtovou adresou je o bajt kratší a o tři T-stavy rychlejší než tatáž instrukce s adresou dvoubajtovou. Rozdíl není kosmetický: v těsné smyčce je to desítky procent.

Je to jediné místo, kde můžou být ukazatele. Režimy `(zp)` a `(zp),Y` neumějí vzít ukazatel odkudkoli — čtou ho vždycky z nulté stránky. Dva sousední bajty tam tvoří šestnáctibitový ukazatel (dolní bajt první, jako všude jinde v tomhle stroji) a instrukce `INW` a `DEW` s ním umějí krokovat.

Nultá stránka proto není jen „rychlá paměť“, ale **soubor ukazatelů**. Většina netriviálních programů si ji rozdělí na pojmenované sloty hned na začátku:

```
ptr     EQU 0x10        ; ukazatel do bufferu (2 bajty)
citac   EQU 0x12
zbytek  EQU 0x13
```

Zásobník

Zásobník leží na stránce 0x0100–0x01FF a **SP** se po resetu nastaví na 0x0200 — tedy **těsně za** její konec.

Je plně sestupný s předdekrementem: uložení nejdřív sníží **SP** a teprve pak zapíše, takže první uložený bajt skončí na 0x01FF a **SP** vždy ukazuje na nejnovější položku. Vyzvednutí je obráceně — přečte a pak **SP** zvýší.

> **Poznámka**
>
> Stránka 0x0100–0x01FF je konvence, ne hardware. **SP** je plnohodnotný šestnáctibitový registr a instrukce `TXYS` do něj umí zapsat cokoli. Nic nebrání přesunout zásobník jinam — jen se tím ztratí vlastnost, že přetečení zásobníku spadne do nulté stránky, kde je nápadné.

Little-endian, všude

Šestnáctibitové hodnoty jsou uložené dolním bajtem napřed. Platí to bez výjimky pro:

- operandy instrukcí (`JMP 0x1234` se v paměti uloží jako `C8 34 12`),
- vektory na konci paměti,
- ukazatele čtené režimy `(zp)` a `(zp),Y`,

- data zapsaná direktivou `.WORD`.

Vektory a boot

Poslední šest bajtů paměti drží tři adresy:

Adresa	Vektor	Kdy se čte
0xFFFA	BRK	při provedení instrukce `BRK`
0xFFFC	RESET	po zapnutí a po resetu
0xFFFE	IRQ	při přijetí hardwarového přerušení

Vektory nejsou nic zvláštního — jsou to obyčejné bajty v RAM, které mikrokód čte přes stejnou datovou sběrnici jako všechno ostatní. Proto se boot dá odkrokovat: reset vynuluje registry, nastaví **SP** a vloží do instrukčního registru pseudoinstrukci, jejíž celý mikrokód jsou dvě čtení z paměti. Podrobně je to v kapitole o přerušeních.

Assembler vektor RESET vyplní sám — nastaví ho na první vygenerovanou instrukci. Explicitně se vektory zapisují direktivou `.VECTOR`:

```
        .VECTOR IRQ, obsluha
        .VECTOR BRK, syscall
```

Kam program patří

Výchozí `.ORG` assembleru je 0x0200, tedy hned za zásobníkovou stránkou. Prostor odtud do 0x7FFF je zhruba 31,5 KB a je to místo pro kód i data. Velká pole se vejdou i za VRAM, do oblasti od 0xC000.

> **Past**
>
> Assembler odmítne umístit kód nebo data do bloku periferií 0xFF00–0xFF7F. Není to zbytečná opatrnost: bajty by se do paměti nikdy nedostaly, protože procesor tyhle adresy zachytí dřív — program by se tvářil, že je přeložený, a přitom by na tom místě nic nebylo.

Videopaměť

Oblast 0x8000–0xBFFF je 16 KB vyhrazených pro obraz. Je rozdělená na znakový buffer, paměť glyfů, barevnou RAM, bitmapový framebuffer a data spritů; podrobný popis každé části je v kapitolách o videu.

Pro paměťový model je podstatná jediná věc: **nic z toho nejsou registry**. Jsou to bajty v RAM. Rozdíl mezi „zapsat znak na obrazovku“ a „zapsat bajt do paměti“ na tomhle stroji neexistuje — a proto se dá obraz stavět blitterem, který o obrazovce nic neví a jen kopíruje paměť.

KAPITOLA 3

Kódování instrukcí

Opkód není položka v tabulce, kterou by si člověk musel pamatovat. Je to struktura. Osm bitů se dělí na tři pole a každé odpovídá na jednu otázku:

gg skupina — o jaký druh instrukce jde
ooo operace uvnitř skupiny
mmm adresní režim (u skupin, kde dává smysl)

Čtyři skupiny

`gg`	Rozsah	Skupina
`00`	0x00–0x3F	bezoperandové instrukce (implied), 1 bajt
`01`	0x40–0x7F	přesuny mezi registry a pamětí (load/store) + `CPX` / `CPY`
`10`	0x80–0xBF	aritmetika a logika, cílem je vždy **A**
`11`	0xC0–0xFF	řízení toku, čti-uprav-zapiš, `BIT`, operace nad slovem

Skupiny 01 a 10 jsou plně systematické: opkód je doslova `gg | ooo << 3 | mmm`. Proto v opkódové matici (příloha A) leží tytéž mnemoniky v souvislých osmicích — jeden řádek je jedna operace, jeden sloupec jeden adresní režim.

Skupiny 00 a 11 systematické být nemůžou, protože jejich instrukce operand buď nemají, nebo mají netypický. Uvnitř nich ale pravidelnost pokračuje po menších blocích: podmíněné skoky 0xC0–0xC7 mají tvar `11000 ff s`, kde `ff` vybírá příznak (Z, C, N, V) a `s` polaritu — skoč, když je nastaven.

Rozebraný příklad

Vezměme bajt 0x67:

Pole	Bity	Znamená
`gg`	`01`	skupina load/store
`ooo`	`100`	operace `STA` — ulož akumulátor
`mmm`	`111`	režim `(zp),Y`

Bajt 0x67 je tedy `STA (zp),Y`. Nebylo potřeba nic hledat — stačilo rozdělit osm bitů na tři pole. Přesně tohle je myšleno cílem „srozumitelnost“ z první kapitoly: dekodér instrukcí se dá popsat jednou větou, a proto se dá nakreslit na tabuli.

Adresní režimy a jejich kódování

Pole `mmm` má osm hodnot a všech osm se používá:

Režim	mmm	Zápis	Operand (B)	Efektivní adresa
`IMM`	`000`	`#n`	1	`–`
`ZP`	`001`	`zp`	1	`0x00:zp`
`ABS`	`010`	`abs`	2	`abs`
`ABS_X`	`011`	`abs,X`	2	`abs + X`
`ABS_Y`	`100`	`abs,Y`	2	`abs + Y`
`IND`	`101`	`(zp)`	1	`[zp, zp+1]`
`ZP_X`	`110`	`zp,X`	1	`0x00:((zp + X) & 0xFF)`
`IND_Y`	`111`	`(zp),Y`	1	`[zp, zp+1] + Y`

Obrázek 2. Adresní režimy a jejich kódování v poli `mmm`.

Sémantika každého režimu je v kapitole o adresních režimech; tady jde jen o to, že hodnota pole `mmm` je **tatáž** v celé skupině 01 i 10. Kdo si zapamatuje osm hodnot, umí přečíst polovinu opkódového prostoru.

Které kombinace jsou legální

Ne každá operace umí každý režim — to by nedávalo smysl (`STA #n` by ukládal do konstanty) a část kombinací se nevyplatí. Matice legálních kombinací je tohle:

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

Obrázek 3. Legální kombinace operace × adresní režim ve skupinách 01 a 10. Prázdné políčko = opkód existuje jako bajt, ale procesor ho odmítne.

Vzor je čitelný: **`A`** umí všechno, indexové registry mají užší nabídku (indexovat index nemá smysl), `STA` neumí immediate a `STX` / `STY` jen dvě základní adresní formy.

Délka instrukce

Délku určuje skupina a režim, ne tabulka:

Skupina	Délka
`00`	vždy 1 bajt
`01`, `10`	2 bajty pro `#n`, `zp`, `(zp)`, `zp,X`, `(zp),Y`; 3 bajty pro `abs` a `abs,X` / `abs,Y`
`11`	podle konkrétní instrukce: `RET` / `RTI` / `BRK` 1 bajt, formy s `zp` 2 bajty, formy s `abs` 3 bajty

Jinými slovy: instrukce je dlouhá 1 bajt plus tolik bajtů, kolik potřebuje její operand — a to je 0, 1 nebo 2.

Ilegální opkódy

Z 256 možných bajtů je 179 legálních instrukcí. Dva bajty jsou vyhrazené pro pseudo-instrukce, které vkládá hardware a nikdy je nelze načíst z paměti (0xFD pro reset, 0xFE pro přerušení). Zbývajících 75 slotů je volných.

Volný slot není `NOP`. Když se procesor pokusí provést bajt bez mikroprogramu, zastaví se a ohlásí ilegální opkód i s adresou, na které ho našel. Dva z těchto případů mají vlastní hlášení, protože jsou to nejčastější způsoby, jak se program utrhne:

Bajt	Diagnostika
0x00	„Provedena prázdná paměť“ — program skočil do nulami vyplněné oblasti
0xFF	„Provedena paměť vyplněná 0xFF“ — typicky přetečení za konec programu

Past

Rozdíl mezi „ilegální opkód“ a „volný slot“ je jen v čase. Volné sloty jsou místo, kam se dá ISA rozšířit — a několik jich už bylo obsazeno v ISA 3.1 (`JGT` , `JLE` , `JGES` , `JLTS` , `BIT` , `ADW` , `SBW` , `LXY` , `SXY` , `PHB` , `PLB`). Program, který spoléhá na to, že konkrétní bajt „nic nedělá“, přestane v příští verzi fungovat.

KAPITOLA 4

Datová cesta a řídicí signály

Procesor nedělá nic jiného, než že v každém taktu otevře a zavře několik hradel. Který registr pustí bajt na sběrnici, který si ho vezme, co s ním mezitím udělá ALU — to je celé. Řídicí signály jsou jména těchto hradel a mikrokód je seznam, které z nich jsou v kterém taktu sepnuté.

Tahle kapitola je slovníček k tomu seznamu.

Dvě sběrnice

V SAP-3/16 vedou dvě cesty a mají velmi rozdílnou šířku.

Datová sběrnice je osmibitová a teče po ní úplně všechno — přenosy mezi registry, čtení a zápisy do paměti, operandy do ALU i výsledky z ní.

Adresní cesta je šestnáctibitová, ale používají ji přesně tři přenosy:

Signál	Přenos
`PCM`	MAR := PC
`SPM`	MAR := SP
`JPW`	PC := OPH:OP (absolutní skok)

Všechny ostatní pohyby adres se dějí po osmi bitech přes datovou sběrnici. Tohle je nejdůležitější věta celé kapitoly, protože z ní plyne cena šestnáctibitové práce na osmibitovém stroji: každý bajt adresy, který se nedá přenést jednou z těch tří cest, stojí vlastní T-stav.

Odtud pochází celý rozdíl mezi `LDA zp` (6 T) a `LDA abs` (9 T) — tři takty navíc jsou načtení druhého bajtu adresy z paměti a jeho vložení do horní poloviny **MAR**.

Konvence pojmenování

Signály mají systematická jména a stojí za to si je přečíst jako zkratky:

- končí na `O` (**out**) — něco **dává** na datovou sběrnici,
- končí na `I` (**in**) — něco si z datové sběrnice **bere**,
- `MI`, `MIL`, `MIH` plní adresní registr **MAR** (celý zero-extended, dolní polovinu, horní polovinu),
- `V…` nastaví **MAR** na některý vektor.

V jednom T-stavu smí být aktivní nejvýš jeden zdroj sběrnice — jinak by dva registry tlačily na stejné vodiče. Cílů může být víc: bajt na sběrnici si může vzít současně třeba akumulátor i příznakové hradlo.

Skupiny signálů

Signálů je 70 a dělí se do šesti vzájemně vylučujících skupin. To dělení není jen dokumentační pomůcka — je to formát mikroslova, ze kterého se generuje ROM pro FPGA, a testy hlídají, že rozdělení je úplné.

Zdroje datové sběrnice

Signál	Popis
`RO`	čtení paměti — bajt z adresy v MAR na datovou sběrnici
`AO`	registr A na sběrnici
`BO`	registr B na sběrnici
`XO`	registr X na sběrnici
`YO`	registr Y na sběrnici
`TO`	registr TMP na sběrnici
`ORO`	operandový registr (dolní bajt) na sběrnici
`OHO`	operandový registr (horní bajt) na sběrnici
`CLO`	dolní polovina PC na sběrnici (uložení při CALL/IRQ)
`CHO`	horní polovina PC na sběrnici (uložení při CALL/IRQ)
`SLO`	dolní polovina SP na sběrnici (TSXY)
`SHO`	horní polovina SP na sběrnici (TSXY)
`FLO`	příznaky na sběrnici — C/Z/N/V/I zabalené do stavového bajtu
`EO`	výsledek ALU na sběrnici
`EXO`	sekundární výsledek ALU na sběrnici (horní bajt součinu, zbytek po dělení)

Cíle datové sběrnice

Signál	Popis
`MI`	MAR := 0x00:sběrnice — adresa v nulté stránce (horní polovina se vynuluje)
`MIL`	dolní polovina MAR := sběrnice
`MIH`	horní polovina MAR := sběrnice
`RI`	zápis do paměti — bajt ze sběrnice na adresu v MAR

Signál	Popis
`II`	instrukční registr := sběrnice
`AI`	registr A := sběrnice
`BI`	registr B := sběrnice
`XI`	registr X := sběrnice
`YI`	registr Y := sběrnice
`TI`	registr TMP := sběrnice
`ORI`	operandový registr (dolní bajt) := sběrnice
`OHI`	operandový registr (horní bajt) := sběrnice
`OI`	výstupní registr OUT := sběrnice
`JPL`	dolní polovina PC := sběrnice (návrat z RET/RTI)
`JPH`	horní polovina PC := sběrnice (návrat z RET/RTI)
`SLI`	dolní polovina SP := sběrnice (TXYS)
`SHI`	horní polovina SP := sběrnice (TXYS)
`FLI`	příznaky := sběrnice — obnovení C/Z/N/V/I ze stavového bajtu

Adresní cesta (MAR)

Signál	Popis
`PCM`	MAR := PC (šestnáctibitová adresní cesta)
`SPM`	MAR := SP (šestnáctibitová adresní cesta)
`MRI`	MAR := MAR + 1 — druhý bajt ukazatele nebo vektoru
`VRS`	MAR := 0xFFFC (báze vektoru RESET)
`VIR`	MAR := 0xFFFE (báze vektoru IRQ)
`VBR`	MAR := 0xFFFA (báze vektoru BRK)

Režimy ALU

Signál	Popis
`SU`	režim ALU: odečítání
`AND`	režim ALU: logický součin
`OR`	režim ALU: logický součet
`XOR`	režim ALU: nonekvivalence
`NOT`	režim ALU: negace bitů
`INC`	režim ALU: zvýšení o jedna
`DEC`	režim ALU: snížení o jedna

Signál	Popis
`SHL`	režim ALU: posun vlevo
`SHR`	režim ALU: posun vpravo
`ROL`	režim ALU: rotace vlevo přes příznak C
`ROR`	režim ALU: rotace vpravo přes příznak C
`CMP`	režim ALU: porovnání — nastaví jen příznaky, výsledek se zahodí
`MUL`	režim ALU: násobení — dolní bajt na výstup, horní přes EXO
`DIV`	režim ALU: dělení — podíl na výstup, zbytek přes EXO
`AHC`	ALU := TMP + addrCarry — přičtení přenosu k hornímu bajtu adresy
`AHB`	ALU := TMP − addrCarry — odečtení výpůjčky od horního bajtu adresy

Vedlejší efekty

Signál	Popis
`CFC`	vynuluj příznak C (CLC)
`CFS`	nastav příznak C (SEC)
`EIF`	povol přerušení — nastav příznak I (EI)
`DIF`	zakaž přerušení — vynuluj příznak I (DI)
`HLT`	zastav procesor, dokud nepřijde povolené přerušení
`RND`	náhodný bajt do A (xorshift32 se zadatelným semínkem)

Samostatné bity mikroslova

Signál	Popis
`JPW`	PC := OPH:OP (šestnáctibitová adresní cesta — absolutní skok)
`CE`	zvýšení programového čítače: PC := PC + 1
`SD`	snížení zásobníkového ukazatele (šestnáctibitově)
`SE`	zvýšení zásobníkového ukazatele (šestnáctibitově)
`CI`	přenos do ALU z příznaku C (dělá z ADD instrukci ADC, ze SUB instrukci SBC)
`ATM`	multiplexor ALU — první vstup z TMP místo z A
`AOP`	multiplexor ALU — druhý vstup z operandového registru místo z B
`ACO`	ulož přenos či výpůjčku z ALU do záklopky addrCarry (příznaky se nemění)
`FI`	zápis příznaků — s EO příznaky z ALU, jinak Z/N podle bajtu na sběrnici

ALU a její dva multiplexory

Aritmeticko-logická jednotka má dva vstupy. Ve výchozím zapojení bere první z **A** a druhý z **B** — proto je **B** „ten druhý operand“ a proto ho každá binární operace přepíše.

Dva signály tohle zapojení přepínají:

ATM první vstup vezmi z **TMP** místo z **A**,
AOP druhý vstup vezmi z operandového registru místo z **B**.

Bez těchhle dvou multiplexorů by výpočet efektivní adresy musel projít akumulátorem — a každé `LDA tabulka,X` by tiše zničilo obsah **A**. Díky nim se indexování počítá stranou, v registrech, které program nevidí.

Když je aktivní `EO` a žádný signál režimu, ALU sčítá. Sčítání je tedy výchozí operace, ne zvláštní případ.

Přenos mezi bajty adresy

Šestnáctibitový součet se na osmibitové ALU dělá po dvou krocích a mezi nimi je potřeba někam schovat přenos. K tomu slouží jednobitová záklopka **addrCarry** a trojice signálů:

Signál	Co udělá
`ACO`	uloží přenos (nebo výpůjčku) z právě proběhlé operace do **addrCarry** příznaky nechá být
`AHC`	ALU := TMP + addrCarry — přičtení přenosu k hornímu bajtu
`AHB`	ALU := TMP − addrCarry — totéž pro odečítání

Typický pár kroků z výpočtu `abs,X` vypadá takhle:

```
[ATM,AOP,EO,MI,ACO]     dolní bajt: TMP + operand -> MAR, přenos
do addrCarry
[ATM,AHC,EO,MIH]        horní bajt: TMP + addrCarry -> MAR horní
polovina
```

> **Poznámka**
>
> `ACO` je záměrně oddělený od `FI`. Kdyby výpočet adresy zapisoval příznak `C`, nešlo by napsat `LDA tabulka,X` mezi porovnáním a podmíněným skokem — a to je vzor, který se v reálném kódu vyskytuje pořád.

Kdy se zapisují příznaky

Zápis příznaků má na starosti jediný signál `FI` a jeho chování závisí na tom, jestli je ve stejném T-stavu aktivní `EO`:

s EO příznaky určí ALU podle provedené operace — a zapíšou se jen ty, které ta operace **definuje**.

bez EO ze sběrnice se odvodí `Z` a `N` podle přenášeného bajtu. Takhle nastavují příznaky přesuny mezi registry (`TAX`, `TYA`, …) a vyzvednutí ze zásobníku.

Množiny příznaků definované jednotlivými režimy ALU:

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

Obrázek 4. Které příznaky který režim ALU definuje. Co v řádku není, zůstane po operaci beze změny.

Past

`DIV` v tabulce definuje `C`, `Z` a `N` — ale to platí jen pro úspěšné dělení. Při dělení nulou operace definuje pouze `C`, takže `Z` a `N` zůstanou z předchozí instrukce. Je to jediné místo v ISA, kde se množina definovaných příznaků liší podle hodnot operandů.

Pořadí uvnitř jednoho T-stavu

Signály v jednom T-stavu nejsou současné — mají dokumentované pořadí, na kterém stojí správnost některých sekvencí:

> úprava SP ⟶ přenos po adresní cestě ⟶ zdroj datové sběrnice ⟶ cíle datové sběrnice ⟶ záklopka ALU (`ACO`) ⟶ inkrement PC (`CE`)

Nejnázornější důsledek je uložení na zásobník. Krok `[SD,SPM]` nejdřív sníží **SP** a **pak** ho pošle do **MAR** — proto je uložení předdekrementové a proto **SP** vždycky ukazuje na naposledy uložený bajt.

Druhý důsledek je jemnější a týká se vstupu do přerušení: v kroku `[FLO,RI,DIF]` se stavový bajt dostane na sběrnici **dřív**, než signál `DIF` zakáže přerušení. Do zásobníku se proto uloží stav s příznakem `I` ještě nastaveným — a instrukce `RTI` tím pádem přerušení při návratu sama povolí, aniž by to musela dělat explicitně.

KAPITOLA 5

Instrukční cyklus a T-stavy

T-stav je nejmenší jednotka času v tomhle stroji: jeden takt hodin, jedno mikroslovo, jedna sada sepnutých hradel. Instrukce je posloupnost T-stavů a nic víc — neexistuje žádná další vrstva, ve které by se něco dělo „mezi“.

Počty T-stavů uváděné v této knize jsou celkové, tedy **včetně** načtení instrukce. Když je u `LDA zp` napsáno 6 T, znamená to, že od okamžiku, kdy na tuhle instrukci přijde řada, do okamžiku, kdy je hodnota v akumulátoru, uplyne šest taktů.

Fetch

Načtení instrukce má tři varianty podle toho, kolik má instrukce operandových bajtů. Vypadají vždycky stejně a vždycky stejně dlouho trvají:

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

Každý pár kroků dělá totéž: pošli **PC** do **MAR**, přečti bajt z paměti, ulož ho do příslušného registru a zvyš **PC**. První bajt jde do instrukčního registru, druhý do **OP**, třetí do **OPH**.

Jednobajtové instrukce používají `F1`, tříbajtové `F3`. Dvoubajtové `F2` — a stejně tak instrukce s absolutní adresou používají `F3`, protože jejich adresa má dva bajty.

> **Poznámka**
>
> Fetch není režie, kterou by šlo schovat. U krátkých instrukcí je to většina jejich času: `LDA #n` trvá 5 T a čtyři z nich jsou fetch. Tomuhle jevu se v kapitole o výkonu říká „fetch tax“ a je to hlavní důvod, proč je na tomhle stroji hustota kódu důležitější než počet registrů.

Sestavení efektivní adresy

Mezi fetchem a vlastní prací leží u paměťových instrukcí kroky, které spočítají, na kterou adresu se vlastně sáhne. Jejich počet je celý rozdíl mezi adresními režimy:

Režim	T navíc	Co se děje
`#n`	0	operand **je** hodnota, nikam se nesahá
`zp`	1	operandový bajt rovnou do **MAR** (zero-extend)
`abs`	2	dolní a horní bajt adresy do **MAR**
`zp,X`	2	**X** do **TMP**, součet s operandem do **MAR** přenos se zahodí
`abs,X` / `abs,Y`	4	součet po bajtech s přenosem přes **addrCarry**
`(zp)`	5	přečti dvoubajtový ukazatel z nulté stránky
`(zp),Y`	8	přečti ukazatel a přičti k němu **Y**

Rozebraný příklad: `LDA 0x50`

Nejjednodušší paměťová instrukce, T-stav po T-stavu:

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

Obrázek 5. Mikroprogram instrukce `LDA zp` (opkód 0x41), 6 T-stavů.

T0 **PC** jde po adresní cestě do **MAR**.

T1 z paměti se přečte opkód, uloží do **IR**, **PC** se zvýší.

T2 **PC** znovu do **MAR** — tentokrát ukazuje na operand.

T3 operandový bajt do **OP**, **PC** se zvýší. Tady končí fetch.

T4 operand jde na sběrnici a `MI` z něj udělá adresu 0x00: `0x50`.

T5 přečte se bajt z té adresy, uloží do **A** a `FI` odvodí `Z` a `N`.

Za povšimnutí stojí, že v kroku T4 se `MI` chová jinak než `MIL`: adresu **nulté stránky** zapíše celou, včetně vynulování horní poloviny. Právě proto je nultá stránka levná — nepotřebuje druhý bajt.

Předání fetche mezi instrukcemi

Tohle je detail, který zaskočí každého, kdo čte mikrokód poprvé.

První dva kroky každé instrukce jsou fetch, ale **neprovádí je ona**. Instrukční registr se přepisuje až v kroku T1 — takže když procesor vykonává T0 a T1, běží ještě podle mikroprogramu instrukce **předchozí**.

Funguje to proto, že všechny mikroprogramy mají tyhle dva kroky totožné. A protože se to nesmí rozbít, platí pravidlo: žádné pole mikrokódu nesmí být kratší než tři kroky. Odtud jediná zvláštnost instrukce `NOP` — má za fetchem jeden prázdný krok navíc, aby pravidlo splnila:

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

Past

Praktický důsledek: `NOP` trvá 3 T, ne 2. Kdo počítá zpožďovací smyčky v T-stavech, musí s tím počítat — a kdo chce nejlevnější zdržení, sáhne spíš po `NOP` v `%rep`, protože kratší instrukce neexistuje.

Podmíněné skoky

Podmíněné skoky mají v posledním kroku signál `JPW` opatřený podmínkou. Když podmínka neplatí, celý krok se přeskočí — ale **neušetří čas**. Skok trvá 7 T bez ohledu na to, jestli se provedl.

Je to vědomé rozhodnutí ve prospěch determinismu: doba běhu programu nezávisí na datech, což je vlastnost, bez které by nefungovaly referenční stopy pro porovnání emulátoru s FPGA. Zároveň to zjednodušuje počítání času ve smyčkách — nemusíš vědět, kolikrát se skok provedl.

Delší instrukce

Nejdelší instrukce stroje trvají 14 T (formy ALU s režimem `(zp),Y`). Třináct T zabere `INW abs`, `DEW abs`, `CALL (zp)`, `ADW` a `SBW`.

Žádná instrukce se nedoplňuje na kulaté číslo. Instrukce trvá přesně tak dlouho, jak dlouho trvá její práce — a nástroje ukazují ta skutečná, různá čísla. Vypadá to méně úhledně než u strojů s pevnou délkou cyklu, ale je to poctivé a je to měřitelné.

KAPITOLA 6

Přerušení, RESET a BRK

Přerušení je jediné místo, kde procesor udělá něco, co v programu není napsané. Proto stojí za to vědět přesně, co udělá — a co naopak nechá na programátorovi.

Boot

Reset udělá tři věci: vynuluje registry, nastaví **SP** na 0x0200 a vynuluje příznak `I`. Pak vloží do instrukčního registru pseudoinstrukci 0xFD, kterou nelze načíst z paměti a která má tenhle mikroprogram:

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

Čtyři kroky, z toho dvě obyčejná čtení z paměti. `VRS` nastaví **MAR** na 0xFFFC, přečte se dolní bajt adresy do dolní poloviny **PC**, `MRI` posune **MAR** o jedna a přečte se horní bajt.

Tohle je celý boot. Není v tom žádná ROM, žádný skrytý stav — jsou to dvě čtení, která se dají odkrokovat a vidět na schématu.

> **Poznámka**
>
> Pseudoinstrukce nemají fetch. Nenačítají se z paměti, takže by se neměly z čeho načíst — jsou vložené hardwarem přímo do **IR**. Proto jejich mikroprogram začíná rovnou prací.

Assembler vektor RESET vyplní automaticky adresou první vygenerované instrukce, takže obyčejný program o bootu vůbec nemusí vědět.

Zdroje přerušení a maskování

Přerušení má tři hardwarové zdroje a dvě úrovně povolení.

Jednotlivé zdroje se povolují v registru `IRQ_ENABLE` (0xFF40):

Bit	Zdroj
0	časovač
1	klávesnice — čeká nepřečtený znak
2	sériová linka — v přijímacím bufferu něco je

Hlavní vypínač je příznak `I`, který se ovládá instrukcemi `EI` a `DI`. Dokud je `I` nula, procesor přerušení nepřijme, i kdyby byla všechna povolená.

Registr `IRQ_STATUS` (0xFF41) ukazuje, co je právě rozdělané; čtení ho maže.

Vstup do přerušení

Přerušení se přijímá jen na hranici instrukce — nikdy uprostřed. Když je na hranici `I` nastaveno a některý povolený zdroj čeká, procesor vloží pseudoinstrukci 0xFE:

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

Deset T-stavů, ve třech fázích:

1. Tři uložení na zásobník: horní bajt **PC**, dolní bajt **PC**, stavový bajt. V tomto pořadí — takže stavový bajt leží nahoře.
2. Signál `DIF` ve stejném kroku jako `FLO` zakáže přerušení.
3. `VIR` nastaví **MAR** na 0xFFFE a stejná dvojice čtení jako u resetu naplní **PC**.

Rámec na zásobníku vypadá takhle (adresy klesají směrem dolů):

```
vyšší adresa  PCH
              PCL
   SP →        F
```

Návrat obstará `RTI`, které rámec sundá v opačném pořadí: nejdřív stavový bajt, pak dolní a horní bajt **PC**.

> **Poznámka**
>
> `RTI` přerušení **nepovoluje** natvrdo — jen obnoví stavový bajt. Že se přerušení po návratu povolí, plyne z pořadí signálů uvnitř jednoho T-stavu: uložený stav má `I` ještě nastavené, protože `DIF` se uplatní až po tom, co stavový bajt opustil sběrnici.
>
> Rozdíl je vidět u vnořených přerušení: obsluha, která si `I` uprostřed vypne instrukcí `DI`, po `RTI` stejně dostane povolený stav — protože se obnovuje to, co platilo **před** přerušením.

Co obsluha musí udělat sama

Hardware uloží **PC** a příznaky. Nic víc. Všechny ostatní registry, které obsluha použije, si musí uložit sama.

Nejzrádnější z nich je **B**. Nestačí se ho „nedotknout“ — přepíše ho každá binární operace ALU, tedy i nevinně vypadající `CMP #0`. Obsluha, která kdekoli sáhne na ALU, musí **B** uložit, jinak tiše rozbije hodnotu, kterou si přerušený kód odložil instrukcí `TAB`.

```
obsluha:
        PUSH            ; ulož A
        PHB             ; ulož B
        PHX             ; ulož X, pokud ho použiješ
        ; ... tělo obsluhy ...
        PLX
        PLB             ; obnov B
        POP             ; obnov A
        RTI
```

Skryté registry **TMP** a **addrCarry** se ukládat nemusí — nepřenášejí stav přes hranici instrukce, a přerušení se přijímá právě jen tam.

Podprogramy volat lze: `CALL` a `RET` používají tentýž zásobník a obsluha přerušení není v ničem výjimečná.

`HLT` je čekání na přerušení

Instrukce `HLT` procesor zastaví, ale ne nutně navždy. Když přijde povolené přerušení, procesor se probudí a obsluha se provede. Skutečné zastavení nastane jen tehdy, když je `I` nula nebo není povolený žádný zdroj.

Odtud typická kostra programu řízeného událostmi:

```
        EI
smycka: HLT             ; spi, dokud se něco nestane
        JMP smycka
```

`BRK` — softwarové přerušení

`BRK` (0xCD) je normální, načítaná instrukce, která provede stejné uložení rámce jako hardwarové přerušení — ale spustí ji program, platí **bez ohledu na příznak** `I` a vektoruje se přes vlastní vektor 0xFFFA.

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

Dvanáct T-stavů: dva navíc oproti hardwarovému vstupu, protože `BRK` se na rozdíl od pseudoinstrukce musí načíst z paměti.

Vlastní vektor znamená, že obsluha `BRK` je jiná rutina než obsluha hardwarových přerušení. To z `BRK` dělá dvě užitečné věci: systémové volání (číslo služby se konvenčně předá v registru) a bod přerušení v ladicím nástroji — jeden bajt přepsaný na `BRK` zastaví program kdekoli.

Návrat se dělá instrukcí `RTI` a program pokračuje bajtem **za** `BRK`.

Past

`BRK` nelze zamaskovat. `DI` na něj nemá vliv — je to instrukce, ne událost. Program, který má vypnutá přerušení a spolehne se, že se do obsluhy nedostane, se splete, pokud někde na `BRK` narazí.

ČÁST II

Programovací model

KAPITOLA 7

Registry a příznaky

Programátor SAP-3/16 vidí šest registrů a pět příznaků. Je to málo — a je to záměr: čím méně stavů, tím snáz se dá stroj celý najednou udržet v hlavě. Cenou za to je, že se přes ten malý počet registrů musí protéct všechno, a proto má tahle kapitola víc pastí než kterákoli jiná.

Registry, které program vidí

Registr	Šířka	Role
A	8 b	akumulátor — cíl skoro každé aritmetiky, jediný registr, který umí všech osm adresních režimů
B	8 b	druhý operand ALU; **přepisuje ho každá binární operace**
X	8 b	index, hlavní pro `mem,X`
Y	8 b	index, nese `(zp),Y` a `mem,Y`
SP	16 b	zásobníkový ukazatel, po resetu 0x0200
PC	16 b	programový čítač
OUT	8 b	výstupní port instrukce `OUT`

Žádný z nich není zbytečný — což zní jako fráze, ale dá se to změřit. Ve všech programech v repozitáři nese **X** přes tisíc indexovaných přístupů, **Y** vedle nich zhruba třetinu, a dvojice `TAB` / `TBA` spolu s `PHB` / `PLB` tvoří skoro sedm procent všech skutečně vykonaných instrukcí. **OUT** je didaktické dědictví po SAP-1 a používá se přes sto dvacetkrát.

Registr B je jiný než ostatní

B není odkládací registr. Je to **vstup ALU**, a proto ho mikrokód naplní pokaždé, když se sáhne na aritmetiku:

```
        LDA delitel
        TAB             ; B := dělitel
        CMP #0          ; ← tady je B pryč, přepsala ho nula
        LDA delenec
        DIV             ; dělí nulou, ne dělitelem
```

Seznam instrukcí, které **B** přepíšou, je prostě „všechny binární operace ALU“: `ADD`, `ADC`, `SUB`, `SBC`, `AND`, `OR`, `XOR`, `CMP`, `BIT`, `CPX`, `CPY`, `ADW` a `SBW`.

> **Past**
>
> Nejzrádnější položka toho seznamu je `CMP`. Vypadá jako nedestruktivní („jen porovnám“), a vůči **A** to tak je — ale operand musí někam zapadnout, a to je **B**.
>
> Prakticky to znamená: mezi `TAB` a odpovídajícím `MUL` / `DIV` nesmí být **nic**, co sahá na ALU. Kdo potřebuje hodnotu v **B** přenést dál, uloží ji instrukcí `PHB`.

Skryté registry

Tyhle registry nemá žádná instrukce jak adresovat, ale bez nich se nedá číst mikrokód:

Registr	Šířka	K čemu
IR	8 b	právě prováděný opkód
OP	8 b	první operandový bajt
OPH	8 b	druhý operandový bajt
MAR	16 b	adresa, na kterou se právě sahá do paměti
TMP	8 b	pracovní vstup ALU pro výpočet adres a operace nad indexy
addrCarry	1 b	přenos mezi bajty adresy

Poslední dva stojí za pozornost, protože řeší konkrétní problém: kdyby se efektivní adresa počítala přes **A** a příznak `C`, zničila by každá instrukce typu `LDA tabulka,X` akumulátor i výsledek předchozího porovnání. Díky **TMP** a **addrCarry** se adresní aritmetika odehrává úplně mimo dosah programu.

> **Poznámka**
>
> Skryté registry nepřenášejí stav přes hranici instrukce — a přerušení se přijímá právě jen tam. Obsluha přerušení je proto ukládat nemusí.

Příznaky

Pět jednobitových příznaků:

Příznak	Bit	Význam
`C`	0	přenos při sčítání, **výpůjčka** při odečítání, vysunutý bit u posuvů

Příznak	Bit	Význam
`Z`	1	výsledek byl nula
`N`	2	nejvyšší bit výsledku (se znaménkem: záporné)
`V`	3	přetečení se znaménkem
`I`	4	přerušení povolena

Sloupec „bit“ je pozice ve stavovém bajtu, jak ho ukládá vstup do přerušení, instrukce `BRK` a `PHF`. Ostatní tři bity jsou nulové.

Zapíše se jen to, co operace definuje

Tohle je pravidlo, kolem kterého se točí půlka knihy: instrukce nastaví **pouze ty příznaky, které dávají smysl pro její operaci**. Ostatní si podrží předchozí hodnotu.

Není to nedbalost, ale funkce. Díky ní jde napsat tohle:

```
        CMP #10         ; nastaví C, Z, N
        AND #0x0F       ; nastaví Z, N — C přežije
        JC  mensi       ; a pořád platí výsledek porovnání
```

Kompletní rozdělení všech mnemonik podle toho, co zapisují:

Příznaky	Instrukce
žádné	`ADW`, `CALL`, `DEW`, `HLT`, `INW`, `JC`, `JGES`, `JGT`, `JLE`, `JLTS`, `JMP`, `JN`, `JNC`, `JNV`, `JNZ`, `JP`, `JV`, `JZ`, `LXY`, `NOP`, `OUT`, `PHB`, `PHF`, `PHX`, `PHY`, `PUSH`, `RET`, `RND`, `SBW`, `STA`, `STX`, `STY`, `SXY`, `TSXY`, `TXYS`
`C`	`CLC`, `SEC`
`I`	`BRK`, `DI`, `EI`
`Z N`	`AND`, `BIT`, `DEC`, `DEX`, `DEY`, `INC`, `INX`, `INY`, `LDA`, `LDX`, `LDY`, `NOT`, `OR`, `PLB`, `PLX`, `PLY`, `POP`, `TAB`, `TAX`, `TAY`, `TBA`, `TXA`, `TYA`, `XOR`
`C Z N`	`CMP`, `CPX`, `CPY`, `DIV`, `MUL`, `ROL`, `ROR`, `SHL`, `SHR`
`C Z N V`	`ADC`, `ADD`, `SBC`, `SUB`
`C Z N V I`	`PLF`, `RTI`

Obrázek 6. Které instrukce zapisují které příznaky. Co v řádku není uvedeno, zůstane po instrukci beze změny.

Za přečtení stojí hlavně první řádek — instrukce, které nemění nic. Jsou to skoro výhradně operace nad adresami a ukazateli (`INW`, `DEW`, `ADW`, `SBW`, `LXY`, `SXY`, `TXYS`) a řízení toku. Počítání s adresou nemá rozbít rozhodnutí, které program právě dělá.

Na druhém konci jsou `RTI` a `PLF`, které obnovují všech pět příznaků najednou, protože načítají celý stavový bajt.

Co určuje příznaky u operací ALU

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

Obrázek 7. Množina příznaků definovaná každým režimem ALU.

Dvě věci z téhle tabulky se hodí znát nazpaměť: logické operace nesahají na `C` (ISA 2.0 ho ještě nulovala) a `CMP` nesahá na `V`. Ta druhá má důsledek, který zaskočí každého — viz dále.

Konvence výpůjčky

Po odečítání znamená `C = 1` „odečtení podteklo“, tedy `A < operand`. Je to opačně než u 6502, kde je `C` nastavené při „bez výpůjčky“.

Aby si to nikdo nemusel pamatovat, má assembler aliasy pojmenované podle toho, jak člověk přemýšlí:

Zápis	Skutečná instrukce	Po `CMP` znamená
`JEQ`	`JZ`	rovná se
`JNE`	`JNZ`	nerovná se
`JLT`	`JC`	menší
`JGE`	`JNC`	větší nebo rovno

Praktický důsledek konvence: `SUB` a `SBC` do sebe zapadají bez přípravy. Nejnižší bajt se odečte instrukcí `SUB` (výpůjčku na vstupu ignoruje), vyšší instrukcí `SBC`. Proto se na tomhle stroji — na rozdíl od 6502 — nikdy nepíše `SEC` před odečítáním ani `CLC` před sčítáním.

Bez znaménka a se znaménkem

Bajt `0xFF` je 255, nebo −1. Procesor to nerozlišuje; rozlišuje to až skok, který si programátor vybere.

Porovnání	Skoky
bez znaménka	`JLT` / `JC`, `JGE` / `JNC`, `JGT`, `JLE`
se znaménkem	`JLTS`, `JGES`

Znaménkové porovnání se řídí podmínkou `N ⊕ V`: záporný výsledek **nebo** přetečení, které znaménko převrátilo.

Past

A tady je ta past: `CMP` nenastavuje `V`. Znaménkové skoky po něm proto čtou hodnotu `V` z nějaké dřívější instrukce a můžou odpovědět úplně špatně — tiše, bez chyby při překladu.

Znaménkové porovnání se páruje se `SUB` nebo `SBC`, které `V` nastaví:

```
        LDA #-100       ; 0x9C
        SUB #100        ; přeteče: N=0, V=1
        JLTS mensi      ; skočí — a je to správně
```

Cena je, že `SUB` na rozdíl od `CMP` přepíše akumulátor.

Pro úplnost — proč `JN` samo o sobě nestačí: u `-1 < 1` vyjde `0xFF - 0x01 = 0xFE`, tedy `N = 1` a `V = 0`, a `JN` odpoví správně. Ale u `-100 < 100` odečtení přeteče, `N` vyjde 0 a `JN` odpoví, že −100 není menší. Podmínka `N ⊕ V` tenhle případ pokrývá, `N` samo ne.

Kdy se příznaky zapisují

Na úrovni mikrokódu má zápis příznaků na starosti jediný signál, a jeho chování se liší podle toho, jestli je ve stejném T-stavu aktivní výstup ALU:

s výstupem ALU zapíšou se příznaky, které provedená operace definuje.
bez výstupu ALU `Z` a `N` se odvodí z bajtu, který je právě na datové sběrnici.

Druhý případ je důvod, proč nastavují příznaky i instrukce, které „nic nepočítají“ — `LDA`, `TAX`, `POP` a spol. Prakticky to šetří instrukce:

```
        LDA stav
        JZ  nic         ; není potřeba psát CMP #0
```

Měření to potvrzuje: `CMP #0` hned po nahrání se v celém korpusu programů vyskytuje třikrát. Zbytek se spoléhá na `Z` a `N` z loadu — správně.

KAPITOLA 8

Adresní režimy

Adresní režim odpovídá na otázku „kde je operand“. SAP-3/16 jich má osm a na rozdíl od většiny osmibitů je má **systematické**: režim je pole `mmm` v opkódu a jeho hodnota znamená totéž u každé instrukce, která ho umí.

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

Obrázek 8. Osm adresních režimů, jejich kódování a výpočet efektivní adresy.

Jednotlivé režimy

Přímý operand — `#n`

Hodnota je součástí instrukce. Nikam se nesahá, a je to proto nejrychlejší způsob, jak dostat konstantu do registru.

> **Past**
>
> Mřížka je povinná a její vynechání **není chyba překladu**. `LDA 5` znamená „nahraj obsah adresy 5“, `LDA #5` znamená „nahraj pětku“. Obojí je legální, takže překlep projde a projeví se až za běhu.
>
> Starší ISA měla pro přímý operand vlastní mnemoniky (`LDI` , `ANI` , `CPI` …), které tenhle omyl znemožňovaly. V ISA 3.0 byly zrušeny ve prospěch pravidelnosti — cenou je právě tahle past.

Nultá stránka — `zp`

Jednobajtová adresa, horní bajt je implicitně nula. Kratší o bajt a rychlejší o tři T-stavy než absolutní adresa.

Absolutní — `abs`

Plná šestnáctibitová adresa. Dosáhne kamkoli.

Indexovaná absolutní — `abs,X` a `abs,Y`

K šestnáctibitové bázi se přičte osmibitový index. Součet je **plný šestnáctibitový**, takže překročení hranice stránky funguje.

> **Poznámka**
>
> Překročení hranice 256 bajtů nestojí nic navíc. U 6502 stojí indexovaný přístup přes hranici stránky takt navíc; tady je součet vždycky dvoukrokový, takže je cena konstantní — což je součást slibu, že doba běhu nezávisí na datech.

Nepřímá — `(zp)`

V nulté stránce leží dvoubajtový ukazatel (dolní bajt první) a operand je tam, kam ukazuje.

Indexovaná v nulté stránce — `zp,X`

Adresa je `(zp + X) & 0xFF` — součet se **zabalí uvnitř nulté stránky**. Přenos se zahodí.

> **Past**
>
> Zabalení je záměrné (převzaté od 6502), ale překvapí. `LDA 0xF0,X` s `X = 0x20` nečte z adresy `0x0110`, ale z `0x0010`. Malé tabulky v nulté stránce se proto nesmějí umístit tak, aby přes konec stránky přetekly.

Nepřímá indexovaná — `(zp),Y`

Nejmocnější a nejdražší režim. Přečte se ukazatel z nulté stránky a teprve k němu se přičte **Y**. Je to **post-indexace**: index se přičítá až k načtené adrese, ne k adrese ukazatele.

Tohle je způsob, jak se prochází buffer, jehož adresa se počítá za běhu:

```
        LDY #0
.dalsi: LDA (zdroj),Y
        STA (cil),Y
        INY
        CPY #DELKA
        JNE .dalsi
```

> **Poznámka**

> Ukazatel se čte plnou šestnáctibitovou aritmetikou, takže ukazatel na adrese `0xFF` přečte svůj horní bajt z adresy 0x0100 — tedy ze zásobníkové stránky, ne ze začátku nulté stránky. Není to zabalení jako u `zp,X`.

Které režimy umí které instrukce

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

Obrázek 9. Legální kombinace ve skupinách 01 a 10.

Vzor má logiku:

- **A** umí všechno. Je to akumulátor a všechno kolem něj se točí.
- `STA` neumí přímý operand — ukládat do konstanty nedává smysl.
- `LDX` umí indexovat registrem **Y** a `LDY` registrem **X**. Indexovat registr sám sebou by nebylo k ničemu.
- `STX` a `STY` mají jen `zp` a `abs`.

Čtyři režimy, které neexistují

Pole `mmm` má osm hodnot a všech osm je obsazených. Proto v ISA nenajdeš `zp,Y` ani `(zp,X)` — nezbylo pro ně místo v kódování.

Past

Absence `zp,Y` má konkrétní důsledek: `STX tabulka,Y` nejde napsat. Indexované uložení indexového registru se musí obejít přes akumulátor:

```
        TXA
        STA tabulka,Y   ; A je tím pádem pryč
```

Cena režimů

Rozdíly v čase jsou dost velké, aby ovlivnily návrh programu. Na příkladu instrukce `LDA`:

Režim	Bajtů	T	Kde se čas ztrácí
`#n`	2	5	operand je hodnota, nikam se nesahá
`zp`	2	6	jeden bajt adresy
`abs`	3	9	dva bajty adresy
`abs,X`	3	11	součet po bajtech s přenosem
`abs,Y`	3	11	součet po bajtech s přenosem
`(zp)`	2	10	čtení dvoubajtového ukazatele
`zp,X`	2	7	součet indexu, přenos se zahodí
`(zp),Y`	2	13	čtení ukazatele plus šestnáctibitový součet

Tabulka 2. Co který adresní režim stojí, na příkladu instrukce `LDA`.

Heuristika, která z toho plyne: **co jde do nulté stránky, tam patří**. Rozdíl mezi `zp` a `abs` je tři T-stavy a jeden bajt na každý přístup, a v těsné smyčce se to sečte do desítek procent.

Automatická volba velikosti

Assembler si vybírá mezi `zp` a `abs` sám podle hodnoty operandu. Dělá to iterativně: začne s krátkými tvary a rozšiřuje je, dokud se velikosti neustálí. Velikosti se přitom smějí jen zvětšovat, takže proces vždycky skončí.

Totéž platí pro `sym,X` — přeloží se jako `zp,X`, pokud se báze vejde do bajtu, jinak jako `abs,X`. Naproti tomu `sym,Y` je **vždycky** `abs,Y`, protože `zp,Y` neexistuje.

Volbu jde vynutit příponou:

Zápis	Význam
`LDA.B sym`	vynuť nultou stránku; chyba, pokud `sym > 0xFF`
`LDA.W 0x42`	vynuť absolutní adresu i pro malou hodnotu
`LDA.W sym,X`	vynuť `abs,X` i pro bázi v nulté stránce

> **Poznámka**
>
> Vynucení se hodí, když se na adresu odkazuje samočinně generovaný kód nebo když se program spoléhá na **délku** instrukce — třeba u tabulky skoků, kde musí mít všechny položky stejnou velikost.

KAPITOLA 9

Zásobník a podprogramy

Zásobník v SAP-3/16 není zvláštní paměť. Je to obyčejná RAM, na kterou ukazuje registr **SP** — a všechno, co ho dělá zásobníkem, je způsob, jak s tím ukazatelem zachází pár instrukcí.

Jak funguje

SP se po resetu nastaví na 0x0200, tedy **těsně za** konec stránky 0x0100–0x01FF.

Uložení je předdekrementové: nejdřív se **SP** sníží, pak se na novou adresu zapíše. První uložený bajt tedy skončí na 0x01FF a **SP** vždycky ukazuje přímo na nejnovější položku. Vyzvednutí je opačné — přečte a pak zvýší.

```
0x01FF   první uložený bajt
0x01FE          druhý
0x01FD          třetí          ← SP
   ⋮            volné
0x0100                         konec stránky
```

Zásobník tedy roste směrem k nižším adresám — do nulté stránky.

Instrukce

Uložit	Vyzvednout	Co
`PUSH`	`POP`	akumulátor
`PHX`	`PLX`	registr X
`PHY`	`PLY`	registr Y
`PHB`	`PLB`	registr B
`PHF`	`PLF`	stavový bajt (příznaky)

Uložení nemění příznaky. Vyzvednutí nastaví `Z` a `N` podle vyzvednuté hodnoty — s výjimkou `PLF`, které obnoví všech pět příznaků najednou.

Každá z těch instrukcí trvá 4 T-stavy.

Volání podprogramů

`CALL` uloží návratovou adresu — nejdřív horní bajt, pak dolní — a skočí. `RET` je vyzvedne v opačném pořadí a vrátí se.

```
        CALL vypis
        HLT

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

Návratová adresa ukazuje na instrukci **za** voláním. Zabírá na zásobníku dva bajty, takže zásobníková stránka pojme 128 vnořených volání — pokud se na ní nic jiného neukládá.

Existuje i nepřímá forma `CALL (zp)`, která vezme cílovou adresu z ukazatele v nulté stránce. Je to způsob, jak udělat callback nebo tabulku funkcí:

```
        LDA #<obsluha
        STA vektor
        LDA #>obsluha
        STA vektor+1
        CALL (vektor)
```

Past

Přetečení zásobníku nikdo nehlídá. Když **SP** klesne pod 0x0100, začne zásobník přepisovat nultou stránku — tedy typicky ukazatele a proměnné programu. Projeví se to jako poškozená data někde úplně jinde, ne jako pád.

Právě proto je výchozí umístění zásobníku užitečné: přeteče do nulté stránky, kde se to obvykle projeví rychle a nápadně.

Předávání parametrů

Hardware žádnou konvenci nevynucuje. V praxi se používají tři vzory, každý pro jinou situaci.

V registrech

Nejrychlejší a nejběžnější. Do tří bajtů se vejde skoro všechno, co podprogram potřebuje.

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

V nulté stránce

Pro víc parametrů nebo pro šestnáctibitové hodnoty. Podprogram si je čte z pevných adres, které jsou součástí jeho „rozhraní“.

```
        LXY #text
        SXY ptr         ; ptr := adresa řetězce
        CALL vypis_text
```

Nevýhoda: takový podprogram není znovuvstupitelný. Když ho zavolá obsluha přerušení uprostřed jeho vlastního běhu, přepíše si parametry.

Na zásobníku

Nejobecnější a nejdražší. Parametry se uloží před voláním a podprogram si je přečte přes **SP**, který si zpřístupní instrukcí `TSXY`.

```
        LDA hodnota
        PUSH
        CALL zpracuj
        POP             ; úklid parametru
```

Poznámka

Úklid parametrů dělá volající. Podprogram to udělat nemůže, protože pod parametry leží jeho vlastní návratová adresa.

Práce se zásobníkovým ukazatelem

Dvojice `TXYS` a `TSXY` přenáší **SP** přes dvojici **X**:**Y**, kde **X** drží horní polovinu.

```
        LXY #0x0200
        TXYS            ; SP := 0x0200 (výchozí hodnota)
```

Typické použití: inicializace při startu a „úklid“ zásobníku při zotavení z chyby, kdy je jednodušší ukazatel nastavit natvrdo než odvíjet volání.

Poznámka

Přenos trvá dva T-stavy, protože šestnáctibitová hodnota jde po osmibitové sběrnici na dvakrát. Přerušení se ale přijímá jen na hranici instrukce, takže obsluha nikdy neuvidí **SP** rozpůlený.

Rekurze

Rekurze funguje, jen je potřeba počítat s hloubkou. Každá úroveň stojí dva bajty za návratovou adresu plus cokoli, co podprogram uloží.

```
; faktoriál(A) do A, rekurzivně
fakt:   CMP #2
        JLT .zaklad
        PUSH            ; ulož n
        DEC
        CALL fakt       ; A := (n−1)!
        TAB             ; B := (n−1)!
        POP             ; A := n
        MUL             ; A := n × (n−1)!
        RET
.zaklad: LDA #1
        RET
```

Při 128 dostupných úrovních a dvou bajtech navíc na úroveň vyjde bezpečná hloubka někde kolem čtyřiceti.

Zásobník a přerušení

Přerušení používá tentýž zásobník. Při vstupu se na něj uloží tři bajty (horní bajt **PC**, dolní bajt **PC**, stavový bajt), takže obsluha začíná o tři bajty hlouběji, než kde byl přerušený kód.

Z toho plynou dvě věci:

1. Obsluha smí volat podprogramy — zásobník je společný a nic zvláštního se neděje.
2. Do rozpočtu hloubky zásobníku je potřeba započítat i obsluhu. Program, který jede na hraně, přeteče až ve chvíli, kdy přijde přerušení.

Past

Obsluha musí zásobník opustit vyrovnaný. Co si uloží, musí vyzvednout — jinak `RTI` vezme jako návratovou adresu něco jiného. Je to tichá chyba, která se projeví skokem do neznáma při **příštím** přerušení.

KAPITOLA 10

Jazyk assembleru

Assembler SAP-3/16 je záměrně malý. Nemá sekce, moduly ani linker — přeloží zdrojový text rovnou na obraz paměti o velikosti 64 KB. Všechno, co umí, se vejde do jedné kapitoly.

Stavba řádku

```
[navesti:]  MNEMONIKA  [operand]   ; komentář
```

Všechny čtyři části jsou nepovinné; prázdný řádek i řádek s pouhým komentářem jsou v pořádku. Návěští končí dvojtečkou, komentář začíná středníkem a sahá do konce řádku.

Assembler nerozlišuje velikost písmen u mnemonik a registrů. U návěští a konstant na velikosti záleží.

Čísla

Zápis	Základ	Poznámka
`0x2A`	16	obvyklý tvar
`$2A`	16	tvar po vzoru 6502 — v tomhle tvaru vypisuje adresy disassembler
`0b101010`	2	hodí se u masek a bitových polí
`42`	10	
`-1`	10	záporná čísla se ukládají ve dvojkovém doplňku

Znakové literály se píšou v apostrofech a znají obvyklé únikové sekvence:

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

Podporované sekvence jsou `\n`, `\r`, `\t`, `\b`, `\f`, `\\`, `\"`, `\'` a `\0`. Cokoli jiného za zpětným lomítkem je chyba.

Výrazy

Výraz je buď číslo, znakový literál, konstanta, návěští, nebo návěští s posunem:

```
        LDA tabulka+2
        LDA konec-1
```

Víc než jeden operátor výraz nezvládne — není to plnohodnotná aritmetika, ale záměrně jednoduchý mechanismus pro adresování prvků.

Výběr bajtu

Šestnáctibitovou hodnotu rozdělí na bajty operátory `#<` (dolní) a `#>` (horní). Používají se všude, kde se plní ukazatel:

```
        LDA #<obsluha   ; dolní bajt adresy
        STA vektor
        LDA #>obsluha   ; horní bajt
        STA vektor+1
```

> **Poznámka**
>
> Od ISA 3.1 jde totéž napsat kratší dvojicí `LXY` / `SXY`, která přenese celou adresu najednou. Výběr bajtu zůstává užitečný tam, kde se s bajty pracuje odděleně.

Návěští

Návěští je jméno adresy. Globální návěští začíná písmenem nebo podtržítkem; lokální začíná tečkou a platí jen uvnitř bloku ohraničeného nejbližším předchozím globálním návěštím.

```
kresli: LDX #0
.smycka: STA (ptr),Y
        INX
        CPX #40
        JNE .smycka     ; odkaz na lokální návěští téhož bloku
        RET

maze:   LDX #0
.smycka: STA (ptr),Y    ; jiné .smycka, konflikt nevzniká
        INX
        CPX #40
```

```
        JNE .smycka
        RET
```

Lokální návěští jsou důvod, proč se v tomhle jazyce nemusí vymýšlet jména jako `smycka_kresli_2`.

Direktivy

.ORG
Nastaví adresu, na kterou se bude překládat. Výchozí hodnota je 0x0200.

```
        .ORG 0x0300
```

Operand musí být konstantní výraz — návěští nejsou dovolena, protože jejich adresa by na `.ORG` závisela.

EQU
Pojmenuje konstantu. Zapisuje se **bez dvojtečky**, což je jediné místo, kde se syntaxe liší od návěští:

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

Hodnota musí být konstantní výraz v šestnáctibitovém rozsahu. Konstanta a návěští se nesmějí jmenovat stejně.

DATA
Uloží bajty na aktuální adresu. Bere čísla i řetězce:

```
zprava: DATA "Ahoj", 0x0A, 0
tabulka: DATA 1, 2, 4, 8, 16, 32, 64, 128
```

.WORD *(**DW**)*
Uloží šestnáctibitová slova, dolním bajtem napřed:

```
skoky:  .WORD start, konec, chyba
```

.VECTOR
Zapíše adresu do jednoho ze tří vektorů na konci paměti:

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

Vektor RESET se vyplní automaticky adresou první vygenerované instrukce, takže obyčejný program ho psát nemusí.

Vynucená velikost

Přípony `.B` a `.W` vynutí kódování do nulté stránky, resp. absolutní:

```
        LDA.B ptr       ; vynuť zp (chyba, když se nevejde)
        LDA.W 0x42      ; vynuť abs i pro malou adresu
```

Bez přípony si assembler vybere sám a iteruje, dokud se velikosti neustálí.

Co překlad vyprodukuje

Kromě obrazu paměti vrací assembler i data, ze kterých žijí nástroje: výpis s adresami a bajty, tabulku symbolů, mapování adresa → řádek zdrojáku, seznam adres instrukcí, dat a operandů a vstupní bod programu.

Mapování adresa → řádek je to, díky čemu umí ladicí nástroj ukázat, na kterém řádku zdrojáku program právě je — a to i uvnitř rozvinutého makra.

Chyby překladu

Chybová hlášení nesou číslo řádku, a u kódu vzniklého z makra i kontext rozvinutí (`Line 12 (in macro DELAY expanded at line 30)`).

Struktura a symboly

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

Operandy a adresní režimy

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

Direktivy a rozsahy

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

Past

Jedno hlášení stojí za zvláštní zmínku, protože se týká paměťové mapy, ne syntaxe: assembler odmítne umístit kód nebo data do bloku periferií 0xFF00–0xFF7F. Bajty by se tam totiž nikdy nedostaly — procesor ty adresy zachytí dřív, než se z nich stane paměť.

KAPITOLA 11

Makroprocesor

Před assemblerem běží textový makroprocesor ve stylu NASM. Pracuje čistě nad řádky — o instrukcích ani adresách neví nic — a jeho výstupem je zdrojový text, který teprve dostane assembler.

Ta oddělenost je důvod, proč jsou makra tak volná: makro může obsahovat cokoli, i direktivy nebo kus jiného makra.

Makra

```
%macro DELAY 1          ; jméno a počet parametrů
        LDA %1          ; %1 až %9 jsou argumenty
%%loop:                 ; %%jméno = návěští na rozvinutí
        DEC
        JNZ %%loop
%endmacro

        DELAY #10       ; vyvolání
```

Parametry jsou poziční, maximálně devět. Makro se volá jménem a argumenty se oddělují čárkami.

Unikátní návěští

Návěští uvnitř makra by se při druhém rozvinutí srazilo samo se sebou. Prefix `%%` to řeší — z `%%loop` se stane jméno unikátní pro každé rozvinutí, takže se makro dá použít v jednom programu kolikrát chce.

> **Poznámka**
>
> Makra se smějí volat navzájem. Hloubka vnoření je omezená na 16 rozvinutí, což je pojistka proti nekonečné rekurzi, ne návrhový limit.

Textové náhrady

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

`%define` nahrazuje **celé identifikátory**, ne podřetězce — `VRAM_END` se tedy nerozbije definicí `VRAM`. Definice bez hodnoty (`%define DEBUG`) uloží jedničku.

> **Past**
>
> Pro pojmenování adres a konstant je skoro vždycky lepší direktiva `EQU` než `%define`. `EQU` zná assembler, takže se konstanta objeví v tabulce symbolů a v ladicím nástroji; `%define` je čistě textová náhrada, po které ve výstupu nezbude stopa.
>
> `%define` má smysl tam, kde `EQU` nestačí — tedy pro podmíněný překlad a pro náhradu, která není číslo.

Podmíněný překlad

```
%ifdef DEBUG
        CALL vypis_stav
%endif

%if VERZE >= 2
        CALL nova_funkce
%elif VERZE == 1
        CALL stara_funkce
%else
%error Nepodporovana verze
%endif
```

K dispozici jsou `%ifdef`, `%ifndef`, `%if`, `%elif`, `%else` a `%endif`.

Výraz v `%if` je jednoduchý: buď číslo, nebo dvě čísla spojená operátorem `==`, `!=`, `<`, `>`, `<=` nebo `>=`. Nenulová hodnota znamená pravdu. Složitější logiku makroprocesor nezná — a to je dobře, protože podmíněný překlad, který se nedá přečíst na první pohled, je zdrojem záhad.

Direktiva `%error` překlad zastaví s vlastní zprávou. Nezauvorkovaná zpráva projde náhradami `%define`, plně uzavorkovaná se vypíše doslova.

Opakování

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

Blok se rozvine daný početkrát, nejvýš tisíckrát. Hodí se na rozvinuté smyčky a generované tabulky.

Vkládání souborů

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

Vložení je textové. Vkládaný soubor smí definovat makra i konstanty.

Podmínky se musí uzavřít uvnitř téhož souboru, ve kterém začaly — ale `%define` hranice souborů překračuje. Právě z toho plyne obvyklá pojistka proti dvojímu vložení:

```
%ifndef MYLIB_INCLUDED
%define MYLIB_INCLUDED 1

; ... obsah knihovny ...

%endif
```

> **Poznámka**
>
> Jak se soubor najde, závisí na prostředí: nástroj příkazové řádky ho hledá na disku, prohlížeč v registru zabudovaných knihoven. Samotný makroprocesor souborový systém nezná — dostane funkci, která mu obsah dodá.

Limity

Limit	Hodnota	Proč existuje
hloubka rozvinutí maker	16	pojistka proti nekonečné rekurzi
průchody náhradami `%define`	8	náhrada může odkrýt další náhradu
počet opakování `%rep`	1024	
celkový výstup `%rep`	65 536 řádků	preprocesor běží při každém stisku klávesy v editoru

Poslední řádek prozrazuje, proč jsou limity zrovna takhle nízko: makroprocesor neběží jen při překladu, ale průběžně v editoru, aby fungovalo zvýrazňování a hlášení chyb. Musí tedy skončit rychle i nad rozepsaným, dočasně nesmyslným kódem.

Chyby a kontext

Každý řádek výstupu si nese původní číslo řádku a — pokud vznikl rozvinutím — i čitelný kontext. Chyba v makru proto neukazuje na vygenerovaný text, ale na oba relevantní řádky zdrojáku:

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

Díky témuž mechanismu funguje krokování a body přerušení v kódu, který vznikl z maker.

> **Past**
>
> Rozpoznávání komentářů je jednoduché a nezná uvozovky: středník kdekoli na řádku začíná komentář. Řetězec obsahující středník se tedy v makru chová jinak, než by člověk čekal.

KAPITOLA 12

Idiomy a vzory

Vzory v téhle kapitole nejsou otázka vkusu. Jsou to postupy, ke kterým tvar stroje programátora dotlačí: jeden akumulátor, osmibitová ALU, ukazatele jen v nulté stránce. Kdo je zná, píše kód, který vypadá jako ostatní kód pro SAP-3/16 — a to je na osmibitu praktická výhoda, ne estetika.

Čísla, která tu jsou uvedená, pocházejí z měření nad celým korpusem programů v repozitáři: 28 288 napsaných instrukčních řádků a 1 305 057 skutečně vykonaných instrukcí.

Akumulátorový shuffle

Nejčastější dvojice instrukcí ve skutečném běhu je `STA zp` následované `LDA zp` — připadá na ni **8,5 % všech po sobě jdoucích dvojic**. Není to neefektivita, je to daň za jediný akumulátor: mezivýsledek se musí odložit, aby se dal načíst další operand.

```
        LDA a
        ADD b
        STA soucet      ; odlož výsledek
        LDA c           ; a načti další operand
```

Ze stejného důvodu je `LDA #konstanta` následované `STA pamet` nejčastější statická dvojice v korpusu — 2 616 výskytů. Instrukce „ulož konstantu do paměti“ v ISA neexistuje, takže tenhle pár je jediný způsob.

Šestnáctibitová aritmetika

Stroj počítá po bajtech, ale adresy a souřadnice jsou šestnáctibitové. Vzory pro překlenutí toho rozdílu jsou čtyři a každý má své místo.

Součet a rozdíl přes ADD/ADC

Nejnižší bajt se sečte instrukcí `ADD`, která přenos na vstupu ignoruje a na výstupu nastaví; vyšší bajty instrukcí `ADC`, která ho použije.

```
        LDA #0xF4               ; 500 = 0x01F4
        ADD #0x2C               ; + 300 = 0x012C
        STA a_lo                ; C nese přenos výš
        LDA #0x01
```

```
        ADC #0x01
        STA a_hi                ; výsledek 0x0320 = 800
```

Odečítání funguje stejně s dvojicí `SUB` / `SBC`, kde `C` znamená výpůjčku.

> **Poznámka**
>
> Na 6502 by tady muselo být `CLC` před `ADD` a `SEC` před `SUB`. Tady ne — `ADD` a `SUB` vstupní příznak nečtou. Je to drobnost, ale ušetří dvě instrukce v každé vícebajtové operaci a je to důvod, proč se `CLC` a `SEC` ve skutečných programech prakticky nevyskytují.

Posun ukazatele přes ADW a INW

Když se má k **paměťovému** slovu přičíst konstanta, není potřeba tahat ho přes akumulátor. `ADW` a `SBW` to udělají přímo v paměti, `INW` a `DEW` o jedničku:

```
        ADW ptr, #40            ; o řádek níž (40 znaků)
        INW ptr                 ; o jeden bajt dál
```

Ani jedna z nich nemění příznaky, takže se dají zamíchat mezi porovnání a skok. `INW` je v korpusu použité 316×, `ADW` 78× — a to je instrukce stará pár dní, kterou si okamžitě vzal raycaster.

Nastavení ukazatele přes LXY a SXY

Naplnit dvoubajtový ukazatel adresou jde dvěma způsoby. Starší používá výběr bajtu:

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

Novější je jedna dvojice instrukcí a nesahá na akumulátor:

```
        LXY #buffer             ; X:Y := adresa (X je horní)
        SXY ptr                 ; ulož jako slovo do zp
```

Průchod bufferem

Kombinace ukazatele v nulté stránce a režimu `(zp),Y` je jediný způsob, jak procházet paměť, jejíž adresa se počítá až za běhu.

```
        LDY #0
.kopie: LDA (zdroj),Y
        STA (cil),Y
        INY
        CPY #4
        JNE .kopie
```

Dynamicky je dvojice `STA (zp),Y` → `INY` 1,4 % všech vykonaných dvojic — tenhle vzor je v běžícím kódu skutečně všude.

> **Poznámka**
>
> Pro delší úseky souvislé paměti existuje rychlejší cesta: blitter, který zkopíruje nebo vyplní celý blok jedním zápisem do řídicího registru. Smyčka výše je na místě tam, kde se s každým bajtem něco dělá.

Staging operandů pro MUL a DIV

`MUL` a `DIV` si berou druhý operand z registru **B**, do kterého se dostane jen přes akumulátor instrukcí `TAB`. Odtud vzor, který vypadá pozpátku:

```
        LDA delitel
        TAB                     ; B := dělitel
        LDA delenec             ; A := dělenec
        DIV                     ; A := podíl, B := zbytek
```

Trojice `TAB` → `LDA` → `MUL` tvoří asi 7 % vykonaných instrukcí — v matematickém kódu je to nejhustší místo celého programu.

> **Past**
>
> Mezi `TAB` a `MUL` / `DIV` nesmí být **nic**, co sahá na ALU. A stejná past číhá i za nimi, protože sekundární výsledek zůstává v **B**:
>
> ```
>         MUL
>         CMP #0x04       ; ← horní bajt součinu je pryč
>         TBA             ; přečte 0x04, ne horní bajt
> ```
>
> Řešení je vyzvednout druhou polovinu dřív, než se sáhne na ALU:

```
        MUL
        PHB             ; horní bajt na zásobník
        CMP #0x04       ; teď už může B klidně zmizet
        POP             ; a vytáhni ho zpět do A
```

Tenhle příklad není hypotetický — ukázkový program k této kapitole přesně na tuhle chybu při psaní narazil.

Posuny a pevná řádová čárka

Posun je nejlevnější násobení a dělení mocninou dvojky. Dynamicky jsou `SHR` a `SHL` nad akumulátorem 6,0 % a 4,9 % vykonaných instrukcí — víc než sčítání.

Vícebajtový posun se skládá z posuvu na jednom konci a rotace na druhém, přes příznak `C`:

```
        SHR a_hi                ; vysunutý bit jde do C
        ROR a_lo                ; a C ho vtáhne shora dovnitř
```

Doleva se skládá obráceně: `SHL` na dolním bajtu, `ROL` na horním.

Dvojice `SHR` → `SHR` je 3,9 % vykonaných dvojic a `SHL` → `SHL` 3,7 %. Za tím číslem je aritmetika s pevnou řádovou čárkou ve formátu Q8.8: po násobení dvou takových čísel je potřeba výsledek posunout zpátky, a to je řetěz posuvů.

Tabulka skoků

Stavový automat nebo dispatch podle indexu se dělá tabulkou adres a nepřímým skokem:

```
        LDA stav
        SHL                     ; index x 2 (adresa = 2 B)
        TAX
        LDA tabulka,X
        STA skok
        LDA tabulka+1,X
        STA skok+1
        JMP (skok)

tabulka: .WORD stav0, stav1, stav2
```

Pro volání místo skoku se použije `CALL (zp)` se stejnou přípravou.

Uzávěry smyček

Tři varianty podle toho, kde je čítač:

Čítač	Uzávěr
v paměti	`DEC citac` + `JNZ` — nesahá na registry, 148 výskytů v korpusu
v indexu, počítá dolů	`DEX` + `JNZ` — nejkratší
v indexu, počítá nahoru	`INX` + `CPX #N` + `JNE` — když index zároveň adresuje

Třetí varianta je dynamicky nejčastější (`INY` / `CPY` / `JNZ` je 1,6 % vykonaných trojic), protože při průchodu polem se index stejně potřebuje.

Držení snímkové frekvence

Program, který kreslí, musí běžet stejně rychle na pomalém i rychlém stroji. K tomu slouží počitadlo reálných milisekund a kotva, ke které se každý snímek vrací:

```
wait_frame:                     ; drž snímek na FRAME_MS ms
.spin:  LDA CLK_MS_LO           ; uplynulo = CLK_MS − kotva
        SUB ANCHOR
        STA EL
        LDA CLK_MS_HI
        SBC ANCHOR+1
        JNZ .done               ; přes 256 ms — dávno pozdě
        LDA EL
        CMP #FRAME_MS           ; C = 1, dokud je míň
        JC .spin
.done:  LDA CLK_MS_LO           ; kotva až při uvolnění
        STA ANCHOR
        LDA CLK_MS_HI
        STA ANCHOR+1
        RET
```

> **Poznámka**
>
> Kotva se posouvá **až při uvolnění**, ne na pevný násobek. Kdyby se počítal absolutní termín, snímek, který se nestihl, by si nesl dluh a program by se pokoušel ho dohnat — až by počitadlo přeteklo. Takhle je zpožděný snímek prostě zpožděný a další začíná od nuly.

Co se v korpusu skutečně vyskytuje

Vzor	Výskytů	Poznámka
`LDA #imm` ⟶ `STA mem`	2 616	uložení konstanty do paměti
`LDA mem` ⟶ `STA mem`	1 469	kopie z paměti do paměti
`LDA a` ⟶ operace ALU ⟶ `STA`	933	čtení, úprava, zápis přes akumulátor
šestnáctibitové porovnání	214	kaskáda `LDA` / `CMP` /skok dvakrát
`DEC mem` ⟶ `JNZ`	148	uzávěr smyčky s čítačem v paměti
`CMP #0` po nahrání	3	téměř nikdo — `Z` z loadu se používá správně

Poslední řádek je nejzajímavější: skoro nikdo nepíše zbytečné `CMP #0` po nahrání hodnoty, protože `LDA` příznaky nastaví samo. Je to drobnost, ale ukazuje, že konvence „příznaky nastavuje i přesun“ se v praxi ujala.

KAPITOLA 13

Výkon, cykly a měření

Výkon se na tomhle stroji nemusí odhadovat. T-stavy jsou součástí specifikace, procesor je počítá sám a program si je umí přečíst — takže odpověď na „je tenhle úsek rychlejší?“ je měření, ne názor.

Model

Cena instrukce je součet dvou položek:

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

Fetch stojí **2 T-stavy na každý bajt instrukce** — jeden na poslání programového čítače do adresního registru, jeden na čtení bajtu. Execute je zbytek: sestavení efektivní adresy plus vlastní práce.

Z toho plyne první a nejdůležitější číslo:

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

Tabulka 3. Podíl fetche na době trvání instrukce.

Víc než polovina strojového času jde na čtení instrukcí, ne na práci. Praktický důsledek: hustota kódu se vyplácí víc než chytré využití registrů. Kratší instrukce je rychlejší instrukce, skoro bez ohledu na to, co dělá.

Co stojí adresní režimy

Rozdíly mezi režimy jsou přímo v tabulce každého hesla slovníku. Na příkladu instrukce `LDA`:

Režim	Bajtů	T	Vůči `zp`
`#n`	2	5	−1 T
`zp`	2	6	—
`abs`	3	9	+3 T
`abs,X`	3	11	+5 T
`abs,Y`	3	11	+5 T
`(zp)`	2	10	+4 T
`zp,X`	2	7	+1 T
`(zp),Y`	2	13	+7 T

Tabulka 4. Cena adresních režimů, na příkladu instrukce `LDA`.

Nultá stránka je o tři T-stavy a jeden bajt levnější než absolutní adresa. V těsné smyčce, která se provede tisíckrát, je to rozdíl v desítkách procent celkového času — a je to nejlevnější optimalizace, kterou na tomhle stroji lze udělat.

Skoky stojí stejně, ať se provedou nebo ne

Podmíněný skok trvá 7 T-stavů bez ohledu na to, jestli podmínka platila. Neprovedený skok nešetří nic.

Vypadá to jako promarněná příležitost, ale je to záměr. Doba běhu programu tak nezávisí na datech, což je podmínka pro dvě věci, na kterých projekt stojí: referenční stopy, kterými se ověřuje, že se FPGA chová stejně jako emulátor, a návrat v čase v ladicím nástroji.

> **Poznámka**
>
> Stejné omezení mělo herní železo osmdesátých let — a právě proto se na něm „dalo počítat takty“. Program, jehož časování je spočítatelné dopředu, může synchronizovat obraz s procesorem, což je trik, který na moderním procesoru s cache a predikcí skoků nejde.

Počitadla

Procesor nabízí čtyři nezávislé časové základny. Všechna počitadla jsou jen pro čtení a nulují se pouze při resetu.

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

Obrázek 10. Registry časovače a počitadel (zařízení 3).

Počitadlo	Šířka	K čemu
`CYC0` – `CYC3`	32 b	T-stavy — přesné měření úseků kódu
`UPT_LO` / `UPT_HI`	16 b	snímky — čas ve snímcích obrazu
`CLK_TPMS_LO` / `HI`	16 b	kolik T-stavů se vejde do reálné milisekundy
`CLK_MS_LO` / `HI`	16 b	reálné milisekundy — základna pro držení tempa

> **Poznámka**
>
> Čtení nejnižšího bajtu **zamkne snímek** celé hodnoty. Program tedy musí číst `CYC0` před `CYC1`, jinak dostane dva bajty z různých okamžiků a u přechodu přes 256 mu vyjde nesmysl. Totéž platí pro `UPT_LO` a `CLK_MS_LO`.

Registr `CLK_TPMS` je jediné okno do toho, jak rychle stroj běží. Program, který se chce chovat stejně na pomalém i rychlém stroji, si z něj spočítá, kolik práce se do snímku vejde — nebo rovnou použije `CLK_MS`.

Jak se měří

Rozdíl dvou odečtů počitadla T-stavů je přesný počet taktů mezi nimi — včetně režie samotného měření. Tu je potřeba změřit zvlášť a odečíst:

```
        ; referenční měření: dva odečty hned za sebou
        CALL vzorek
        CALL rozdil
        LDA d_lo
        STA rezie

        ; měřený úsek
        CALL vzorek
        LDX #10
.smycka: DEX
        JNZ .smycka
        CALL rozdil

        LDA d_lo
        SUB rezie               ; odečti režii
        STA d_lo
```

Kompletní program je v `book/reference/examples/13-mereni.asm`. Vypíše `0073`, tedy 115 T-stavů — a to je přesně to, co vyjde ručním součtem:

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

Shoda teorie a měření na jednotku není náhoda; je to ta samá vlastnost, díky které se dá stroj krokovat a vracet v čase.

> **Past**
>
> Ten program sám ukázal, proč se má měření ověřovat, a ne odhadovat: v první verzi bydlela konstanta `rezie` na adrese `0x15`, což je druhý bajt dvoubajtové proměnné `tmp` na `0x14`. Podprogram `rozdil` si tím režii přepisoval a měření vycházelo o 46 T-stavů vyšší. Nultá stránka je malá a kolize v ní nikdo nehlásí.

Rozpočet na snímek

Snímková základna je 1 000 T-stavů (`VSYNC_PERIOD`), ale pro plánování je užitečnější počítat v reálném čase: kolik práce se stihne mezi dvěma snímky na cílové frekvenci.

Průměrná instrukce stojí 7 až 9 T-stavů. Na desce s jádrem AGATA-1, které běží na 25,2 MHz s jedním T-stavem na takt, to vychází na zhruba **3 miliony instrukcí za sekundu**. Pro srovnání: 6502 na 1 MHz zvládne asi 0,35 MIPS — deska je tedy zhruba desetinásobek klasického osmibitu.

Při 30 snímcích za sekundu vychází rozpočet kolem 100 000 instrukcí na snímek. To je hodně — a zároveň to rychle dojde, jakmile se začne kreslit po pixelech.

Rychlost	Kde se používá
zlomky Hz	krokování s viditelnými řídicími signály — výukový režim
do 2 MHz	emulátor v prohlížeči
25,2 MHz	turbo režim na FPGA; horní mez, kterou připouští specifikace

> **Poznámka**
>
> Specifikace frekvenci nefixuje. Registr `CLK_TPMS` může číst až 25 200, což odpovídá jednomu T-stavu na takt pixelových hodin FPGA. Program by na konkrétní frekvenci neměl spoléhat — má si ji přečíst.

Co se vyplatí optimalizovat

Měření na skutečném korpusu programů dává jasné pořadí.

1. **Přesuň data do nulté stránky.** Tři T-stavy a jeden bajt na každý přístup, bez jakékoli změny logiky.
2. **Zkraťkód.** Fetch je nadpoloviční položka; každý ušetřený bajt instrukce je ušetřený T-stav při každém průchodu.
3. **Nech práci blitteru.** Kopie bloku paměti přes řídicí registr proběhne bez instrukční smyčky.
4. **Vyhni se zbytečnému shufflu.** Dvojice `STA zp` / `LDA zp` je 8,5 % vykonaných dvojic; část z toho jde odstranit přeuspořádáním výpočtu tak, aby mezivýsledek zůstal v akumulátoru.

A co se nevyplatí: mikro-optimalizace uvnitř výrazů. Rozdíl mezi dvěma způsoby zápisu téhož výpočtu bývá jednotky T-stavů, zatímco přesun proměnné do nulté stránky nebo zkrácení smyčky o jednu instrukci se násobí počtem průchodů.

> **Past**
>
> Počty T-stavů uvedené v této knize platí pro ISA 3.1. Kdyby budoucí verze jádra zkrátila fetch — což je nejnadějnější směr, právě protože je fetch dominantní — změní se tabulky v celé knize. Proto se generují ze zdroje a ne opisují.

ČÁST III

Instrukční slovník

KAPITOLA 14

Jak číst slovník

Slovník v následující kapitole popisuje každou mnemoniku zvlášť, abecedně. Heslo je stavěné tak, aby se dalo přečíst shora dolů a přestat kdykoli: nahoře je to, co člověk hledá nejčastěji, dole to, co potřebuje jednou za rok.

Anatomie hesla

Každé heslo začíná mnemonikou, jednou větou o tom, co dělá, a tabulkou všech jejích zakódovaných tvarů. Sloupce tabulky znamenají:

Sloupec	Význam
Režim	adresní režim; `implied` znamená, že instrukce nemá operand
Zápis	jak se tvar píše v assembleru
Opkód	bajt, který assembler vygeneruje
Bajtů	celková délka instrukce v paměti
T	počet T-stavů včetně fetche — přesně to, co instrukce stojí
Příznaky	příznaky, které tvar **definuje**

Za tabulkou následuje popis sémantiky, u netriviálních instrukcí mikrokód, ukázka a pasti.

Co znamená prázdný sloupec příznaků

Tohle je nejdůležitější konvence celé knihy. Sloupec „Příznaky“ vypisuje pouze ty příznaky, které instrukce nastaví na novou hodnotu. Příznak, který tam není uvedený, si **ponechá předchozí hodnotu** — nevynuluje se.

Rozdíl není kosmetický. Právě proto jde napsat tohle:

```
CMP #10         ; nastaví C, Z, N
AND #0x0F       ; nastaví Z, N — C zůstává
JC  mensi       ; skok se pořád řídí výsledkem CMP
```

U instrukcí, které nedefinují žádný příznak, je v tabulce místo výčtu slovo „žádné“. Typicky jde o operace nad adresami a ukazateli (`INW`, `DEW`, `ADW`, `SBW`, `LXY`, `SXY`, `TXYS`): počítání s adresou nemá důvod rozbít příznaky, podle kterých se program právě rozhoduje.

> **Poznámka**
>
> Instrukce `RTI` a `PLF` jsou opačný extrém — obnovují **všech pět** příznaků najednou včetně `I`, protože načítají celý stavový bajt ze zásobníku. V tabulce mají proto uvedeno `C Z N V I`.

Mikrokódové výpisy

U instrukcí, kde je zajímavé, **jak** se výsledku dosáhne, je uvedený výpis mikrokódu. Jeden pár hranatých závorek je jeden T-stav; uvnitř jsou řídicí signály, které jsou v tom T-stavu aktivní. Číslo na konci je celkový počet T-stavů.

První dva až šest kroků je vždy fetch a vypadají u všech instrukcí stejně (`[PCM] [RO,II,CE]` a případné načtení operandových bajtů). Zajímavá část začíná až za nimi.

Význam jednotlivých signálů je v příloze o řídicích signálech; pro čtení výpisu stačí vědět, že signál končící na `O` typicky něco na sběrnici **dává** a signál končící na `I` si to ze sběrnice **bere**.

Ukázky

Ukázky kódu jsou úryvky ze skutečných programů, ne pseudokód. Komentáře jsou české, mnemoniky pochopitelně ne. Kde ukázka něco vypisuje, je to výstup, který stroj opravdu vyprodukuje.

Pasti

Rámeček označený jako past neupozorňuje na chybu, ale na chování, které je správné a přitom překvapivé. Většina z nich má společný původ: procesor dělá přesně to, co je napsané v mikrokódu, a lidská intuice někdy čeká něco jiného.

> **Past**
>
> Ukázka takového rámečku. Nejčastější past celého stroje: každá binární operace ALU přepíše registr **B** svým operandem. Hodnota, kterou jsi tam dal instrukcí `TAB`, nepřežije nejbližší `ADD`, `CMP` ani `AND`.

Viz také: `LDA`, `CMP`, `DIV`

KAPITOLA 15

Slovník instrukcí

Hesla jsou seřazená abecedně podle mnemoniky. Tvary, které se liší jen adresním režimem, jsou v jednom hesle; aliasy skoků (`JEQ`, `JNE`, `JGE`, `JLT`) mají vlastní krátká hesla s odkazem na instrukci, kterou zastupují.

Slovník pokrývá všech 79 mnemonik, které procesor umí dekódovat. Anatomie hesla a konvence zápisu jsou v předchozí kapitole.

Přehled podmíněných skoků

Skoků je dvanáct a snadno se pletou, takže tady je celá rodina najednou. Všechny mají jen absolutní tvar, jsou tříbajtové a trvají 7 T bez ohledu na to, jestli se skok provedl.

Instrukce	Podmínka	Po `CMP` znamená
`JZ` / `JEQ`	`Z`	rovná se
`JNZ` / `JNE`	`¬Z`	nerovná se
`JC` / `JLT`	`C`	menší (bez znaménka)
`JNC` / `JGE`	`¬C`	větší nebo rovno (bez znaménka)
`JGT`	`¬C ∧ ¬Z`	ostře větší (bez znaménka)
`JLE`	`C ∨ Z`	menší nebo rovno (bez znaménka)
`JN`	`N`	záporné — spolehlivé jen při `V = 0`
`JP`	`¬N`	kladné nebo nula — **není** nepodmíněný skok
`JLTS`	`N ⊕ V`	menší se znaménkem — páruj se `SUB`, ne s `CMP`
`JGES`	`¬(N ⊕ V)`	větší nebo rovno se znaménkem — dtto
`JV`	`V`	přetečení se znaménkem
`JNV`	`¬V`	bez přetečení

Tři řádky téhle tabulky jsou nejčastější zdroje chyb: `JP` není `JMP`, znaménkové skoky nefungují po `CMP` (které nenastavuje `V`), a `C` je po odečítání výpůjčka, ne přenos.

ADC

Přičte operand k akumulátoru včetně příznaku C. Vyšší bajty vícebajtového součtu.

Režim	Zápis	Opkód	Bajtů	T	Příznaky
`#n`	`ADC #n`	`0x88`	2	6	`C Z N V`
`zp`	`ADC addr8`	`0x89`	2	7	`C Z N V`
`abs`	`ADC addr16`	`0x8A`	3	10	`C Z N V`
`abs,X`	`ADC addr16,X`	`0x8B`	3	12	`C Z N V`
`abs,Y`	`ADC addr16,Y`	`0x8C`	3	12	`C Z N V`
`(zp)`	`ADC (addr8)`	`0x8D`	2	11	`C Z N V`
`zp,X`	`ADC addr8,X`	`0x8E`	2	8	`C Z N V`
`(zp),Y`	`ADC (addr8),Y`	`0x8F`	2	14	`C Z N V`

`ADC` je `ADD` plus přenos: `A := A + operand + C`. Existuje kvůli sčítání čísel širších než jeden bajt.

Rozdělení práce mezi `ADD` a `ADC` je jednoduché a nepotřebuje žádnou přípravu příznaků: nejnižší bajt sečti instrukcí `ADD` (ta přenos na vstupu ignoruje a nastaví ho na výstupu), všechny vyšší instrukcí `ADC`.

```
        LDA a_lo
        ADD b_lo        ; C := přenos z dolního bajtu
        STA soucet_lo
        LDA a_hi
        ADC b_hi        ; přičte i ten přenos
        STA soucet_hi
```

Příznak `V` hlásí přetečení se znaménkem — tedy že výsledek nesedí, pokud operandy čteme jako čísla se znaménkem.

Viz také: `ADD`, `SBC`, `ADW`, `CLC`

ADD

Přičte operand k akumulátoru. Přenos na vstupu ignoruje, na výstupu nastaví.

Režim	Zápis	Opkód	Bajtů	T	Příznaky
`#n`	`ADD #n`	`0x80`	2	6	`C Z N V`
`zp`	`ADD addr8`	`0x81`	2	7	`C Z N V`
`abs`	`ADD addr16`	`0x82`	3	10	`C Z N V`
`abs,X`	`ADD addr16,X`	`0x83`	3	12	`C Z N V`
`abs,Y`	`ADD addr16,Y`	`0x84`	3	12	`C Z N V`
`(zp)`	`ADD (addr8)`	`0x85`	2	11	`C Z N V`
`zp,X`	`ADD addr8,X`	`0x86`	2	8	`C Z N V`
`(zp),Y`	`ADD (addr8),Y`	`0x87`	2	14	`C Z N V`

`A := A + operand`. Příznak `C` na vstupu nehraje roli — na rozdíl od 6502 tedy **není potřeba před sčítáním volat** `CLC`.

```
        LDA cena
        ADD dan
        STA celkem
        JC  preteklo    ; C = 1 → nevešlo se do bajtu
```

> **Past**
>
> Operand skončí v registru **B**. Platí to o každé binární operaci ALU a je to nejčastější příčina toho, že hodnota uložená instrukcí `TAB` „záhadně“ zmizí.

Viz také: `ADC`, `SUB`, `ADW`, `MUL`

ADW

Přičte osmibitovou konstantu k šestnáctibitovému slovu v nulté stránce. Nemění příznaky.

Režim	Zápis	Opkód	Bajtů	T	Příznaky
`zp`	`ADW addr8,#n`	`0xE8`	3	13	*žádné*

`ADW zp, #imm8` sečte celé dvoubajtové slovo uložené na adrese `zp` (dolní bajt první) s konstantou a výsledek zapíše zpátky. Provede se to celé v paměti — akumulátor ani indexy se nepoužijí.

Je to instrukce pro krokování ukazatelů o víc než jedna. Typicky posun na další řádek obrazovky:

```
        ADW ptr, #40    ; ukazatel o řádek níž (40 znaků)
```

Žádný příznak se nemění. Je to záměr: posouvání ukazatele nemá rozbít výsledek porovnání, podle kterého se smyčka rozhoduje.

> **Poznámka**
>
> Konstanta je osmibitová a **nezáporná**. Odečítání se dělá instrukcí `SBW`, ne přičtením záporného čísla — to by přeneslo výpůjčku špatně.

Viz také: `SBW`, `INW`, `DEW`, `LXY`, `SXY`

AND

Logický součin akumulátoru s operandem po bitech.

Režim	Zápis	Opkód	Bajtů	T	Příznaky
`#n`	`AND #n`	`0xA0`	2	6	`Z N`
`zp`	`AND addr8`	`0xA1`	2	7	`Z N`
`abs`	`AND addr16`	`0xA2`	3	10	`Z N`
`abs,X`	`AND addr16,X`	`0xA3`	3	12	`Z N`
`abs,Y`	`AND addr16,Y`	`0xA4`	3	12	`Z N`
`(zp)`	`AND (addr8)`	`0xA5`	2	11	`Z N`
`zp,X`	`AND addr8,X`	`0xA6`	2	8	`Z N`
`(zp),Y`	`AND (addr8),Y`	`0xA7`	2	14	`Z N`

`A := A & operand`. Nastaví `Z` a `N`; `C` a `V` zůstanou beze změny.

Používá se k maskování — nechat z bajtu jen část bitů:

```
        LDA stav
        AND #0x0F       ; ponech jen dolní půlbajt
```

> **Poznámka**

> To, že `AND` nemění `C`, je záměrné rozhodnutí ISA 3.0 (starší verze příznak `C` nulovaly). Díky tomu jde mezi porovnáním a podmíněným skokem maskovat bity, aniž by se výsledek porovnání ztratil.

Viz také: `OR`, `XOR`, `NOT`, `BIT`

BIT

Otestuje bity operandu proti akumulátoru. Nastaví jen příznaky, akumulátor nechá být.

Režim	Zápis	Opkód	Bajtů	T	Příznaky
`#n`	`BIT #n`	`0xE4`	2	6	`Z N`
`zp`	`BIT addr8`	`0xE5`	2	7	`Z N`
`abs`	`BIT addr16`	`0xE6`	3	10	`Z N`

`BIT` provede totéž co `AND`, ale výsledek zahodí — zůstanou jen příznaky `Z` a `N`. Je to obdoba, jakou je `CMP` pro `SUB`.

```
        LDA maska
        BIT stav        ; je některý bit masky nastavený?
        JZ  zadny
```

Nejčastější použití je test jednoho bitu bez ztráty akumulátoru:

```
        LDA #0b00000100
        BIT priznaky
        JNZ bit2_je_nastaven
```

> **Past**
>
> `BIT` sice nechá **A** být, ale **B** přepíše jako každá binární operace ALU. „Nedestruktivní“ znamená jen vůči akumulátoru.

Viz také: `AND`, `CMP`

BRK

Softwarové přerušení. Uloží stejný rámec jako hardwarové IRQ, ale má vlastní vektor.

Režim	Zápis	Opkód	Bajtů	T	Příznaky
implied	`BRK`	`0xCD`	1	12	`I`

`BRK` je normální načítaná instrukce, která uloží na zásobník horní bajt **PC**, dolní bajt **PC** a stavový bajt, zakáže přerušení a skočí na adresu z vektoru BRK (`0xFFFA`).

Od hardwarového přerušení se liší třemi věcmi: spouští ji program, **nelze ji zamaskovat** příznakem `I`, a vektoruje se jinam — obsluha `BRK` je tedy jiná rutina než obsluha IRQ.

```
        .VECTOR BRK, syscall

        LDA #SLUZBA_VYPIS
        BRK             ; zavolej systémovou službu
        ; sem se program vrátí po RTI
```

Návrat se dělá instrukcí `RTI` a pokračuje se bajtem **za** `BRK`. Kromě systémových volání se `BRK` používá jako bod přerušení v ladicím nástroji — jeden bajt přepsaný na `0xCD` zastaví program kdekoli.

Viz také: `RTI`, `EI`, `DI`

CALL

Zavolá podprogram: uloží návratovou adresu na zásobník a skočí.

Režim	Zápis	Opkód	Bajtů	T	Příznaky
`abs`	`CALL addr16`	`0xCA`	3	11	*žádné*
`(zp)`	`CALL (addr8)`	`0xCE`	2	13	*žádné*

`CALL` uloží na zásobník adresu následující instrukce (nejdřív horní bajt, pak dolní) a skočí na cíl. Návrat obstará `RET`.

Existují dvě formy. Přímá (`CALL navesti`) je běžná. Nepřímá (`CALL (zp)`) vezme cílovou adresu z ukazatele v nulté stránce — což je způsob, jak na tomhle stroji udělat tabulku funkcí nebo callback:

```
        LDA #<obsluha   ; dolní bajt adresy
        STA vektor
```

```
        LDA #>obsluha   ; horní bajt
        STA vektor+1
        CALL (vektor)
```

Zásobník je společný s přerušeními, takže volání uvnitř obsluhy přerušení je v pořádku.

Past

Vnoření volání není nijak omezené shora — zásobník má 256 bajtů, tedy 128 návratových adres, a nikdo nehlídá, že přeteče. Rekurze do hloubky přepíše nultou stránku a projeví se to jako poškozená data, ne jako pád.

Viz také: `RET`, `JMP`, `BRK`

CLC

Vynuluje příznak přenosu.

Režim	Zápis	Opkód	Bajtů	T	Příznaky
implied	`CLC`	`0x04`	1	3	`C`

`C := 0`. Neovlivní nic jiného.

Poznámka

Na rozdíl od 6502 se `CLC` **nepotřebuje před sčítáním**. `ADD` přenos na vstupu ignoruje a `SUB` také; přenos čte jen `ADC` a `SBC`, a ty se používají výhradně na vyšší bajty, kde má být přenos právě ten z předchozího kroku.

Měření to potvrzuje důkladně: ve všech ukázkových programech a hrách v repozitáři se `CLC` ani `SEC` jako instrukce nevyskytují ani jednou — jediné použití je v testu, který schválně provádí každý opkód. Jeden program dokonce používá `CLC` jako jméno konstanty, což může fungovat jen proto, že se ta instrukce nikde nepíše.

Neznamená to, že jsou zbytečné: nastavit vstupní bit pro `ROL` nebo `ROR` jinak nejde. Znamená to, že rozdělení práce mezi `ADD` a `ADC` vyšlo tak, že se o přenos není potřeba starat ručně.

Viz také: `SEC`, `ADC`, `SBC`, `ROL`, `ROR`

CMP

Porovná A s operandem: nastaví příznaky podle A − operand, ale A nechá být.

Režim	Zápis	Opkód	Bajtů	T	Příznaky
`#n`	`CMP #n`	`0xB8`	2	6	`C Z N`
`zp`	`CMP addr8`	`0xB9`	2	7	`C Z N`
`abs`	`CMP addr16`	`0xBA`	3	10	`C Z N`
`abs,X`	`CMP addr16,X`	`0xBB`	3	12	`C Z N`
`abs,Y`	`CMP addr16,Y`	`0xBC`	3	12	`C Z N`
`(zp)`	`CMP (addr8)`	`0xBD`	2	11	`C Z N`
`zp,X`	`CMP addr8,X`	`0xBE`	2	8	`C Z N`
`(zp),Y`	`CMP (addr8),Y`	`0xBF`	2	14	`C Z N`

`CMP` provede odečtení `A − operand`, zahodí výsledek a ponechá si jen příznaky. Je to jediný způsob, jak se v SAP-3/16 ptát „je A menší?“, aniž by se akumulátor rozbil.

Příznaky po `CMP`:

Příznak	Význam
`Z`	`A == operand`
`C`	výpůjčka: nastaven, právě když `A < operand` (bez znaménka)
`N`	nejvyšší bit rozdílu
`V`	**nemění se** — `CMP` přetečení nepočítá

Konvence u příznaku `C` je opačná než u 6502: tady je `C` **výpůjčka**, tedy „odečtení podteklo“. Aby se z toho nemusela dělat vědomost, má assembler aliasy, které se čtou tak, jak člověk uvažuje:

Zápis	Skutečná instrukce	Skočí, když
`JEQ`	alias pro `JZ`	`A == operand`
`JNE`	alias pro `JNZ`	`A != operand`
`JLT`	alias pro `JC`	`A < operand` (bez znaménka)
`JGE`	alias pro `JNC`	`A >= operand` (bez znaménka)
`JGT`	vlastní opkód	`A > operand` (bez znaménka)
`JLE`	vlastní opkód	`A <= operand` (bez znaménka)

První čtyři řádky jsou jen jiná jména pro tytéž opkódy — assembler je přeloží na `JZ`, `JNZ`, `JC` a `JNC`. `JGT` a `JLE` jsou naproti tomu samostatné instrukce se složenou podmínkou (`C v Z`), protože ta se z jednoho příznaku poskládat nedá.

```
        LDA pocet
        CMP #10
        JLT malo        ; pocet < 10
        JEQ presne      ; pocet == 10
                        ; sem se propadne pro pocet > 10
```

Mikrokód — `CMP zp  (0xB9)`

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

Poslední krok je jediný, kde se něco děje: ALU v režimu `CMP` odečte a `FI` zapíše příznaky. Chybějící `AI` je celý rozdíl oproti `SUB` — výsledek se nikam neuloží.

Past

`CMP` nenastavuje `V`, takže znaménkové skoky `JGES` a `JLTS` po něm čtou **starou** hodnotu přetečení a můžou odpovědět špatně. Pro znaménkové porovnání se používá `SUB` nebo `SBC`, které `V` nastaví:

```
        LDA #-100       ; 0x9C
        SUB #100        ; přeteče: N=0, V=1
        JLTS mensi      ; skočí — −100 < 100 se znaménkem
```

Cena je, že `SUB` na rozdíl od `CMP` přepíše `A`.

Past

Operand `CMP` končí — jako u každé binární operace ALU — v registru **B**. Cokoli tam bylo předtím, je pryč.

Viz také: `SUB`, `SBC`, `BIT`, `CPX`, `CPY`

CPX

Porovná registr X s operandem. Akumulátor zůstane nedotčený.

Režim	Zápis	Opkód	Bajtů	T	Příznaky
`#n`	`CPX #n`	`0x58`	2	7	`C Z N`
`zp`	`CPX addr8`	`0x59`	2	8	`C Z N`
`abs`	`CPX addr16`	`0x5A`	3	11	`C Z N`

Totéž co `CMP`, jen se porovnává **X**. Nastaví `C`, `Z` a `N` podle `X − operand`; **X** ani **A** se nemění.

Typicky ukončuje smyčku počítanou indexem:

```
smycka: LDA pole,X
        STA cil,X
        INX
        CPX #DELKA
        JNE smycka
```

Mikrokód — `CPX #n  (0x58)`

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

Na výpisu je vidět, jak se **A** vyhne: index jde přes **TMP** a multiplexor `ATM` ho pustí do ALU místo akumulátoru.

> **Past**
>
> **B** se přepíše i tady. `CPX` je vůči akumulátoru nedestruktivní, vůči **B** ne.

Viz také: `CPY`, `CMP`, `INX`, `DEX`

CPY

Porovná registr Y s operandem. Akumulátor zůstane nedotčený.

Režim	Zápis	Opkód	Bajtů	T	Příznaky
`#n`	`CPY #n`	`0x78`	2	7	`C Z N`
`zp`	`CPY addr8`	`0x79`	2	8	`C Z N`
`abs`	`CPY addr16`	`0x7A`	3	11	`C Z N`

Zrcadlový obraz instrukce `CPX` pro registr **Y**. Nastaví `C`, `Z` a `N` podle `Y − operand`.

Viz také: `CPX`, `CMP`, `INY`, `DEY`

DEC

Sníží o jedna: akumulátor, nebo bajt přímo v paměti.

Režim	Zápis	Opkód	Bajtů	T	Příznaky
implied	`DEC`	`0x09`	1	3	`Z N`
`zp`	`DEC addr8`	`0xD2`	2	7	`Z N`
`abs`	`DEC addr16`	`0xD3`	3	10	`Z N`

Má dva tvary. Bez operandu pracuje nad **A** s adresou jde o operaci čti-uprav-zapiš přímo v paměti — akumulátor se přitom nepoužije vůbec.

```
        DEC             ; A := A − 1
        DEC citac       ; sníží bajt přímo v paměti
```

Nastaví `Z` a `N`, `C` nechá být. Přetečení pod nulu tedy **není** vidět v příznaku `C`; hodnota se prostě zabalí z `0x00` na `0xFF`.

Paměťový tvar je základ počítané smyčky, která nesahá na registry:

```
smycka: ; ... tělo ...
        DEC citac
        JNZ smycka
```

Viz také: `INC`, `DEW`, `DEX`, `DEY`

DEW

Sníží o jedna šestnáctibitové slovo v paměti. Nemění příznaky.

Režim	Zápis	Opkód	Bajtů	T	Příznaky
`zp`	`DEW addr8`	`0xD6`	2	10	*žádné*
`abs`	`DEW addr16`	`0xD7`	3	13	*žádné*

`DEW zp` (nebo `DEW abs`) odečte jedničku od dvoubajtového slova uloženého dolním bajtem napřed. Výpůjčka mezi bajty se řeší uvnitř instrukce přes záklopku **`addrCarry`**.

Žádný příznak se nemění — ani `Z`. Test na nulu se musí udělat ručně:

```
        DEW delka
        LDA delka
        OR  delka+1     ; obě poloviny nulové?
        JZ  hotovo
```

Past

Chybějící `Z` je nejčastější překvapení u `INW` / `DEW`. Je to důsledek pravidla, že operace nad ukazateli nemají rozbíjet příznaky — ale znamená to, že šestnáctibitový čítač nejde ukončit jedním skokem.

Viz také: `INW`, `SBW`, `DEC`

DEX

Sníží registr X o jedna.

Režim	Zápis	Opkód	Bajtů	T	Příznaky
implied	`DEX`	`0x11`	1	4	`Z N`

`X := X - 1`, nastaví `Z` a `N`. Akumulátor se nepoužije — index prochází přes skrytý **`TMP`**.

Viz také: `INX`, `DEY`, `CPX`

DEY

Sníží registr Y o jedna.

Režim	Zápis	Opkód	Bajtů	T	Příznaky
implied	`DEY`	`0x13`	1	4	`Z N`

`Y := Y - 1`, nastaví `Z` a `N`.

Viz také: `INY`, `DEX`, `CPY`

DI

Zakáže přerušení (vynuluje příznak I).

Režim	Zápis	Opkód	Bajtů	T	Příznaky
implied	`DI`	`0x07`	1	3	`I`

Po `DI` procesor nepřijme hardwarové přerušení, dokud ho `EI` znovu nepovolí. Instrukce `BRK` se tím ale zakázat nedá — ta není událost, ale instrukce.

Používá se k ochraně kritické sekce, typicky při čtení vícebajtové hodnoty, kterou obsluha přerušení mění:

```
        DI
        LDA cas_lo
        LDX cas_hi
        EI
```

Viz také: `EI`, `RTI`, `HLT`, `BRK`

DIV

Vydělí A registrem B: podíl do A, zbytek do B. Dělení nulou nespadne.

Režim	Zápis	Opkód	Bajtů	T	Příznaky
implied	`DIV`	`0x29`	1	4	`C Z N`

`DIV` je bezoperandová instrukce — oba vstupy si bere z registrů. Dělenec je v **A**, dělitel v **B** po provedení je v **A** podíl a v **B** zbytek. Obojí je osmibitové a celočíselné.

Protože **B** není přímo adresovatelný žádnou instrukcí typu „nahraj“, plní se přes akumulátor instrukcí `TAB`. Odtud typické pořadí, které vypadá pozpátku, ale je jediné správné:

```
        LDA delitel
        TAB             ; B := dělitel
        LDA delenec     ; A := dělenec
        DIV             ; A := podíl, B := zbytek
        STA podil
        TBA             ; A := zbytek
        STA zbytek
```

Mikrokód — `DIV (0x29)`

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

Dva kroky za fetchem: první nechá ALU spočítat podíl, pošle ho na sběrnici do **A** a zapíše příznaky, druhý vytáhne přes `EXO` sekundární výsledek — zbytek — do **B**. Stejnou dvojici používá i `MUL`, jen tam je sekundárním výsledkem horní bajt součinu.

Dělení nulou

Dělení nulou v SAP-3/16 neshazuje procesor a nemá vlastní vektor. Místo toho se chová jako operace, která oznámí, že nic neudělala:

Po `DIV` s `B = 0`	Stav
`C`	1 — příznak chyby
`A`	beze změny (dělenec zůstává)
`B`	beze změny (nula zůstává)
`Z`, `N`	**nezmění se** — operace je nedefinovala

Při úspěšném dělení je naopak `C = 0`. Test se tak píše jedním skokem hned za instrukcí:

```
        DIV
        JC  deleni_nulou
```

> **Past**
>
> `Z` a `N` po dělení nulou drží hodnotu z **předchozí** operace, ne z `DIV`. Je to důsledek obecného pravidla, že příznakové hradlo zapíše jen ty příznaky, které operace definovala — a dělení nulou nedefinuje žádné kromě `C`. Kdo chce po neúspěšném dělení testovat nulovost, musí si `A` otestovat sám.

> **Past**
>
> Podíl i zbytek jsou osmibitové, takže `DIV` neumí dělit šestnáctibitová čísla. Dělení většího rozsahu se skládá po bajtech ze `SHR` a `SBC`.

Viz také: `MUL`, `TAB`, `TBA`, `SBC`

EI

Povolí přerušení (nastaví příznak I).

Režim	Zápis	Opkód	Bajtů	T	Příznaky
implied	`EI`	`0x06`	1	3	`I`

Hlavní vypínač přerušení. Aby se přerušení skutečně přijalo, musí být navíc povolený jeho zdroj v registru `IRQ_ENABLE`.

```
        LDA #0b00000001
        STA IRQ_ENABLE  ; povol časovač
        EI              ; a odemkni přerušení jako taková
```

Poznámka

Po resetu je `I` vynulované, takže program musí `EI` zavolat sám. Naopak po vstupu do obsluhy přerušení se `I` vynuluje automaticky a `RTI` ho obnoví — obsluha nemusí dělat nic.

Viz také: `DI`, `RTI`, `HLT`

HLT

Zastaví procesor. Povolené přerušení ho zase probudí.

Režim	Zápis	Opkód	Bajtů	T	Příznaky
implied	`HLT`	`0x02`	1	3	*žádné*

`HLT` není nutně konec programu. Když je přerušení povolené a některý zdroj je aktivní, procesor se probudí a provede obsluhu; po `RTI` pokračuje instrukcí za `HLT`.

Skutečné zastavení nastane jen tehdy, když je `I` nula nebo není povolený žádný zdroj — tak se `HLT` používá na konci programu.

```
        EI
cekej:  HLT             ; spí, dokud něco nepřijde
        JMP cekej
```

Poznámka

Zastavený procesor neznamená zastavený stroj. Fronta terminálu se dál vyprazdňuje, takže text zapsaný těsně před `HLT` se ještě dopíše — přesně jako skutečná sériová linka, která dokončí větu po tom, co procesor přestane pracovat.

Viz také: `EI`, `DI`, `NOP`

INC

Zvýší o jedna: akumulátor, nebo bajt přímo v paměti.

Režim	Zápis	Opkód	Bajtů	T	Příznaky
implied	`INC`	`0x08`	1	3	`Z N`
`zp`	`INC addr8`	`0xD0`	2	7	`Z N`
`abs`	`INC addr16`	`0xD1`	3	10	`Z N`

Zrcadlový obraz instrukce `DEC`. Bez operandu pracuje nad **A**, s adresou jde o operaci čti-uprav-zapiš v paměti.

Nastaví `Z` a `N`, `C` nechá být — přetečení z `0xFF` na `0x00` se v příznaku `C` neprojeví. Kdo potřebuje přenos, sečte instrukcí `ADD #1`.

Viz také: `DEC`, `INW`, `INX`, `INY`, `ADD`

INW

Zvýší o jedna šestnáctibitové slovo v paměti. Nemění příznaky.

Režim	Zápis	Opkód	Bajtů	T	Příznaky
`zp`	`INW addr8`	`0xD4`	2	10	*žádné*
`abs`	`INW addr16`	`0xD5`	3	13	*žádné*

Krokování ukazatele o jedna. Slovo je uložené dolním bajtem napřed a přenos mezi bajty se řeší uvnitř instrukce.

```
        LDA (ptr)       ; přečti bajt, na který se ukazuje
        INW ptr         ; a posuň ukazatel na další
```

Nemění žádný příznak — ani `Z`.

Viz také: `DEW`, `ADW`, `INC`

INX

Zvýší registr X o jedna.

Režim	Zápis	Opkód	Bajtů	T	Příznaky
implied	`INX`	`0x10`	1	4	`Z N`

`X := X + 1`, nastaví `Z` a `N`. Nejčastější instrukce průchodu polem:

```
smycka: LDA data,X
        STA cil,X
        INX
        CPX #DELKA
        JNE smycka
```

Viz také: `DEX`, `INY`, `CPX`

INY

Zvýší registr Y o jedna.

Režim	Zápis	Opkód	Bajtů	T	Příznaky
implied	`INY`	`0x12`	1	4	`Z N`

`Y := Y + 1`, nastaví `Z` a `N`. Používá se hlavně s režimem `(zp),Y`, kde **Y** slouží jako posun uvnitř bufferu.

Viz také: `DEY`, `INX`, `CPY`

JC

Skočí, když je nastavený příznak C. Po porovnání znamená „menší než“.

Režim	Zápis	Opkód	Bajtů	T	Příznaky
`abs`	`JC addr16`	`0xC3`	3	7	*žádné*

Podmíněný skok podle příznaku přenosu. Protože `C` je po odečítání **výpůjčka**, po instrukci `CMP` znamená `C = 1` právě „A bylo menší“. Čitelnější jméno pro tenhle případ je alias `JLT`.

Mimo porovnání se `JC` používá tam, kde přenos znamená přetečení: po `ADD`, po `SHL`, po `DIV` (kde signalizuje dělení nulou).

> **Poznámka**
>
> Podmíněný skok trvá 7 T bez ohledu na to, jestli se provedl. Neprovedený skok nešetří čas — doba běhu programu tak nezávisí na datech.

Viz také: `JNC`, `CMP`, `JLT`, `SHL`

JEQ

Skočí, když se porovnávané hodnoty rovnaly.

Alias pro `JZ` — assembler ho přeloží na tentýž opkód, kódování i časování jsou shodné.

JGE

Skočí, když bylo A větší nebo rovno operandu (bez znaménka).

Alias pro `JNC` — assembler ho přeloží na tentýž opkód, kódování i časování jsou shodné.

JGES

Skočí, když je hodnota větší nebo rovna — se znaménkem.

Režim	Zápis	Opkód	Bajtů	T	Příznaky
`abs`	`JGES addr16`	`0xE2`	3	7	*žádné*

Znaménková obdoba instrukce `JGE`. Skočí, když `N` a `V` mají stejnou hodnotu.

Znaménkové porovnání se nedá udělat po `CMP`, protože `CMP` nenastavuje `V`. Páruje se proto se `SUB` nebo `SBC`:

```
        LDA teplota
        SUB #0          ; nastaví N i V
        JGES nezaporna
```

Past

Po `CMP` čte `JGES` starou hodnotu `V` z nějaké dřívější instrukce a odpoví nesmysl. Je to tichá chyba — program se přeloží i poběží.

Viz také: `JLTS`, `SUB`, `JGE`, `CMP`

JGT

Skočí, když bylo A ostře větší než operand (bez znaménka).

Režim	Zápis	Opkód	Bajtů	T	Příznaky
`abs`	`JGT addr16`	`0xE0`	3	7	*žádné*

Složená podmínka: skočí, když **není** nastavené `C` ani `Z` — tedy „ani menší, ani rovno“. Není to alias; je to samostatný opkód, protože z jednoho příznaku se tahle podmínka poskládat nedá.

```
        LDA skore
        CMP #100
        JGT rekord
```

Viz také: `JLE`, `CMP`, `JGE`

JLE

Skočí, když bylo A menší nebo rovno operandu (bez znaménka).

Režim	Zápis	Opkód	Bajtů	T	Příznaky
`abs`	`JLE addr16`	`0xE1`	3	7	*žádné*

Opak instrukce `JGT`: skočí, když je nastavené `C` **nebo** `Z`. Také samostatný opkód, ne alias.

Viz také: `JGT`, `CMP`, `JLT`

JLT

Skočí, když bylo A menší než operand (bez znaménka).

Alias pro `JC` — assembler ho přeloží na tentýž opkód, kódování i časování jsou shodné.

JLTS

Skočí, když je hodnota menší — se znaménkem.

Režim	Zápis	Opkód	Bajtů	T	Příznaky
`abs`	`JLTS addr16`	`0xE3`	3	7	*žádné*

Skočí, když se `N` a `V` liší. Stejně jako `JGES` se páruje se `SUB` nebo `SBC`, ne s `CMP`.

```
        LDA #-100       ; 0x9C
        SUB #100        ; přeteče: N=0, V=1 → N≠V
        JLTS mensi      ; skočí, a je to správně
```

Bez `JLTS` by se znaménkové porovnání muselo psát jako dvojice skoků podle `N` a `V` — proto tahle instrukce v ISA 3.1 přibyla.

Viz také: `JGES`, `SUB`, `JN`

JMP

Nepodmíněný skok.

Režim	Zápis	Opkód	Bajtů	T	Příznaky
`abs`	`JMP addr16`	`0xC8`	3	7	*žádné*
`(zp)`	`JMP (addr8)`	`0xC9`	2	9	*žádné*

Dvě formy. Přímá skočí na adresu zapsanou v instrukci. Nepřímá (`JMP (zp)`) vezme cílovou adresu z ukazatele v nulté stránce — to je způsob, jak se dělá tabulka skoků nebo stavový automat:

```
        LDA stav
        SHL             ; index × 2 (adresa má dva bajty)
        TAX
        LDA tabulka,X
        STA cil
        LDA tabulka+1,X
        STA cil+1
        JMP (cil)
```

Žádný příznak se nemění.

Viz také: `CALL`, `JZ`, `JNZ`

JN

Skočí, když je nastavený příznak N (nejvyšší bit výsledku).

Režim	Zápis	Opkód	Bajtů	T	Příznaky
`abs`	`JN addr16`	`0xC5`	3	7	*žádné*

Po operaci nad čísly se znaménkem znamená `N = 1` „záporné“. Po `CMP` je `JN` spolehlivé jen tehdy, když nedošlo k přetečení — obecný znaménkový test je `JLTS`.

Viz také: `JP`, `JLTS`, `JGES`

JNC

Skočí, když příznak C není nastavený. Po porovnání znamená „větší nebo rovno“.

Režim	Zápis	Opkód	Bajtů	T	Příznaky
`abs`	`JNC addr16`	`0xC2`	3	7	*žádné*

Opak instrukce `JC`. Čitelnější jméno pro použití po porovnání je alias `JGE`.

Viz také: `JC`, `JGE`, `CMP`

JNE

Skočí, když se porovnávané hodnoty nerovnaly.

Alias pro `JNZ` — assembler ho přeloží na tentýž opkód, kódování i časování jsou shodné.

JNV

Skočí, když příznak přetečení není nastavený.

Režim	Zápis	Opkód	Bajtů	T	Příznaky
`abs`	`JNV addr16`	`0xC6`	3	7	*žádné*

Opak instrukce `JV`. Používá se po znaménkové aritmetice ke kontrole, že se výsledek do bajtu vešel.

Viz také: `JV`, `ADD`, `SUB`

JNZ

Skočí, když příznak Z není nastavený — tedy když poslední výsledek nebyl nula.

Režim	Zápis	Opkód	Bajtů	T	Příznaky
`abs`	`JNZ addr16`	`0xC0`	3	7	*žádné*

Spolu s `JZ` nejpoužívanější podmíněný skok stroje — ve zdrojových kódech v repozitáři na ně připadají zhruba dvě třetiny všech podmíněných skoků. Ukončuje počítané smyčky:

```
        LDX #10
smycka: ; ... tělo ...
        DEX
        JNZ smycka
```

Po porovnání se čte jako „nerovná se“ a má pro ten případ alias `JNE`.

Viz také: `JZ`, `JNE`, `DEC`, `DEX`

JP

Skočí, když příznak N není nastavený — tedy když je výsledek kladný nebo nula.

Režim	Zápis	Opkód	Bajtů	T	Příznaky
`abs`	`JP addr16`	`0xC4`	3	7	*žádné*

Zkratka od **jump if positive**.

> **Past**
>
> Jméno `JP` znamená u některých jiných procesorů (třeba Z80) nepodmíněný skok. Tady je to **podmíněný** skok podle znaménkového bitu; nepodmíněný skok se jmenuje `JMP`. Je to nejsnadnější překlep v celé instrukční sadě, protože se přeloží bez chyby.

Viz také: `JN`, `JMP`, `JGES`

JV

Skočí, když je nastavený příznak přetečení.

Režim	Zápis	Opkód	Bajtů	T	Příznaky
`abs`	`JV addr16`	`0xC7`	3	7	*žádné*

`V` nastavuje sčítání a odečítání a znamená, že výsledek nesedí, když operandy čteme se znaménkem. Logické operace, `CMP` ani posuvy `V` nemění.

Viz také: `JNV`, `ADD`, `SUB`, `CMP`

JZ

Skočí, když je nastavený příznak Z — tedy když byl poslední výsledek nula.

Režim	Zápis	Opkód	Bajtů	T	Příznaky
`abs`	`JZ addr16`	`0xC1`	3	7	*žádné*

Po porovnání se čte jako „rovná se“ a má pro ten případ alias `JEQ`. Hodí se ale i bez porovnání, protože `Z` nastavuje skoro každá operace včetně nahrání do registru:

```
        LDA stav
        JZ  nic_se_nedeje   ; není potřeba CMP #0
```

Viz také: `JNZ`, `JEQ`, `LDA`, `BIT`

LDA

Nahraje bajt do akumulátoru. Jediná instrukce, která umí všech osm adresních režimů.

Režim	Zápis	Opkód	Bajtů	T	Příznaky
`#n`	`LDA #n`	`0x40`	2	5	`Z N`
`zp`	`LDA addr8`	`0x41`	2	6	`Z N`
`abs`	`LDA addr16`	`0x42`	3	9	`Z N`
`abs,X`	`LDA addr16,X`	`0x43`	3	11	`Z N`
`abs,Y`	`LDA addr16,Y`	`0x44`	3	11	`Z N`
`(zp)`	`LDA (addr8)`	`0x45`	2	10	`Z N`
`zp,X`	`LDA addr8,X`	`0x46`	2	7	`Z N`
`(zp),Y`	`LDA (addr8),Y`	`0x47`	2	13	`Z N`

`LDA` je nejpoužívanější instrukce celého stroje — spolu s `STA` tvoří zhruba polovinu všech instrukcí ve skutečných programech. Umí kompletní sadu adresních režimů, což z ní dělá nejlepší místo, kde si je prohlédnout pohromadě.

```
        LDA #42          ; konstanta
        LDA hodnota      ; z paměti (zp/abs zvolí assembler)
        LDA tabulka,X    ; prvek pole
        LDA (ukazatel)   ; přes ukazatel v nulté stránce
        LDA (radek),Y    ; přes ukazatel, plus offset v Y
```

Instrukce nastaví `Z` a `N` podle nahraného bajtu, takže se hned za ní dá větvit bez porovnávání:

```
        LDA stav
        JZ  nic_se_nedeje
```

Cena režimů

Rozdíl mezi režimy je vidět na počtu T-stavů v tabulce nahoře a stojí za to si ho zapamatovat: nultá stránka je o tři T-stavy levnější než absolutní adresa, protože se nemusí načíst ani zpracovat druhý bajt adresy. Nejdražší je `(zp),Y` — přečíst ukazatel a přičíst k němu index zabere víc než dvojnásobek času oproti `zp`.

Mikrokód — `LDA zp  (0x41)`

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

Mikrokód — `LDA (zp),Y  (0x47)`

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

Na druhém výpisu je vidět, kam ten čas jde: osm kroků za fetchem jen skládá adresu, a teprve devátý přečte hodnotu. Dvojice `ACO` a `AHC` je přenos mezi bajty adresy — `ACO` si uschová přenos z dolního bajtu do záklopky **`addrCarry`**, `AHC` ho přičte k hornímu. Díky tomu zvládne procesor šestnáctibitový součet po osmibitových kouscích, aniž by na to potřeboval registr programátora.

> **Poznámka**
>
> Právě proto je nultá stránka v SAP-3/16 popisovaná jako „soubor ukazatelů“. Není to jen rychlejší paměť — je to jediné místo, kde můžou ležet ukazatele pro režimy `(zp)` a `(zp),Y`.

> **Past**
>
> Mřížka u konstanty není volitelná. `LDA 42` znamená „nahraj obsah adresy 42“, `LDA #42` znamená „nahraj číslo 42“. Assembler obojí přeloží bez varování, protože obojí je legální.

> **Past**
>
> `LDA` nemění `C` ani `V`. Nahrání hodnoty tedy nezruší výsledek předchozího porovnání — což se dá využít, ale taky to znamená, že `C` může být „starý“, když se na něj člověk spolehne po dlouhé sekvenci přesunů.

Viz také: `STA`, `LDX`, `LDY`, `BIT`

LDX

Nahraje bajt do registru X.

Režim	Zápis	Opkód	Bajtů	T	Příznaky
`#n`	`LDX #n`	`0x48`	2	5	`Z N`
`zp`	`LDX addr8`	`0x49`	2	6	`Z N`
`abs`	`LDX addr16`	`0x4A`	3	9	`Z N`
`abs,Y`	`LDX addr16,Y`	`0x4C`	3	11	`Z N`

Nastaví `Z` a `N`. Nabídka adresních režimů je užší než u `LDA` — a ta asymetrie má logiku: `LDX` umí indexovat registrem **Y**, ne sebou samým.

```
        LDX #0
        LDX delka
        LDX tabulka,Y   ; index druhým indexovým registrem
```

Past

Neexistuje `LDX zp,X` ani `LDX zp,Y` — režim `zp,Y` v ISA vůbec není. Kdo potřebuje krátkou indexovanou formu, má ji jen u `LDY` (`LDY zp,X`).

Viz také: `LDY`, `STX`, `TAX`, `TXA`

LDY

Nahraje bajt do registru Y.

Režim	Zápis	Opkód	Bajtů	T	Příznaky
`#n`	`LDY #n`	`0x50`	2	5	`Z N`
`zp`	`LDY addr8`	`0x51`	2	6	`Z N`
`abs`	`LDY addr16`	`0x52`	3	9	`Z N`
`abs,X`	`LDY addr16,X`	`0x53`	3	11	`Z N`
`zp,X`	`LDY addr8,X`	`0x56`	2	7	`Z N`

Zrcadlový obraz instrukce `LDX`, ale s jinou sadou režimů: `LDY` umí indexovat registrem **X**, a to i v krátké formě `zp,X`.

Viz také: `LDX`, `STY`, `TAY`, `TYA`

LXY

Nahraje šestnáctibitovou konstantu do dvojice X:Y. Nemění příznaky.

Režim	Zápis	Opkód	Bajtů	T	Příznaky
`#n`	`LXY #nn`	`0xEA`	3	8	*žádné*

`LXY #imm16` rozdělí šestnáctibitovou konstantu na dva bajty: horní jde do **X**, dolní do **Y**. Je to stejná konvence, jakou používá `TXYS` pro zásobníkový ukazatel — **X** drží horní polovinu.

Bez téhle instrukce by nastavení adresy zabralo dvě `LDA` / `TAX` / `LDY` sekvence; takhle je to jedna instrukce o třech bajtech.

```
        LXY #buffer     ; X:Y := adresa bufferu
        SXY ptr         ; ulož jako ukazatel do zp
```

Žádný příznak se nemění.

Viz také: `SXY`, `TXYS`, `TSXY`, `ADW`

MUL

Vynásobí A registrem B: dolní bajt součinu do A, horní do B.

Režim	Zápis	Opkód	Bajtů	T	Příznaky
implied	`MUL`	`0x28`	1	4	`C Z N`

Jako `DIV` je i `MUL` bezoperandová — oba činitele si bere z **A** a **B**, druhý činitel se tam dostane instrukcí `TAB`.

```
        LDA sirka
        TAB             ; B := šířka
        LDA vyska
        MUL             ; A := dolní bajt součinu, B := horní
        STA plocha_lo
        TBA
        STA plocha_hi
```

Příznak `C` je nastavený, právě když je horní bajt nenulový — tedy když se součin nevešel do jednoho bajtu. `Z` a `N` se řídí **dolním** bajtem.

> **Past**
>
> `Z` po `MUL` neznamená „součin je nula“. Znamená „dolní bajt součinu je nula“, což platí například pro 16 × 16 = 256. Test na skutečnou nulu musí vzít v úvahu i `C` nebo horní bajt.

Viz také: `DIV`, `TAB`, `TBA`, `SHL`

NOP

Neudělá nic. Trvá tři T-stavy.

Režim	Zápis	Opkód	Bajtů	T	Příznaky
implied	`NOP`	`0x01`	1	3	*žádné*

Jediná instrukce bez efektu. Používá se k vyplnění místa (třeba po vyjmutí kódu, aby se nemusely přepočítat adresy) a k jemnému časování.

> **Poznámka**
>
> `NOP` trvá 3 T, ne 2. Důvod je konstrukční: první dva kroky mikroprogramu jsou fetch, který ale patří **následující** instrukci, a žádný mikroprogram nesmí být kratší než tři kroky. `NOP` má proto za fetchem jeden prázdný krok navíc.
>
> Kratší instrukce v ISA neexistuje — 3 T je dolní hranice pro cokoli.

Viz také: `HLT`

NOT

Znegová všechny bity akumulátoru.

Režim	Zápis	Opkód	Bajtů	T	Příznaky
implied	`NOT`	`0x0E`	1	3	`Z N`

`A := ~A`. Nastaví `Z` a `N`; `C` ani `V` se nemění.

Spolu s `ADD #1` tvoří změnu znaménka:

```
        NOT
        ADD #1          ; A := −A (dvojkový doplněk)
```

Viz také: `AND`, `OR`, `XOR`, `SUB`

OR

Logický součet akumulátoru s operandem po bitech.

Režim	Zápis	Opkód	Bajtů	T	Příznaky
`#n`	`OR #n`	`0xA8`	2	6	`Z N`
`zp`	`OR addr8`	`0xA9`	2	7	`Z N`
`abs`	`OR addr16`	`0xAA`	3	10	`Z N`
`abs,X`	`OR addr16,X`	`0xAB`	3	12	`Z N`
`abs,Y`	`OR addr16,Y`	`0xAC`	3	12	`Z N`
`(zp)`	`OR (addr8)`	`0xAD`	2	11	`Z N`
`zp,X`	`OR addr8,X`	`0xAE`	2	8	`Z N`
`(zp),Y`	`OR (addr8),Y`	`0xAF`	2	14	`Z N`

`A := A | operand`. Nastaví `Z` a `N`, `C` a `V` nechá být.

Používá se k nastavení bitů a — ve dvojici — k testu, jestli je šestnáctibitová hodnota nulová:

```
        LDA delka
        OR  delka+1     ; Z = 1, jen když jsou oba nulové
        JZ  prazdne
```

Viz také: `AND`, `XOR`, `NOT`

OUT

Zkopíruje akumulátor do výstupního registru.

Režim	Zápis	Opkód	Bajtů	T	Příznaky
implied	`OUT`	`0x03`	1	3	*žádné*

`OUT` pošle obsah **A** do registru OUT — jednoduchého výstupního portu, který v emulátoru zobrazuje displej a který zachytává i výstup nástroje `npm run asm`. Je to nejstarší způsob, jak z programu něco dostat ven, a pochází ještě z původních SAP strojů.

Nemění žádný příznak.

```
        LDA vysledek
        OUT             ; ukaž číslo na displeji
        HLT
```

> **Poznámka**
>
> Na rozdíl od terminálu je `OUT` čistě numerický a jednobajtový — žádná fronta, žádné čekání, žádný ASCII překlad. Pro text slouží terminál na adrese `TERM_DATA`.

Viz také: `HLT`

PHB

Uloží registr B na zásobník.

Režim	Zápis	Opkód	Bajtů	T	Příznaky
implied	`PHB`	`0x2A`	1	4	*žádné*

Bez téhle instrukce nešlo **B** uložit jinak než přes akumulátor, čímž se zničil. `PHB` proto přibylo v ISA 3.1 a je klíčové pro obsluhy přerušení: každá binární operace ALU (i `CMP #0`) přepíše **B**, takže obsluha, která sáhne na ALU, musí **B** zachránit.

```
obsluha:
        PUSH            ; A
        PHB             ; B
        ; ... tělo ...
        PLB
        POP
        RTI
```

Nemění příznaky.

Viz také: `PLB`, `PUSH`, `TAB`, `RTI`

PHF

Uloží stavový bajt (příznaky) na zásobník.

Režim	Zápis	Opkód	Bajtů	T	Příznaky
implied	`PHF`	`0x26`	1	4	*žádné*

Uloží příznaky `C`, `Z`, `N`, `V` a `I` zabalené do jednoho bajtu — ve stejném formátu, jaký ukládá vstup do přerušení.

Nemění příznaky.

Viz také: `PLF`, `RTI`, `PUSH`

PHX

Uloží registr X na zásobník.

Režim	Zápis	Opkód	Bajtů	T	Příznaky
implied	`PHX`	`0x22`	1	4	*žádné*

Nemění příznaky.

Viz také: `PLX`, `PHY`, `PUSH`

PHY

Uloží registr Y na zásobník.

Režim	Zápis	Opkód	Bajtů	T	Příznaky
implied	`PHY`	`0x24`	1	4	*žádné*

Nemění příznaky.

Viz také: `PLY`, `PHX`, `PUSH`

PLB

Vyzvedne registr B ze zásobníku.

Režim	Zápis	Opkód	Bajtů	T	Příznaky
implied	`PLB`	`0x2B`	1	4	`Z N`

Protějšek instrukce `PHB`. Nastaví `Z` a `N` podle vyzvednutého bajtu.

> **Past**
>
> To, že `PLB` nastavuje příznaky, může překvapit v obsluze přerušení. Vadí to ale jen zdánlivě: `RTI` hned poté obnoví celý stavový bajt ze zásobníku, takže se změna zahodí. Problém by nastal jen tehdy, kdyby se `PLB` použilo mimo obsluhu mezi porovnáním a skokem.

Viz také: `PHB`, `POP`, `TBA`

PLF

Obnoví příznaky ze zásobníku.

Režim	Zápis	Opkód	Bajtů	T	Příznaky
implied	`PLF`	`0x27`	1	4	`C Z N V I`

Vyzvedne bajt a rozbalí z něj **všech pět** příznaků včetně `I` — na rozdíl od `POP`, který jen odvodí `Z` a `N` z hodnoty. Je to tentýž mechanismus, jaký používá `RTI`.

> **Past**
>
> `PLF` obnovuje i příznak `I`, takže může nečekaně povolit nebo zakázat přerušení. Kdo ukládá příznaky uprostřed kritické sekce, musí počítat s tím, že je při obnovení dostane zpátky celé.

Viz také: `PHF`, `RTI`, `POP`

PLX

Vyzvedne registr X ze zásobníku.

Režim	Zápis	Opkód	Bajtů	T	Příznaky
implied	`PLX`	`0x23`	1	4	`Z N`

Nastaví `Z` a `N` podle vyzvednuté hodnoty.

Viz také: `PHX`, `PLY`, `POP`

PLY

Vyzvedne registr Y ze zásobníku.

Režim	Zápis	Opkód	Bajtů	T	Příznaky
implied	`PLY`	`0x25`	1	4	`Z N`

Nastaví `Z` a `N` podle vyzvednuté hodnoty.

Viz také: `PHY`, `PLX`, `POP`

POP

Vyzvedne akumulátor ze zásobníku.

Režim	Zápis	Opkód	Bajtů	T	Příznaky
implied	`POP`	`0x21`	1	4	`Z N`

Přečte bajt z vrcholu zásobníku do **A** a zvýší **SP**. Nastaví `Z` a `N` podle vyzvednuté hodnoty.

Viz také: `PUSH`, `PLB`, `PLX`, `PLY`

PUSH

Uloží akumulátor na zásobník.

Režim	Zápis	Opkód	Bajtů	T	Příznaky
implied	`PUSH`	`0x20`	1	4	*žádné*

Sníží **SP** a zapíše na novou adresu obsah **A**. Uložení je tedy **předdekrementové** — **SP** po něm ukazuje přímo na uložený bajt.

Nemění příznaky.

```
        PUSH            ; odlož A
        ; ... něco, co A přepíše ...
        POP             ; a vrať ho
```

Viz také: `POP`, `PHX`, `PHY`, `PHB`, `PHF`

RET

Návrat z podprogramu.

Režim	Zápis	Opkód	Bajtů	T	Příznaky
implied	`RET`	`0xCB`	1	6	*žádné*

Vyzvedne ze zásobníku dolní a horní bajt návratové adresy a skočí na ni. Nemění příznaky, takže podprogram může vracet výsledek nejen v registru, ale i v příznacích:

```
je_prazdny:
        LDA delka
        OR  delka+1
        RET             ; Z nese odpověď
```

Past

`RET` nekontroluje, že na zásobníku opravdu leží návratová adresa. Když podprogram uloží něco na zásobník a zapomene to vyzvednout, `RET` skočí na data — a program se utrhne obvykle až o kus dál, s hlášením o ilegálním opkódu na úplně jiném místě.

Viz také: `CALL`, `RTI`, `PUSH`, `POP`

RND

Vloží do akumulátoru náhodný bajt.

Režim	Zápis	Opkód	Bajtů	T	Příznaky
implied	`RND`	`0x0F`	1	3	*žádné*

Zdrojem je generátor xorshift32, jehož semínko lze nastavit — takže běh programu je reprodukovatelný. Nástroj `npm run asm` k tomu má přepínač `--seed`.

```
        RND
        AND #0x0F       ; náhodné číslo 0–15
```

Past

`RND` nenastavuje žádný příznak, ani `Z`. Test na konkrétní hodnotu proto potřebuje `CMP` — samotné `RND` následované `JZ` skáče podle výsledku **předchozí** instrukce.

Poznámka

Generátor je součástí stavu procesoru a ukládá se do žurnálu. Návrat v čase tedy vrátí i náhodná čísla — přehrání téhož úseku dá tytéž hodnoty.

Viz také: `AND`, `CMP`

ROL

Rotace vlevo přes příznak C: akumulátor, nebo bajt v paměti.

Režim	Zápis	Opkód	Bajtů	T	Příznaky
implied	`ROL`	`0x0C`	1	3	`C Z N`
`zp`	`ROL addr8`	`0xDC`	2	7	`C Z N`
`abs`	`ROL addr16`	`0xDD`	3	10	`C Z N`

Bit 7 jde do `C`, `C` jde do bitu 0. Na rozdíl od `SHL` se tedy nic neztratí — hodnota se otáčí dokola přes příznak přenosu.

Hlavní použití je posun čísel širších než bajt. `SHL` na nejnižším bajtu vysune bit do `C`, `ROL` na vyšších ho zase vtáhne:

```
        SHL hodnota     ; dolní bajt, vysunutý bit do C
        ROL hodnota+1   ; horní bajt, C vtáhne dovnitř
```

Nastaví `C`, `Z` a `N`.

Viz také: `ROR`, `SHL`, `SHR`, `CLC`, `SEC`

ROR

Rotace vpravo přes příznak C: akumulátor, nebo bajt v paměti.

Režim	Zápis	Opkód	Bajtů	T	Příznaky
implied	`ROR`	`0x0D`	1	3	`C Z N`
`zp`	`ROR addr8`	`0xDE`	2	7	`C Z N`
`abs`	`ROR addr16`	`0xDF`	3	10	`C Z N`

Bit 0 jde do `C`, `C` jde do bitu 7. Vícebajtový posun vpravo se skládá obráceně než u `ROL` — od nejvyššího bajtu k nejnižšímu:

```
        SHR hodnota+1   ; horní bajt
        ROR hodnota     ; dolní bajt
```

Nastaví `C`, `Z` a `N`.

Viz také: `ROL`, `SHR`, `SHL`

RTI

Návrat z obsluhy přerušení.

Režim	Zápis	Opkód	Bajtů	T	Příznaky
implied	`RTI`	`0xCC`	1	8	`C Z N V I`

Vyzvedne ze zásobníku stavový bajt a pak dolní a horní bajt návratové adresy — tedy přesně to, co tam uložil vstup do přerušení nebo instrukce `BRK`.

Obnoví **všech pět** příznaků včetně `I`.

> **Poznámka**
>
> `RTI` přerušení nepovoluje natvrdo. Že jsou po návratu povolená, plyne z toho, že uložený stavový bajt měl `I` ještě nastavené — signál `DIF` se při ukládání uplatní až po tom, co bajt opustí sběrnici.
>
> Prakticky to znamená, že obsluha smí uprostřed použít `DI` a `EI`, jak se jí zlíbí; po `RTI` bude platit stav, který byl **před** přerušením.

Viz také: `BRK`, `EI`, `DI`, `RET`, `PLF`

SBC

Odečte operand od akumulátoru včetně výpůjčky. Vyšší bajty vícebajtového rozdílu.

Režim	Zápis	Opkód	Bajtů	T	Příznaky
`#n`	`SBC #n`	`0x98`	2	6	`C Z N V`
`zp`	`SBC addr8`	`0x99`	2	7	`C Z N V`
`abs`	`SBC addr16`	`0x9A`	3	10	`C Z N V`
`abs,X`	`SBC addr16,X`	`0x9B`	3	12	`C Z N V`
`abs,Y`	`SBC addr16,Y`	`0x9C`	3	12	`C Z N V`
`(zp)`	`SBC (addr8)`	`0x9D`	2	11	`C Z N V`
`zp,X`	`SBC addr8,X`	`0x9E`	2	8	`C Z N V`
`(zp),Y`	`SBC (addr8),Y`	`0x9F`	2	14	`C Z N V`

`A := A − operand − C`. Příznak `C` je tu **výpůjčka**, ne přenos.

Rozdělení práce je stejné jako u `ADD` / `ADC`: nejnižší bajt odečti instrukcí `SUB`, vyšší instrukcí `SBC`.

```
        LDA a_lo
        SUB b_lo        ; C := výpůjčka
        STA rozdil_lo
        LDA a_hi
        SBC b_hi
        STA rozdil_hi
```

Nastaví `C`, `Z`, `N` i `V`.

Viz také: `SUB`, `ADC`, `SBW`, `CMP`

SBW

Odečte osmibitovou konstantu od šestnáctibitového slova v paměti. Nemění příznaky.

Režim	Zápis	Opkód	Bajtů	T	Příznaky
`zp`	`SBW addr8,#n`	`0xE9`	3	13	*žádné*

Protějšek instrukce `ADW`. Odečte konstantu od dvoubajtového slova na adrese `zp` a výsledek zapíše zpátky; výpůjčka mezi bajty se řeší uvnitř instrukce.

```
        SBW ptr, #40    ; ukazatel o řádek výš
```

Žádný příznak se nemění.

Viz také: `ADW`, `DEW`, `SBC`

SEC

Nastaví příznak přenosu.

Režim	Zápis	Opkód	Bajtů	T	Příznaky
implied	`SEC`	`0x05`	1	3	`C`

`C := 1`. Používá se hlavně k přípravě vstupního bitu pro `ROL` nebo `ROR`. Před odečítáním potřeba není — `SUB` výpůjčku na vstupu ignoruje.

Viz také: `CLC`, `ROL`, `ROR`, `SBC`

SHL

Posun vlevo o jeden bit: akumulátor, nebo bajt v paměti.

Režim	Zápis	Opkód	Bajtů	T	Příznaky
implied	`SHL`	`0x0A`	1	3	`C Z N`
`zp`	`SHL addr8`	`0xD8`	2	7	`C Z N`
`abs`	`SHL addr16`	`0xD9`	3	10	`C Z N`

Bit 7 jde do `C`, zprava se nasouvá nula. Je to násobení dvěma — a nejlevnější způsob, jak na tomhle stroji násobit mocninou dvojky.

```
        LDA index
        SHL             ; × 2
        SHL             ; × 4
```

Nastaví `C`, `Z` a `N`.

> **Poznámka**
>
> Paměťový tvar (`SHL zp`, `SHL abs`) pracuje přímo v paměti a akumulátoru se nedotkne. Právě kvůli vícebajtovým posunům existuje — dvojice `SHL` dolní bajt, `ROL` horní bajt je základní stavební kámen šestnáctibitové aritmetiky.

Viz také: `SHR`, `ROL`, `MUL`

SHR

Posun vpravo o jeden bit: akumulátor, nebo bajt v paměti.

Režim	Zápis	Opkód	Bajtů	T	Příznaky
implied	`SHR`	`0x0B`	1	3	`C Z N`
`zp`	`SHR addr8`	`0xDA`	2	7	`C Z N`
`abs`	`SHR addr16`	`0xDB`	3	10	`C Z N`

Bit 0 jde do `C`, zleva se nasouvá nula. Je to celočíselné dělení dvěma bez znaménka.

Nastaví `C`, `Z` a `N`.

> **Past**

Posun je **logický**, ne aritmetický — nejvyšší bit se nekopíruje. U čísel se znaménkem tedy `SHR` nedělí dvěma správně: z `0xFE` (−2) udělá `0x7F` (127), ne `0xFF` (−1).

Znaménkový posun se musí poskládat ručně. Trik je dostat původní bit 7 do `C` a pak použít `ROR`, který ho zase zasune zpátky nahoru:

```
        CLC
        BIT #0x80       ; Z = 1, právě když je bit 7 nulový
        JZ  .kladne
        SEC             ; záporné: C := 1
.kladne: ROR            ; posun, do bitu 7 se vrátí znaménko
```

Tohle je zároveň jedno z mála míst, kde se `CLC` a `SEC` opravdu hodí.

Viz také: `SHL`, `ROR`, `DIV`

STA

Uloží akumulátor do paměti.

Režim	Zápis	Opkód	Bajtů	T	Příznaky
`zp`	`STA addr8`	`0x61`	2	6	*žádné*
`abs`	`STA addr16`	`0x62`	3	9	*žádné*
`abs,X`	`STA addr16,X`	`0x63`	3	11	*žádné*
`abs,Y`	`STA addr16,Y`	`0x64`	3	11	*žádné*
`(zp)`	`STA (addr8)`	`0x65`	2	10	*žádné*
`zp,X`	`STA addr8,X`	`0x66`	2	7	*žádné*
`(zp),Y`	`STA (addr8),Y`	`0x67`	2	13	*žádné*

Druhá polovina nejčastější dvojice instrukcí. Umí všech sedm paměťových režimů — chybí jen immediate, který by nedával smysl.

```
        STA hodnota
        STA obrazovka,X
        STA (radek),Y
```

Nemění žádný příznak. To je důležité: uložení mezi porovnáním a skokem výsledek porovnání nezruší.

Viz také: `LDA`, `STX`, `STY`

STX

Uloží registr X do paměti.

Režim	Zápis	Opkód	Bajtů	T	Příznaky
`zp`	`STX addr8`	`0x69`	2	6	*žádné*
`abs`	`STX addr16`	`0x6A`	3	9	*žádné*

Jen dva režimy: `zp` a `abs`. Nemění příznaky.

> **Past**
>
> `STX tabulka,Y` neexistuje. Uložení indexovaného prvku se musí udělat přes akumulátor (`TXA` a `STA tabulka,Y`), což zničí **A**.

Viz také: `LDX`, `STY`, `STA`, `TXA`

STY

Uloží registr Y do paměti.

Režim	Zápis	Opkód	Bajtů	T	Příznaky
`zp`	`STY addr8`	`0x71`	2	6	*žádné*
`abs`	`STY addr16`	`0x72`	3	9	*žádné*

Jen `zp` a `abs`, stejně jako `STX`. Nemění příznaky.

Viz také: `LDY`, `STX`, `STA`, `TYA`

SUB

Odečte operand od akumulátoru. Výpůjčku na vstupu ignoruje, na výstupu nastaví.

Režim	Zápis	Opkód	Bajtů	T	Příznaky
`#n`	`SUB #n`	`0x90`	2	6	`C Z N V`
`zp`	`SUB addr8`	`0x91`	2	7	`C Z N V`
`abs`	`SUB addr16`	`0x92`	3	10	`C Z N V`
`abs,X`	`SUB addr16,X`	`0x93`	3	12	`C Z N V`
`abs,Y`	`SUB addr16,Y`	`0x94`	3	12	`C Z N V`
`(zp)`	`SUB (addr8)`	`0x95`	2	11	`C Z N V`
`zp,X`	`SUB addr8,X`	`0x96`	2	8	`C Z N V`
`(zp),Y`	`SUB (addr8),Y`	`0x97`	2	14	`C Z N V`

`A := A − operand`. Příznak `C` po operaci znamená **výpůjčku** — je nastavený, právě když odečtení podteklo pod nulu.

Nastaví `C`, `Z`, `N` a `V`. Právě to `V` je důvod, proč se znaménkové skoky `JGES` a `JLTS` párují se `SUB`, a ne s `CMP`.

```
        LDA a
        SUB b
        JC  podteklo
```

Viz také: `SBC`, `ADD`, `CMP`, `JLTS`, `JGES`

SXY

Uloží dvojici X:Y jako šestnáctibitové slovo do nulté stránky. Nemění příznaky.

Režim	Zápis	Opkód	Bajtů	T	Příznaky
`zp`	`SXY addr8`	`0xEB`	2	8	*žádné*

`SXY zp` zapíše **Y** na adresu `zp` a **X** na `zp+1` — tedy dolním bajtem napřed, jak se v tomhle stroji ukládají všechna slova.

Spolu s `LXY` je to nejkratší způsob, jak nastavit ukazatel:

```
        LXY #buffer
        SXY ptr         ; ptr nyní ukazuje na buffer
        LDA (ptr),Y
```

Viz také: `LXY`, `TXYS`, `TSXY`, `INW`

TAB

Zkopíruje akumulátor do registru B.

Režim	Zápis	Opkód	Bajtů	T	Příznaky
implied	`TAB`	`0x1C`	1	3	`Z N`

Jediný způsob, jak do **B** něco dostat cíleně. Používá se ke stagingu operandu pro `MUL` a `DIV`, které si druhý činitel berou právě odtud.

```
        LDA delitel
        TAB
        LDA delenec
        DIV
```

Nastaví `Z` a `N` podle přenesené hodnoty.

> **Past**
>
> Hodnota v **B** je krátkodobá. Přepíše ji **každá** binární operace ALU — `ADD`, `SUB`, `AND`, `OR`, `XOR`, `CMP`, `BIT` i `ADW`. Mezi `TAB` a `MUL` / `DIV` proto nesmí být nic, co sahá na ALU. Kdo potřebuje **B** přenést přes delší úsek, uloží ho instrukcí `PHB`.

Viz také: `TBA`, `MUL`, `DIV`, `PHB`, `PLB`

TAX

Zkopíruje akumulátor do registru X.

Režim	Zápis	Opkód	Bajtů	T	Příznaky
implied	`TAX`	`0x18`	1	3	`Z N`

Nastaví `Z` a `N`. Typický způsob, jak spočítaný index dostat tam, kde ho umí použít adresní režim:

```
        LDA stav
        SHL
        TAX
        LDA tabulka,X
```

Viz také: `TXA`, `TAY`, `LDX`

TAY

Zkopíruje akumulátor do registru Y.

Režim	Zápis	Opkód	Bajtů	T	Příznaky
implied	`TAY`	`0x1A`	1	3	`Z` `N`

Nastaví `Z` a `N`.

Viz také: `TYA`, `TAX`, `LDY`

TBA

Zkopíruje registr B do akumulátoru.

Režim	Zápis	Opkód	Bajtů	T	Příznaky
implied	`TBA`	`0x1D`	1	3	`Z` `N`

Způsob, jak si vyzvednout sekundární výsledek: po `MUL` je v **B** horní bajt součinu, po `DIV` zbytek po dělení.

```
        DIV
        TBA             ; A := zbytek
```

Nastaví `Z` a `N`.

Viz také: `TAB`, `MUL`, `DIV`

TSXY

Přečte zásobníkový ukazatel do dvojice X:Y. Nemění příznaky.

Režim	Zápis	Opkód	Bajtů	T	Příznaky
implied	`TSXY`	`0x1F`	1	4	*žádné*

X dostane horní polovinu **SP**, **Y** dolní — stejná konvence jako u `LXY` a `TXYS`.

Používá se k uložení stavu zásobníku, nebo k přístupu k parametrům předaným na zásobníku.

> **Poznámka**

> Přenos proběhne ve dvou T-stavech (šestnáctibitová hodnota po osmibitové sběrnici), ale přerušení se přijímá jen na hranici instrukce — takže se nemůže stát, že by obsluha viděla **SP** rozpůlený.

Viz také: `TXYS`, `LXY`, `SXY`

TXA

Zkopíruje registr X do akumulátoru.

Režim	Zápis	Opkód	Bajtů	T	Příznaky
implied	`TXA`	`0x19`	1	3	`Z N`

Nastaví `Z` a `N`. Nutný krok tam, kde chybí indexovaná forma uložení — například `STX tabulka,Y` neexistuje, takže se to musí obejít přes **A**.

Viz také: `TAX`, `TYA`, `STX`

TXYS

Nastaví zásobníkový ukazatel z dvojice X:Y. Nemění příznaky.

Režim	Zápis	Opkód	Bajtů	T	Příznaky
implied	`TXYS`	`0x1E`	1	4	*žádné*

`SP := X:Y`, kde **X** je horní polovina. Obraz instrukce `TSXY`.

Používá se při inicializaci, když program chce zásobník jinde než na výchozí adrese, a při návratu z chyby, kdy je potřeba zásobník „uklidit“ najednou.

```
        LXY #0x0200
        TXYS            ; SP zpět na výchozí hodnotu
```

Past

`TXYS` nekontroluje nic. Nastavení **SP** doprostřed dat je legální a projeví se až tím, že první `CALL` přepíše, co tam bylo.

Viz také: `TSXY`, `LXY`, `PUSH`, `POP`

TYA

Zkopíruje registr Y do akumulátoru.

Režim	Zápis	Opkód	Bajtů	T	Příznaky
implied	`TYA`	`0x1B`	1	3	`Z N`

Nastaví `Z` a `N`.

Viz také: `TAY`, `TXA`, `STY`

XOR

Nonekvivalence akumulátoru s operandem po bitech.

Režim	Zápis	Opkód	Bajtů	T	Příznaky
`#n`	`XOR #n`	`0xB0`	2	6	`Z N`
`zp`	`XOR addr8`	`0xB1`	2	7	`Z N`
`abs`	`XOR addr16`	`0xB2`	3	10	`Z N`
`abs,X`	`XOR addr16,X`	`0xB3`	3	12	`Z N`
`abs,Y`	`XOR addr16,Y`	`0xB4`	3	12	`Z N`
`(zp)`	`XOR (addr8)`	`0xB5`	2	11	`Z N`
`zp,X`	`XOR addr8,X`	`0xB6`	2	8	`Z N`
`(zp),Y`	`XOR (addr8),Y`	`0xB7`	2	14	`Z N`

`A := A ^ operand`. Nastaví `Z` a `N`, `C` a `V` nechá být.

Dvě obvyklá použití: převrácení vybraných bitů maskou a test na rovnost, který zároveň řekne, **ve kterých bitech** se hodnoty liší.

```
        LDA stav
        XOR #0b00000001 ; přepni nejnižší bit
        STA stav
```

> **Poznámka**
>
> Znalci jiných architektur tu budou hledat trik „vynuluj registr sám sebou“. Nefunguje — `XOR` nemá tvar s registrovým operandem, takže nejde napsat „`XOR` s **A**“. Nulování se dělá prostě instrukcí `LDA #0`, která je stejně dlouhá a rychlejší než cokoli jiného.

Viz také: `AND`, `OR`, `NOT`, `BIT`

ČÁST IV

Periferie a systém

KAPITOLA 16

Mapa MMIO a pravidla přístupu

Periferie v SAP-3/16 nemají vlastní instrukce. Sedí v paměti — v bloku 0xFF00–0xFF7F — a mluví se s nimi obyčejným `LDA` a `STA`. Procesor ty adresy zachytí dřív, než se z nich stane paměť, a předá je zařízení.

Praktický důsledek: všechno, co umíš s pamětí, umíš i s periferiemi. Registr zařízení jde adresovat indexovaně, dá se na něj dát watchpoint a jeho hodnotu je vidět ve výpisu paměti.

Jak se dekóduje adresa

Blok má 128 bajtů a dělí se na osm zařízení po šestnácti bajtech. Číslo zařízení jsou bity 6 až 4 adresy:

```
0xFF00 + (zařízení << 4) + registr
```

Základ	Č.	Zařízení
`0xFF00`	0	Klávesnice
`0xFF10`	1	Terminál
`0xFF20`	2	Zvuk
`0xFF30`	3	Časovač a počitadla
`0xFF40`	4	Řízení přerušení
`0xFF50`	5	Video
`0xFF60`	6	Úložiště a sériová linka
`0xFF70`	7	Gamepady a blitter

Obrázek 11. Rozdělení bloku periferií na osm zařízení.

Ne každé zařízení svých šestnáct bajtů zaplní, a volné místo se dvakrát hodilo: počitadla bydlí v horní půlce slotu časovače, sériová linka v horní půlce slotu úložiště a blitter v horní půlce slotu gamepadů.

> **Poznámka**

Nepřiřazené adresy uvnitř bloku čtou nulu a zápis ignorují. Program tedy nemá jak zjistit, že sáhl vedle — proto se registry vždycky pojmenovávají konstantami `EQU` a nepíšou se čísla přímo.

Pravidla, která platí napříč zařízeními

Zařízení se liší, ale několik konvencí je společných a vyplatí se je znát dřív než jednotlivé registry.

Chybové bity se čtou jednou

Stavové registry s chybovým bitem (`DSK_STATUS`, `BLT_STATUS`, `SER_STATUS`, `TERM_STATUS`) se čtením **mažou**. Program proto musí přečíst registr jednou, uložit si hodnotu a testovat z ní všechny bity:

```
.cekej: LDA DSK_STATUS
        STA tmp             ; jeden odečet, oba bity odtud
        AND #0x01
        JZ  .cekej
        LDA tmp
        AND #0x02
        JNZ chyba
```

Past

Nejčastější chyba při práci s periferiemi je přečíst stavový registr dvakrát — jednou na test připravenosti, podruhé na test chyby. Druhý odečet už chybu nevidí, protože ji smazal ten první.

Vysílač nemusí být připraven

Terminál i sériová linka mají frontu, která se vyprazdňuje vlastním tempem. Zápis do plné fronty **bajt zahodí** a nastaví bit přetečení. Krátký shluk projde bez ptaní, trvalý výstup ne — před každým zápisem se má číst bit připravenosti.

Přenos trvá

Úložiště potvrdí příkaz okamžitě, ale data se přesunou až po chvíli. Program musí čekat na bit připravenosti, ne počítat takty. Naopak blitter je hotový uvnitř instrukce, která ho spustil.

Úrovňové versus hranové

Rozdíl, na kterém stojí návrh vstupu:

Typ	Chování
úrovňový	registr ukazuje **současný** stav; čtení ho nemění (gamepady, bit připravenosti)
hranový	událost se ve frontě drží, dokud si ji program nevyzvedne; čtení ji spotřebuje (klávesnice)

Hra proto čte gamepad každý snímek a nezajímá ji historie, zatímco textový vstup si vybírá klávesy z fronty a nesmí o žádnou přijít.

Co je a co není součástí stavu stroje

Skoro všechno ve stroji je součástí žurnálu, a tedy se s návratem v čase vrací. Tři věci ne, a všechny tři ze stejného důvodu — jsou to **média nebo okolní svět**, ne stav procesoru:

Nevrací se	Proč
obsah disku	krok zpět přes zápis na disk ho neodepíše — stejně jako u skutečného hardwaru
odeslané bajty na sériové lince	co odešlo protějšku, nejde vzít zpět
připojení linky	kabel přežije reset i převinutí

Naopak registry všech zařízení, obsah front, odpočty rozpracovaných přenosů i držené klávesy gamepadu ve stavu stroje jsou — takže se přehrají přesně.

> **Poznámka**
>
> Z toho plyne jedna nezvyklá vlastnost ladění: když se program vrátí v čase před odeslání bajtu a pošle ho znovu, protějšek ho uvidí dvakrát. Časová osa se větví na téhle straně kabelu, ne na druhé.

Adresy zařízení

Zbytek části IV probírá zařízení po jednom. Pro rychlé nahlédnutí je tady kompletní seznam registrů podle zařízení.

Klávesnice

Adresa	Registr
`0xFF00`	`KBD_DATA`
`0xFF01`	`KBD_STATUS`

Řízení přerušení

Adresa	Registr
`0xFF40`	`IRQ_ENABLE`
`0xFF41`	`IRQ_STATUS`

Terminál

Adresa	Registr
0xFF10	TERM_DATA
0xFF11	TERM_STATUS

Zvuk

Adresa	Registr
0xFF20	AUD_CTRL
0xFF21	AUD_FREQ_LO
0xFF22	AUD_FREQ_HI
0xFF23	AUD_WAVE
0xFF24	AUD_AD
0xFF25	AUD_SR

Časovač a počitadla

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

Video

Adresa	Registr
0xFF50	VID_CTRL
0xFF51	VID_STATUS
0xFF52	VID_CURX
0xFF53	VID_CURY
0xFF54	VID_BORDER
0xFF55	VID_SCROLLX
0xFF56	VID_SCROLLY
0xFF57	VID_MCOLOR
0xFF58	SPR_ENABLE

Úložiště a sériová linka

Adresa	Registr
0xFF60	DSK_CMD
0xFF61	DSK_STATUS
0xFF62	DSK_BLK
0xFF63	DSK_ADDR_LO
0xFF64	DSK_ADDR_HI
0xFF65	SER_DATA
0xFF66	SER_STATUS
0xFF67	SER_CTRL
0xFF68	DSK_DISK

Gamepady a blitter

Adresa	Registr
0xFF70	PAD1
0xFF71	PAD2
0xFF72	BLT_SRC_LO
0xFF73	BLT_SRC_HI
0xFF74	BLT_DST_LO
0xFF75	BLT_DST_HI
0xFF76	BLT_LEN_LO
0xFF77	BLT_LEN_HI

Adresa	Registr
`0xFF78`	`BLT_FILL`
`0xFF79`	`BLT_CTRL`
`0xFF7A`	`BLT_STATUS`

KAPITOLA 17

Klávesnice a gamepady

Stroj má dvě zařízení pro vstup a liší se ne technologií, ale **modelem času**. Klávesnice dodává události, gamepad stav. Který z nich je pro danou úlohu správný, se pozná podle jediné otázky: vadí, když program o jeden stisk přijde?

Klávesnice

Adresa	Registr
`0xFF00`	`KBD_DATA`
`0xFF01`	`KBD_STATUS`

`KBD_DATA` obsahuje **ASCII kód** stisknuté klávesy. Ne scancode — překlad dělá řadič klávesnice, ještě než se bajt k procesoru dostane. Tisknutelné znaky mají svůj kód, Enter je 0x0D, Backspace 0x08, Escape 0x1B.

Jsou to **jen události stisku**. Puštění klávesy se nikam nehlásí a délka stisku není nijak vidět.

`KBD_STATUS` má v bitu 0 příznak „čeká nepřečtený znak“. Čtení `KBD_DATA` znak spotřebuje — vyprázdní buffer, shodí příznak i případné čekající přerušení.

```
cti_znak:
        LDA KBD_STATUS
        AND #0x01
        JZ  cti_znak        ; nic nečeká, čekej dál
        LDA KBD_DATA        ; a tímhle ho spotřebuj
        RET
```

Past

Buffer je **jednobajtový**. Když uživatel stiskne druhou klávesu dřív, než program přečte první, ta první je nenávratně pryč. Program, který mezi čteními dělá něco dlouhého, ztrácí znaky.

Řešením je buď číst klávesnici přerušením (bit 1 v `IRQ_ENABLE`), nebo přijmout, že rychlé psaní se neuloží celé.

Klávesnice přes přerušení

Zdroj přerušení je **úrovňový**: drží se, dokud je v bufferu nepřečtený znak. Obsluha ho tedy musí přečíst, jinak se přerušení vyvolá znovu hned po návratu.

```
        LDA #0b00000010     ; bit 1 = klávesnice
        STA IRQ_ENABLE
        EI

obsluha:
        PUSH
        PHB
        LDA KBD_DATA        ; přečtení zdroj přerušení uklidí
        CALL uloz_do_fronty
        PLB
        POP
        RTI
```

Gamepady

Adresa	Registr
`0xFF70`	`PAD1`
`0xFF71`	`PAD2`
`0xFF72`	`BLT_SRC_LO`
`0xFF73`	`BLT_SRC_HI`
`0xFF74`	`BLT_DST_LO`
`0xFF75`	`BLT_DST_HI`
`0xFF76`	`BLT_LEN_LO`
`0xFF77`	`BLT_LEN_HI`
`0xFF78`	`BLT_FILL`
`0xFF79`	`BLT_CTRL`
`0xFF7A`	`BLT_STATUS`

Dva nezávislé osmibitové registry, jeden na hráče. Nastavený bit znamená „tlačítko je **právě teď** stisknuté“. Registry jsou úrovňové — čtení je nemaže a číst je lze kolikrát chce.

Bit	Tlačítko	Bit	Tlačítko
0	nahoru	4	Fire / A
1	dolů	5	B

Bit	Tlačítko	Bit	Tlačítko
2	vlevo	6	Start
3	vpravo	7	Select

Typické použití je jedno čtení za snímek:

```
        LDA PAD1
        AND #0x04           ; vlevo?
        JZ  .nevlevo
        DEC hrac_x
.nevlevo:
```

Hra pro jednoho hráče, která má přijmout oba ovladače, si je sloučí jedinou instrukcí:

```
        LDA PAD1
        OR  PAD2
```

Gamepad **nevyvolává přerušení**. Je určený k dotazování a nic jiného nedává smysl — hra stejně kreslí každý snímek.

> **Poznámka**
>
> Proč vedle sebe dvě vstupní zařízení? Protože z klávesnice nejde poznat, že je klávesa **držená**. Na hostitelském systému se opakování stisků řídí nastavením uživatele a v emulátoru navíc jeho prohlížečem; plynulý pohyb postavy by tím pádem záležel na věcech, se kterými program nic nenadělá. Gamepad ten problém odstraňuje tím, že hlásí stav, ne události.

V emulátoru se na porty mapuje hostitelská klávesnice podle fyzických kláves: WASD, mezerník a levý Shift na `PAD1`, šipky, Enter a pravý Shift na `PAD2`. Na desce jsou to dva skutečné konektory.

> **Poznámka**
>
> Oba registry jsou v emulátoru zapisovatelné jako testovací vstup — program si může držený stav nastavit a přečíst zpátky. Na hardwaru jsou jen pro čtení. Držený stav je součástí žurnálu, takže se s návratem v čase přehraje přesně.

KAPITOLA 18

Terminál a sériová linka

Dvě zařízení, která posílají bajty ven, a je snadné je zaměnit. Rozdíl je v tom, kdo je na druhém konci: terminál mluví s **člověkem u téhle mašiny**, sériová linka s **druhým strojem**.

Terminál

Adresa	Registr
`0xFF10`	`TERM_DATA`
`0xFF11`	`TERM_STATUS`

Zápis do `TERM_DATA` zařadí znak do vysílací fronty o velikosti 64 bajtů. Na obrazovku se znak dostane ve chvíli, kdy frontu **opustí** — jedna položka za `max(1, CLK_TPMS/8)` T-stavů, což je zhruba 8 000 znaků za sekundu reálného času na jakýchkoli hodinách nad 8 kHz.

```
        LDA #'A'
        STA TERM_DATA
```

`TERM_STATUS` je jen pro čtení:

Bit	Význam
0	vysílač připraven — fronta **není** plná
1	přetečení: zápis se zahodil (drží se, čtení maže)

Past

Vysílač není vždycky připraven. Zápis do plné fronty **bajt zahodí** a nastaví bit 1. Shluk do 64 znaků projde bez ptaní, delší výstup ne.

Správně napsaný výpis se proto ptá:

```
putc:   LDA TERM_STATUS
        AND #0x01
        JZ  putc            ; fronta plná, počkej
```

```
        LDA znak
        STA TERM_DATA
        RET
```

Řídicí znaky

Kód	Znak	Účinek
0x0A	`\n`	nový řádek
0x0D	`\r`	návrat na začátek řádku
0x08	`\b`	smaže předchozí znak
0x0C	`\f`	smaže obrazovku terminálu
0x09	`\t`	tabulátor po osmi sloupcích

Terminál si poradí se všemi obvyklými konvencemi konce řádku: `\n`, `\r\n` i `\n\r` udělají jeden nový řádek. Samotné `\r` vrátí kurzor na začátek řádku, což se hodí na přepisování — třeba ukazatel průběhu.

> **Poznámka**
>
> Fronta se vyprazdňuje dál i po instrukci `HLT`. Text zapsaný těsně před zastavením se tedy ještě dopíše — přesně jako skutečná sériová linka, která dokončí větu po tom, co procesor přestane pracovat.

Sériová linka

Adresa	Registr
`0xFF60`	`DSK_CMD`
`0xFF61`	`DSK_STATUS`
`0xFF62`	`DSK_BLK`
`0xFF63`	`DSK_ADDR_LO`
`0xFF64`	`DSK_ADDR_HI`
`0xFF65`	`SER_DATA`
`0xFF66`	`SER_STATUS`
`0xFF67`	`SER_CTRL`
`0xFF68`	`DSK_DISK`

Registry `SER_DATA`, `SER_STATUS` a `SER_CTRL` sdílejí slot se zařízením úložiště — bydlí v jeho volné horní půlce.

Linka je obousměrná, s frontami po 16 bajtech v každém směru.

Bit `SER_STATUS`	Význam
0	přijímací fronta není prázdná (úrovňový)
1	vysílací fronta není plná (úrovňový)
2	přetečení příjmu: bajt od protějšku se ztratil (drží se, čtení maže)
3	přetečení vysílání: zápis se zahodil (drží se, čtení maže)
4	linka je připojená (úrovňový)

Čtení `SER_DATA` vybere nejstarší přijatý bajt; prázdná fronta čte nulu, takže se má nejdřív testovat bit 0. Zápis zařadí bajt k odeslání.

`SER_CTRL` je příkazový registr (čte nulu): bit 0 vyprázdní příjem, bit 1 vysílání. Bity se dají kombinovat.

```
posli:  LDA SER_STATUS
        AND #0x02           ; vysílač připraven?
        JZ  posli
        LDA bajt
        STA SER_DATA
        RET

prijmi: LDA SER_STATUS
        AND #0x01           ; něco přišlo?
        JZ  prijmi
        LDA SER_DATA
        RET
```

Přerušení

Sériová linka umí vyvolat přerušení při příjmu — bit 2 v `IRQ_ENABLE`. Zdroj je **úrovňový**: drží se, dokud je v přijímací frontě aspoň jeden bajt, takže obsluha musí frontu vyprázdnit.

Přerušení „vysílač je volný“ schválně neexistuje. Na odesílání se čeká dotazováním bitu 1 — stejně jako u skutečného UARTu.

Rychlost

Jeden bajt opustí vysílací frontu každých `max(128, CLK_TPMS/8)` T-stavů. Na hodinách do 1 MHz je to nezměněných 128 T-stavů; výš se rychlost zastropuje zhruba na 8 kB/s reálného času.

Poznámka

Ten strop je záměrně **pomalejší** než skutečný UART na 115 200 baudech. Vede k tomu tatáž zásada jako u terminálu: program, který stíhá v emulátoru, bude stíhat i na desce. Opačné pořadí by znamenalo, že se chyby objeví až na hardwaru.

Kabel je sdílená sběrnice

Když je připojených víc strojů, slyší každý bajt všechny. Spojení dvou bodů je konvence, ne vlastnost zařízení.

Past

Návrat v čase se zastaví na hraně kabelu. Fronty, odpočet rozesílaného bajtu i příznaky přetečení jsou součástí stavu stroje, takže se přehrají přesně — ale **protějšek se nepřevíjí**. Co už odešlo, nejde vzít zpět, a když se program vrátí před odeslání a pošle znovu, protějšek uvidí bajt dvakrát.

Bajty, které přijdou během pauzy nebo převíjení, spadnou do fronty toho stavu, ve kterém stroj právě stojí — úplně stejně jako stisk klávesy.

KAPITOLA 19

Čas, časovač a přerušení

Stroj nabízí čtyři nezávislé časové základny a liší se v jedné podstatné věci: tři z nich se odvíjejí od T-stavů, a tedy zrychlí, když se zrychlí hodiny. Jen jedna je uvázaná na reálný čas.

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

Časovač

`TMR_DIV` nastavuje dělič, `TMR_CNT` čítač. Jakýkoli zápis do `TMR_CNT` ho znovu naplní. Když čítač dojde k nule, vyvolá přerušení (bit 0 v `IRQ_ENABLE`).

```
        LDA #100
        STA TMR_DIV
        STA TMR_CNT
        LDA #0b00000001     ; povol přerušení od časovače
        STA IRQ_ENABLE
        EI
```

Počitadla

Všechna počitadla jsou jen pro čtení, běží pořád a nulují se pouze při resetu.

Počitadlo	Šířka	Krok
`CYC0` – `CYC3`	32 b	jeden T-stav
`UPT_LO` / `UPT_HI`	16 b	jeden snímek obrazu
`CLK_TPMS_LO` / `HI`	16 b	nemění se — je to rychlost hodin v T-stavech na milisekundu
`CLK_MS_LO` / `HI`	16 b	jedna reálná milisekunda

Past

Čtení nejnižšího bajtu **zamkne snímek** celé hodnoty. Program tedy musí číst `CYC0` před `CYC1`, `UPT_LO` před `UPT_HI` a `CLK_MS_LO` před `CLK_MS_HI` — jinak dostane dvě poloviny z různých okamžiků a při přechodu přes 256 mu vyjde nesmysl.

Rozdíl mezi základnami je vidět, jakmile se změní rychlost hodin:

Základna	Chování při zrychlení hodin
`CYC`	tiká rychleji — je to počítadlo **práce**, ne času
`UPT`	tiká rychleji — snímek je odvozený od T-stavů
`CLK_MS`	tiká stejně — je uvázaná na reálný čas

Z toho plyne dělba práce: `CYC` slouží k **měření kódu** (kolik práce něco stálo), `CLK_MS` k **držení tempa** (jak dlouho se má čekat).

Snímkové tempo

Registr `VID_STATUS` má v bitu 0 příznak snímku, který se maže čtením. Perioda je 1000 T-stavů.

```
cekej_snimek:
        LDA VID_STATUS
        AND #0x01
        JZ  cekej_snimek
        RET
```

Tenhle způsob je jednoduchý, ale má vadu: rychlost snímků závisí na rychlosti hodin. Program, který se má chovat stejně v prohlížeči i na desce, se místo toho paceuje podle `CLK_MS`:

```
wait_frame:                     ; drž snímek na FRAME_MS ms
.spin:  LDA CLK_MS_LO           ; uplynulo = CLK_MS − kotva
        SUB ANCHOR
        STA EL
        LDA CLK_MS_HI
        SBC ANCHOR+1
        JNZ .done               ; přes 256 ms — dávno pozdě
        LDA EL
        CMP #FRAME_MS           ; C = 1, dokud je míň
        JC .spin
.done:  LDA CLK_MS_LO           ; kotva až při uvolnění
        STA ANCHOR
        LDA CLK_MS_HI
        STA ANCHOR+1
        RET
```

Poznámka

Kotva se posouvá až v okamžiku uvolnění, ne na pevný násobek. Kdyby se počítal absolutní termín, snímek, který se nestihl, by si nesl dluh, program by se ho snažil dohnat a počitadlo by nakonec přeteklo. Takhle je zpožděný snímek prostě zpožděný.

Řízení přerušení

Adresa	Registr
`0xFF40`	`IRQ_ENABLE`
`0xFF41`	`IRQ_STATUS`

`IRQ_ENABLE` povoluje jednotlivé zdroje:

Bit	Zdroj
0	časovač
1	klávesnice — čeká nepřečtený znak
2	sériová linka — v přijímací frontě něco je

Zapisují se jen dolní tři bity, zbytek se maskuje pryč.

`IRQ_STATUS` ukazuje, co je právě rozdělané; čtení ho maže.

Nad tím vším je hlavní vypínač — příznak `I`, ovládaný instrukcemi `EI` a `DI`. Aby se přerušení přijalo, musí být povolený zdroj **i** nastavené `I`.

> **Past**
>
> Zdroje se liší v tom, jak se uklidí. Časovač je hranový: přerušení nastane a tím to hasne. Klávesnice a sériová linka jsou **úrovňové** — drží se, dokud obsluha nepřečte `KBD_DATA`, resp. nevyprázdní přijímací frontu.
>
> Obsluha, která se zdroje nedotkne, se tedy zavolá znovu okamžitě po `RTI` a program se zacyklí, aniž by cokoli spadlo.

Podrobný popis vstupu do přerušení, rámce na zásobníku a instrukce `RTI` je v kapitole o přerušeních v části I; tahle kapitola popisuje jen zařízení, které je vyvolává.

KAPITOLA 20

Video I: text, glyfy a barvy

Video v SAP-3/16 nemá „příkaz nakresli“. Displej prostě **čte paměť** — a co tam program napíše, to se objeví. Rozdíl mezi „zapsat znak na obrazovku“ a „zapsat bajt do paměti“ tady neexistuje.

Adresa	Registr
`0xFF50`	`VID_CTRL`
`0xFF51`	`VID_STATUS`
`0xFF52`	`VID_CURX`
`0xFF53`	`VID_CURY`
`0xFF54`	`VID_BORDER`
`0xFF55`	`VID_SCROLLX`
`0xFF56`	`VID_SCROLLY`
`0xFF57`	`VID_MCOLOR`
`0xFF58`	`SPR_ENABLE`

Registrů je jen devět a žádný z nich nenese obrazová data. Ta jsou ve VRAM, což je obyčejná RAM: zápisy do ní jsou v žurnálu, dá se na ně dát watchpoint a dají se vrátit v čase.

Textový režim

Výchozí režim po resetu. Mřížka je 40 × 25 znaků, každý znak 8 × 8 pixelů — dohromady tedy 320 × 200 bodů, což je nativní rozlišení celého stroje.

Znakový buffer

Leží na 0x8000–0x83E7 a adresa buňky se počítá takhle:

```
adresa = 0x8000 + y × 40 + x
```

Napsat znak je tedy jeden zápis do paměti:

```
        LDA #'A'
        STA 0x8000          ; levý horní roh
```

Glyfy

Tvary znaků nejsou v ROM, ale v paměti na 0x8400–0x8BFF: 256 glyfů po osmi bajtech, jeden bajt na řádek, **nejvyšší bit vlevo**.

```
GLYF_A  EQU 0x8608          ; 0x8400 + 'A' * 8, spočítáno předem
        LDA #0xFF           ; plný řádek
        STA GLYF_A          ; první z osmi řádků znaku 'A'
```

> **Poznámka**
>
> Při resetu i při nahrání programu se glyfy naplní zabudovaným písmem — **pokud** nahraný obraz nepoložil do oblasti glyfů žádný nenulový bajt. Program, který si veze vlastní písmo, tedy vyhrává a nemusí nic vypínat.

Barevná RAM

Na 0x8C00–0x8FE7 leží jeden atributový bajt na buňku, se stejným rozvržením jako znakový buffer. Adresa barvy je adresa znaku plus 0x0C00 — což je pěkně praktické, protože se spočítaný ukazatel na obrazovku převede jediným `ADD #0x0C` na horním bajtu.

Půlbajt	Význam
dolní	barva popředí — index do palety
horní	barva pozadí — index do palety

Hodnota 0x00 je zvláštní: znamená **výchozí vzhled** (světlé na tmavém), ne černou na černé. Je to důležité, protože RAM se při resetu nuluje — program, který se barev nedotkne, tedy vypadá přesně jako dřív.

> **Past**
>
> Tahle výjimka stojí jednu kombinaci: skutečnou černou na černé nejde zapsat, protože 0x00 je obsazená. Kdo chce černý obdélník, nakreslí mezeru na černém pozadí.

Paleta

Šestnáct barev, sdílených barevnou RAM, registrem `VID_MCOLOR` a okrajem `VID_BORDER`.

Index	Barva	Index	Barva
0	#000000	8	#8b5429
1	#ffffff	9	#574200
2	#883932	10	#b86962
3	#67b6bd	11	#505050
4	#8b3f96	12	#787878
5	#55a049	13	#94e089
6	#40318d	14	#7869c4
7	#bfce72	15	#9f9f9f

Obrázek 12. Šestnáctibarevná paleta, inspirovaná Commodore 64.

Kurzor a okraj

`VID_CURX` a `VID_CURY` určují polohu blikajícího podtržítka. Kurzor **není uložený v bufferu** — kreslí ho displej — a objeví se jen tehdy, když je na mřížce, tedy `VID_CURX` < 40 a `VID_CURY` < 25.

Past

Oba registry se resetují na nulu, takže grafický program, který se jich nedotkne, ukazuje blikající kurzor v levém horním rohu. Vypíná se odparkováním mimo mřížku — konvenčně `VID_CURY = 25`:

```
        LDA #25
        STA VID_CURY        ; kurzor pryč
```

`VID_BORDER` vybírá barvu okraje z palety (maskuje se na dolní čtyři bity) a funguje ve všech režimech.

Přepínání režimů

`VID_CTRL` má v bitu 0 povolení obrazu a v bitech 1–2 režim:

Režim	Popis
0	text 40 × 25
1	bitmapa 160 × 100, 16 barev
2	bitmapa 320 × 200, monochromaticky

Režim	Popis
3	rezervováno — kreslí prázdný raster

Bitmapové režimy čtou jinou oblast VRAM než text, takže se mezi nimi dá přepínat sem a tam a textová obrazovka mezitím zůstane netknutá. Podrobně jsou popsané v následující kapitole.

> **Poznámka**
>
> Že je VRAM obyčejná paměť, má jeden nečekaně užitečný důsledek: znakový buffer se dá číst jako **mapa kolizí**. Hra se nemusí ptát „co je na souřadnici“, stačí přečíst bajt z obrazovky. Barevná RAM do toho nemluví — leží jinde a čtení znaku neovlivňuje.

KAPITOLA 21

Video II: bitmapy, sprity a blitter

Textový režim je pro znaky. Když se má kreslit po bodech, přepne se `VID_CTRL` a tatáž paměť se začne číst jinak.

Společný framebuffer

Oba bitmapové režimy čtou stejných 8000 bajtů od adresy 0x9000. Přepnutí režimu tedy paměť **nemaže**, jen ji začne interpretovat jinak.

Režim 1 — 160 × 100 v 16 barvách

Dva pixely na bajt, **horní půlbajt je levý pixel**. Půlbajt je přímo index do palety.

```
adresa = 0x9000 + y × 80 + x/2
```

Pixely jsou čtvercové — každý pokrývá blok 2 × 2 v nativním rastru 320 × 200.

Tady **neplatí** výjimka s hodnotou 0x00: vynulovaná paměť je černá, protože půlbajt vybírá jednu barvu, ne dvojici. Sentinel existuje jen tam, kde jeden bajt volí dvě barvy — tedy v barevné RAM a v `VID_MCOLOR`.

Režim 2 — 320 × 200 monochromaticky

Osm pixelů na bajt, nejvyšší bit vlevo.

```
adresa = 0x9000 + y × 40 + x/8
```

Nastavený bit se kreslí barvou popředí, nulový barvou pozadí; obě určuje `VID_MCOLOR` stejným způsobem jako atribut v barevné RAM (dolní půlbajt popředí, horní pozadí, 0x00 = výchozí vzhled).

Jemný scroll

`VID_SCROLLX` a `VID_SCROLLY` posouvají obraz o 0 až 7 bodů (zapisují se jen dolní tři bity). Není to posun paměti — je to **posun místa, odkud displej čte**, se zabalením:

```
výstupní bod (x, y) ukazuje bod ((x + SCROLLX) mod Š, (y + SCROLLY) mod V)
```

Osm kroků odpovídá přesně jedné znakové buňce, jednomu bajtu v režimu 2 nebo dvěma bajtům v režimu 1. Plynulý scroll se proto skládá ze dvou vrstev: jemný posun registrem, a jakmile dojde na osmičku, hrubý posun celé paměti blitterem.

> **Past**
>
> Scroll hýbe **jen hrací plochou**. Sprity, kurzor ani okraj se neposunou — jejich souřadnice jsou v souřadnicích obrazovky. To je pro hru obvykle to, co člověk chce (postava zůstane, kde je), ale překvapí to, když se scroll zkouší poprvé.

Sprity

Osm spritů, každý 16 × 16 „tučných“ pixelů — jeden takový pixel pokrývá blok 32 × 32 v nativním rastru. Kreslí se nad hrací plochou ve všech třech zobrazovacích režimech a pod kurzorem.

Všechna data spritů jsou obyčejná VRAM. Jediný registr je `SPR_ENABLE`, jehož bit **i** zapíná sprite **i**.

Tabulka atributů

Na 0xAFE0, čtyři bajty na sprite (sprite **i** na `0xAFE0 + i*4`):

Posun	Bajt	Význam
+0	X	poloha v souřadnicích 160 × 100, **posunutá o +16**
+1	Y	totéž svisle
+2	vzor	index vzoru, bere se modulo 32
+3	příznaky	bit 0 = převrátit vodorovně, bit 1 = svisle

Posun o +16 je trik, díky kterému jeden bajt pokryje všechny polohy včetně částečného zajetí za okraj: X = 16 je zarovnaný s levým okrajem, X = 0 je sprite úplně mimo obraz.

Paměť vzorů

Na 0xB000, 32 vzorů po 128 bajtech (vzor **p** na `0xB000 + p*128`): 16 řádků po osmi bajtech, dva čtyřbitové pixely na bajt, horní půlbajt vlevo — stejné balení jako v režimu 1.

Půlbajt 0 je průhledný, hodnoty 1–15 jsou indexy do palety.

```
        ; sprite 0 na střed obrazovky
        LDA #96             ; X = 80 + 16 (posun souřadnic)
        STA 0xAFE0
        LDA #66             ; Y = 50 + 16
        STA 0xAFE1
        LDA #0
        STA 0xAFE2          ; vzor 0
        STA 0xAFE3          ; bez převrácení
```

```
        LDA #0b00000001
        STA SPR_ENABLE
```

> **Poznámka**
>
> Sprite se zobrazuje jen tehdy, když je nastavený jeho bit v `SPR_ENABLE` — v atributech žádný povolovací bit není. Sprite 0 má nejvyšší prioritu, kreslí se přes sprity 1 až 7.

Blitter

Adresa	Registr
`0xFF70`	`PAD1`
`0xFF71`	`PAD2`
`0xFF72`	`BLT_SRC_LO`
`0xFF73`	`BLT_SRC_HI`
`0xFF74`	`BLT_DST_LO`
`0xFF75`	`BLT_DST_HI`
`0xFF76`	`BLT_LEN_LO`
`0xFF77`	`BLT_LEN_HI`
`0xFF78`	`BLT_FILL`
`0xFF79`	`BLT_CTRL`
`0xFF7A`	`BLT_STATUS`

Blitter je DMA jednotka, která sama zkopíruje nebo vyplní úsek paměti. Je to hardwarová obdoba `memcpy` a `memset`, za které by procesor jinak platil smyčkou `LDA`/`STA` na každý bajt.

Postup je vždycky stejný: naplnit adresy a délku, pak zapsat příkaz.

Registr	Význam
`BLT_SRC_LO` / `HI`	zdrojová adresa (jen pro kopírování)
`BLT_DST_LO` / `HI`	cílová adresa
`BLT_LEN_LO` / `HI`	délka v bajtech; nula je prázdná operace
`BLT_FILL`	bajt pro vyplnění
`BLT_CTRL`	1 = kopíruj, 2 = vyplň — zápis operaci spustí
`BLT_STATUS`	bit 0 připraven (vždy 1), bit 1 chyba (čtení maže)

```
        ; smaž textovou obrazovku mezerami
        LDA #<0x8000
        STA BLT_DST_LO
        LDA #>0x8000
        STA BLT_DST_HI
        LDA #<1000
        STA BLT_LEN_LO
        LDA #>1000
        STA BLT_LEN_HI
        LDA #' '
        STA BLT_FILL
        LDA #2              ; FILL
        STA BLT_CTRL
```

Celý přenos proběhne uvnitř jedné instrukce

Na rozdíl od úložiště blitter **nemá čekací okno**. Celý blok se přesune uvnitř zápisu do `BLT_CTRL` a nestojí to ani jeden T-stav navíc. Není tedy co dotazovat — bit připravenosti je konstantní jednička.

Kopírování se chová jako memmove

Směr kopírování si jednotka vybere podle adres: sestupně, když je cíl nad zdrojem, jinak vzestupně. **Překrývající se** kopie je proto vždycky správně — a to je přesně, co potřebuje hardwarový scroll, který kopíruje obrazovku sám na sebe posunutou o řádek. Žádný příznak směru se nenastavuje.

Vyplňování jde vždy vzestupně.

Past

Okno přenosu musí celé zůstat pod blokem periferií. Když `dst + len` (a u kopírování i `src + len`) dosáhne 0xFF00, jednotka nastaví chybový bit a **nepřenese nic**.

Není to buzerace: tahle jediná podmínka drží DMA mimo periferie, boot ROM i vektory a zároveň znemožňuje přetečení za konec paměti.

Poznámka

Každý bajt, který blitter zapíše, jde do žurnálu stejnou cestou jako zápis z procesoru. Návrat v čase tedy přehraje i blit — jen jeden velký blit umí zkrátit dosah převíjení, protože spotřebuje hodně položek žurnálu.

KAPITOLA 22

Zvuk

Tři nezávislé hlasy, každý s vlastní frekvencí, průběhem a obálkou. Zvukové zařízení je **zapiš a zapomeň** — procesor mu nastaví registry a dál se o nic nestará; generování obálky i míchání dělá hostitel a zpátky do procesoru z toho nejde nic.

Adresa	Registr
`0xFF20`	`AUD_CTRL`
`0xFF21`	`AUD_FREQ_LO`
`0xFF22`	`AUD_FREQ_HI`
`0xFF23`	`AUD_WAVE`
`0xFF24`	`AUD_AD`
`0xFF25`	`AUD_SR`

Rozvržení kanálů

Tři kanály po pěti registrech vyplní slot zařízení přesně. Kanál **n** začíná na `0xFF21 + n*5`:

Kanál	Rozsah	Registry
0	`0xFF21–0xFF25`	`FREQ_LO FREQ_HI WAVE AD SR`
1	`0xFF26–0xFF2A`	`FREQ_LO FREQ_HI WAVE AD SR`
2	`0xFF2B–0xFF2F`	`FREQ_LO FREQ_HI WAVE AD SR`

Obrázek 13. Tři kanály zvukového zařízení a jejich registry.

Posun	Registr	Význam
+0 / +1	`AUD_FREQ_LO` / `HI`	frekvence, 16 bitů, dolní bajt první
+2	`AUD_WAVE`	průběh
+3	`AUD_AD`	horní půlbajt náběh, dolní pokles
+4	`AUD_SR`	horní půlbajt úroveň držení, dolní dozvuk

> **Poznámka**

Odstup pěti bajtů je zvolený tak, aby se ke kterémukoli kanálu dalo dostat jedinou indexovanou instrukcí:

```
        LDX #5              ; 0, 5 nebo 10 = kanál 0, 1, 2
        STA AUD_FREQ_LO,X
```

Frekvence je v hertzích

Dvojice registrů **je** výška tónu: `f = {AUD_FREQ_HI, AUD_FREQ_LO}` hertzů, s rozlišením 1 Hz. Nula znamená ticho.

```
        LDA #<440
        STA AUD_FREQ_LO
        LDA #>440
        STA AUD_FREQ_HI     ; komorní a
```

Past

Ve starší verzi zvukového zařízení byl jediný kanál a frekvence se počítala ze vzorce `100 + (FREQ_LO/255) × 1900`, přičemž horní bajt byl mrtvý. Program napsaný proti staré křivce se pořád přeloží a poběží — jen bude hrát jinak: `AUD_FREQ_LO = 46` dřív znamenalo asi 443 Hz, dnes znamená 46 Hz.

Převod je proto vždycky psát frekvenci jako šestnáctibitové číslo.

Průběhy

Hodnota `AUD_WAVE`	Průběh
0	sinus
1	obdélník
2	trojúhelník
3	pila
4	šum
5–7	rezervováno — hrají ticho

Šum nahradí oscilátor posuvným registrem taktovaným frekvencí kanálu, takže `AUD_FREQ` u něj vybírá **barvu** šumu, ne výšku: nízké hodnoty duní, vysoké syčí.

Brána a její hrana

`AUD_CTRL` je bitová mapa bran — bit **n** otevírá kanál **n** (bity 3–7 čtou nulu). Brána je **hranová**:

Přechod	Co se stane
0 ⟶ 1	začne náběh — tón se rozezní
1 ⟶ 0	začne dozvuk — tón dozní

> **Past**
>
> Zápis bitu, který už nastavený je, tón **nespustí znovu**. Přehrávač, který chce zopakovat tutéž notu, musí bit nejdřív shodit a pak zvednout.
>
> Jeden zápis do `AUD_CTRL` naopak umí spustit nebo zastavit všechny tři hlasy na stejné hraně — což je přesně to, co se hodí u akordu.

Obálka

Každý kanál má vlastní ADSR obálku, která se čte při každé hraně brány.

Časové půlbajty (náběh, pokles, dozvuk) se převádějí takhle: hodnota 0 znamená okamžitě, jinak

$$t = 8\ \text{ms} \times 2^{(n-1)/2}$$

Tedy 8 ms pro 1, zdvojnásobení každé dva kroky, 1024 ms pro 15.

Půlbajt držení je úroveň `s/15`, na které tón zůstane, dokud se brána nezavře. Zároveň je to jediný způsob, jak nastavit **hlasitost** kanálu — žádný registr hlasitosti neexistuje. Držení na nule dělá perkusivní zvuk, který dozní sám.

```
        ; cvaknutí: hned náběh, rychlý pokles, bez držení
        LDA #0x03           ; náběh 0, pokles 3
        STA AUD_AD
        LDA #0x02           ; držení 0, dozvuk 2
        STA AUD_SR
```

Výchozí stav

Po resetu má každý kanál bránu zavřenou, frekvenci 440 Hz, sinus, `AUD_AD` = 0x00 a `AUD_SR` = 0xF0 — tedy okamžitý náběh, plné držení, okamžitý dozvuk. Obálka je tím pádem prostý vypínač, dokud ji program nezformuje.

Tři kanály se míchají se stejnou vahou a rezervou pro všechny tři naplno.

> **Poznámka**
>
> Registry jsou obyčejný stav stroje, takže jsou v žurnálu — návrat v čase přehraje zvuk přesně tak, jak zněl. Obálka a míchání ale žijí u hostitele a do procesoru se z nich nevrací nic, takže se program nemá jak zeptat, „jestli ještě hraje“.

KAPITOLA 23

Úložiště, boot sektor a SAP-DOS

Úložiště je jediné zařízení, se kterým se nedá mluvit okamžitě. Přenos má **trvání** — a to je ta nejdůležitější věc, kterou o něm program musí vědět.

Adresa	Registr
`0xFF60`	`DSK_CMD`
`0xFF61`	`DSK_STATUS`
`0xFF62`	`DSK_BLK`
`0xFF63`	`DSK_ADDR_LO`
`0xFF64`	`DSK_ADDR_HI`
`0xFF65`	`SER_DATA`
`0xFF66`	`SER_STATUS`
`0xFF67`	`SER_CTRL`
`0xFF68`	`DSK_DISK`

Médium

Médium je **karta až 256 disků**, ne jeden velký disk. Každý disk má 256 bloků po 256 bajtech, tedy 64 KB.

Registr `DSK_DISK` vybírá, který disk je „v mechanice“ pro příští příkaz. Čte se z něj naposledy zapsaná hodnota a při resetu se vrací na nulu — takže program, který se ho nikdy nedotkne, žije celý život na disku 0.

Kolik slotů je skutečně obsazených, je vlastnost vložené karty; emulátor jich nabízí 16. Příkaz mířící na slot, který na kartě není, je chyba.

> **Poznámka**
>
> Zařízení nezná formát. Jestli je na disku adresář, boot sektor nebo nic, je věcí softwaru — a každý slot může mít vlastní uspořádání.

Příkazy

Postup je vždy stejný: nastavit disk, blok a adresu v paměti, pak zapsat příkaz.

Registr	Význam
`DSK_DISK`	číslo slotu na kartě
`DSK_BLK`	číslo bloku na disku
`DSK_ADDR_LO` / `HI`	adresa v RAM
`DSK_CMD`	1 = čti blok do RAM, 2 = zapiš blok z RAM
`DSK_STATUS`	bit 0 připraven, bit 1 chyba (čtení maže)

Přenos trvá

Platný zápis do `DSK_CMD` si zapamatuje **tehdejší** hodnoty ostatních registrů a spustí čekací okno dlouhé `max(2, CLK_TPMS × 2)` T-stavů — přibližně 2 ms reálného času.

> **Past**
>
> Nikdy nepočítej takty, vždycky se ptej. Skutečné SD karty potřebují na blok od stovek mikrosekund po jednotky milisekund a to číslo není nijak zaručené.
>
> Data z čtení **nejsou v paměti**, dokud se `DSK_STATUS` bit 0 nevrátí na jedničku.

Přepisování `DSK_DISK`, `DSK_BLK` nebo `DSK_ADDR` během čekání je dovolené a ovlivní až **příští** příkaz.

Dotazování jedním odečtem

Protože čtení `DSK_STATUS` maže chybový bit, musí se stav přečíst jednou a oba bity testovat z téže hodnoty:

```
        STA DSK_CMD
.cekej: LDA DSK_STATUS
        STA tmp             ; jeden odečet, oba bity odtud
        AND #0x01
        JZ  .cekej          ; ještě pracuje
        LDA tmp
        AND #0x02
        JNZ chyba           ; příkaz byl odmítnut
```

Chyby vznikají při zadání příkazu

Chyba se nastaví **okamžitě** a přenos vůbec nezačne. Nastane ve čtyřech případech:

1. neplatná hodnota příkazu,
2. adresa v RAM nad 0xFE00,
3. slot `DSK_DISK` mimo vložené médium,

4. zápis do `DSK_CMD` během probíhajícího přenosu (rozpracovaný přenos doběhne).

Horní mez adresy je důvod, proč se DMA nikdy nedostane k periferiím, systémové RAM ani vektorům.

> **Poznámka**
>
> Zápis vzorkuje paměť v **neurčeném okamžiku** uvnitř čekacího okna — emulátor na jeho konci, hardware při zadání příkazu. Program, který si zdrojový buffer během čekání změní, dostane obsah média závislý na implementaci. Prostě se to nedělá.

Návrat v čase a médium

Bajty, které DMA zapíše do RAM, jsou v žurnálu po jednom, takže se převíjení přehraje přesně a bez opětovného čtení disku.

Médium samo se ale nepřevíjí. Krok zpět přes zápis na disk ho neodepíše — stejně jako u skutečného hardwaru. A když se program vrátí před čtení a provede ho znovu, přečte médium v tom stavu, v jakém je **teď**.

Boot z disku

Normálně hostitel položí přeložený program rovnou do RAM a vektor RESET ukáže na něj. **Bootovací režim** místo toho nechá stroj nastartovat jako skutečný počítač: do 0xFF80 se nainstaluje malá boot ROM, vektor RESET ukazuje na ni a RAM je prázdná.

Boot ROM je **skutečný strojový kód SAP-3**, takže se celý start dá odkrokovat v ladicím nástroji.

Co dělá:

1. DMA přenese blok 0 na odkládací stránku 0xFE00,
2. zkontroluje dvoubajtovou signaturu `"S3"` — když nesedí, vypíše `NO OS` a zastaví se,
3. přečte hlavičku,
4. DMA přenese bloky 1..N do RAM na uvedenou adresu,
5. skočí na vstupní bod přes `JMP (zp)`.

> **Poznámka**
>
> ROM bydlí v okně 0xFF80–0xFFF9, kam DMA nedosáhne. Zavedení, které sama spustí, ji tedy nemůže přepsat. Pracovní bajty si drží v horní části nulté stránky, takže užitečné zatížení se nahrává od 0x0200 výš.

Boot sektor

Blok 0 bootovacího disku, dolním bajtem napřed:

Posun	Délka	Význam
0x00	2	signatura `"S3"`
0x02	2	adresa nahrání, zarovnaná na blok
0x04	1	počet bloků užitečného zatížení
0x05	2	vstupní bod
0x07	1	datový disk k připojení; 0 = nechat zavedený
0x08	16	jméno programu, ASCII doplněné nulami

Užitečné zatížení leží v blocích 1 až N.

Poslední dvě položky boot ROM **ignoruje** — čte je až zavaděč druhého stupně, tedy nabídka SAP-DOS. Ta podle jména sestaví seznam programů a podle datového disku připojí správný slot, než skočí.

> **Poznámka**
>
> Bootovací disk a datový disk Tiny BASICu se liší jen formátem bloku 0. Zařízení mezi nimi nerozlišuje; rozliší je až software podle signatury.

> **Past**
>
> Reset ani nahrání programu se média **nedotknou** — resetují se jen registry a `DSK_DISK` se vrátí na nulu. Je to jako by se nic nevysunulo. Program, který spoléhá na to, že po resetu bude disk „čistý“, se plete.

KAPITOLA 24

Od emulátoru ke křemíku

Tahle kapitola je přehledová. Neslouží k tomu, aby si podle ní někdo postavil desku — na to jsou návrhové dokumenty a schémata. Odpovídá na jinou otázku: **co znamená, že tenhle procesor existuje ve třech implementacích, a proč se nerozešly.**

Tři stroje, jedna specifikace

Implementace	Co to je
emulátor	TypeScript v prohlížeči na sap-3.com — krokování po T-stavech, návrat v čase, viditelné řídicí signály
AGATA-1	jádro ve Verilogu na FPGA Sipeed Tang Nano 20K; srdce konzole MAXIK
port pro ESP32	tentýž procesor v C++ na mikrokontroléru s malým displejem

Kdyby to byly tři nezávislé implementace téhož popisu, rozešly by se během týdnů. Nejsou — a stojí to na třech mechanismech.

Mikrokód má jediný zdroj

Mikrokód není v HDL napsaný. Je **vygenerovaný** z téhož zdroje, ze kterého ho bere emulátor, a nahraný do ROM. Kdo chce mikrokód změnit, mění ho na jednom místě; hardware ho dostane při příštím sestavení.

S tím souvisí jedna nezvyklá vlastnost taxonomie řídicích signálů z kapitoly 4: pořadí signálů uvnitř skupin **je součástí kódování mikroslova**. Signál se smí přidat na konec, ale ne přeskupit — jinak by se změnila ROM, aniž by se změnil mikrokód.

ALU se ověřuje vyčerpávajícím výčtem

Osmibitová ALU má konečně mnoho vstupů, takže se dá otestovat **celá**. Ze zdroje se vygeneruje sada vektorů — všechny kombinace operandů pro všechny operace — a simulace HDL je musí trefit do jednoho bitu, včetně příznaků.

Programy se přehrávají proti referenčním stopám

Emulátor umí vyrobit stopu: pro každý T-stav stav registrů, sběrnic a signálů. Simulace hardwaru pak spustí tentýž program a musí projít stejnou posloupností. Ne stejným výsledkem — stejnou posloupností.

> **Poznámka**

> Tohle je důvod, proč se v celé knize dá tvrdit, že počty T-stavů platí i na desce. Rovnost „prohlížeč se chová jako deska“ není příslib v dokumentaci; je to podmínka, kterou musí splnit sestavení.

Co za to determinismus stojí

Cena je konkrétní a byla vidět už v kapitole o výkonu: klasické zrychlovací triky moderních procesorů — cache, predikce skoků, přeuspořádání instrukcí — jsou **z principu** mimo hru. Všechny totiž dělají dobu běhu závislou na historii, a tím by rozbily porovnávání se stopou, krokování i návrat v čase.

Povolené je jen deterministické zrychlení: pevně kratší počty T-stavů, širší načítání instrukcí, případně pipeline s pevným rozvrhem.

Co se za to koupí: program lze krokovat po jednotlivých hradlech, vracet v čase, a naměřený počet taktů z prohlížeče platí na desce. U stroje, který má především učit, jak počítač funguje, je to výhodný obchod.

AGATA-1 na Tang Nano 20K

Jádro obsadí zhruba 700 hradlových buněk, celý systém na čipu asi 8 % dostupné logiky. Úzkým hrdlem není logika, ale bloková paměť: z 46 bloků je obsazených 45.

Zdroj	Stav
bloková paměť	45 ze 46 bloků — volný je jeden
logika	zhruba 8 % obsazeno
maximální frekvence	asi 32 MHz po umístění a propojení

Kritická cesta vede z paměti přes multiplexor sběrnice a ALU až do řetězu příznaků, a to všechno v jednom taktu.

> **Poznámka**
>
> Ten jediný volný blok paměti je nejtvrdší omezení celého návrhu. Cokoli, co by chtělo novou paměť — cache, širší ROM, vyrovnávací buffery — nemá na téhle desce kde bydlet. Naopak logiky je skoro zadarmo.

Turbo

V turbo režimu běží jádro na 25,2 MHz s jedním T-stavem na takt. Při průměrné instrukci za 7 až 9 T-stavů to dělá zhruba 3 miliony instrukcí za sekundu.

Právě 25 200 je horní mez, kterou připouští specifikace pro registr `CLK_TPMS` — odpovídá jednomu T-stavu na takt pixelových hodin. Šestnáct bitů toho registru přitom nechává prostor až někam k 65 MHz pro budoucí implementace.

Konzole MAXIK

Deska kolem jádra přidává to, co z procesoru dělá počítač: výstup HDMI, SD kartu s multidiskovou kartou, dva porty pro ovladače NES a sériovou linku.

Samostatnou kapitolou je čelní panel — deska plná svítivých diod a sedmisegmentovek, která v reálném čase ukazuje obsah sběrnic, registrů a příznaků. Barevný jazyk je jednotný: jantarová jsou adresy, zelená data, červená příznaky.

Panel není ozdoba. Je to fyzické provedení téhož schématu, které ukazuje emulátor — a jediný způsob, jak vidět, co procesor dělá, aniž by se člověk díval do počítače.

Co z toho plyne pro program

Prakticky nic — a to je pointa. Program napsaný podle této knihy poběží na všech třech strojích stejně. Jediné, na co si má dát pozor, je **rychlost hodin**, protože ta se liší řádově:

Stroj	Typická rychlost
emulátor, výukový režim	zlomky Hz až kilohertze
emulátor, běžný provoz	statisíce Hz až 2 MHz
AGATA-1, turbo	25,2 MHz

Program, který si rychlost přečte z `CLK_TPMS` nebo se paceuje podle `CLK_MS`, běží všude stejně rychle. Program, který má zabudovanou konstantu, poběží na desce dvanáctkrát rychleji, než čekal.

> **Past**
>
> Nejčastější chyba při přenosu programu z prohlížeče na desku není chyba v logice, ale ve zpožďovací smyčce spočítané na počet průchodů. Na 25,2 MHz proběhne prakticky okamžitě.

Přílohy

Opkódová matice

Celý osmibitový opkódový prostor v jedné tabulce. Řádek je horní půlbajt, sloupec dolní; políčko `4` × `1` je tedy opkód `0x41`. Tečka znamená volný slot — takový bajt procesor odmítne provést a spadne s hlášením o ilegálním opkódu.

	_0	_1	_2	_3	_4	_5	_6	_7	_8	_9	_A	_B	_C	_D	_E	_F
0_	·	NOP	HLT	OUT	CLC	SEC	EI	DI	INC	DEC	SHL	SHR	ROL	ROR	NOT	RND
1_	INX	DEX	INY	DEY	·	·	·	·	TAX	TXA	TAY	TYA	TAB	TBA	TXYS	TSXY
2_	PUSH	POP	PHX	PLX	PHY	PLY	PHF	PLF	MUL	DIV	PHB	PLB	·	·	·	·
3_	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·
4_	LDA #n	LDA zp	LDA abs	LDA abs,X	LDA abs,Y	LDA (zp)	LDA zp,X	LDA (zp),Y	LDX #n	LDX zp	LDX abs	·	LDX abs,Y	·	·	·
5_	LDY #n	LDY zp	LDY abs	LDY abs,X	·	·	LDY zp,X	·	CPX #n	CPX zp	CPX abs	·	·	·	·	·
6_	·	STA zp	STA abs	STA abs,X	STA abs,Y	STA (zp)	STA zp,X	STA (zp),Y	·	STX zp	STX abs	·	·	·	·	·
7_	·	STY zp	STY abs	·	·	·	·	·	CPY #n	CPY zp	CPY abs	·	·	·	·	·
8_	ADD #n	ADD zp	ADD abs	ADD abs,X	ADD abs,Y	ADD (zp)	ADD zp,X	ADD (zp),Y	ADC #n	ADC zp	ADC abs	ADC abs,X	ADC abs,Y	ADC (zp)	ADC zp,X	ADC (zp),Y
9_	SUB #n	SUB zp	SUB abs	SUB abs,X	SUB abs,Y	SUB (zp)	SUB zp,X	SUB (zp),Y	SBC #n	SBC zp	SBC abs	SBC abs,X	SBC abs,Y	SBC (zp)	SBC zp,X	SBC (zp),Y
A_	AND #n	AND zp	AND abs	AND abs,X	AND abs,Y	AND (zp)	AND zp,X	AND (zp),Y	OR #n	OR zp	OR abs	OR abs,X	OR abs,Y	OR (zp)	OR zp,X	OR (zp),Y
B_	XOR #n	XOR zp	XOR abs	XOR abs,X	XOR abs,Y	XOR (zp)	XOR zp,X	XOR (zp),Y	CMP #n	CMP zp	CMP abs	CMP abs,X	CMP abs,Y	CMP (zp)	CMP zp,X	CMP (zp),Y
C_	JNZ abs	JZ abs	JNC abs	JC abs	JP abs	JN abs	JNV abs	JV abs	JMP abs	JMP (zp)	CALL abs	RET	RTI	BRK	CALL (zp)	·
D_	INC zp	INC abs	DEC zp	DEC abs	INW zp	INW abs	DEW zp	DEW abs	SHL zp	SHL abs	SHR zp	SHR abs	ROL zp	ROL abs	ROR zp	ROR abs
E_	JGT abs	JLE abs	JGES abs	JLTS abs	BIT #n	BIT zp	BIT abs	·	ADW zp	SBW zp	LXY #n	SXY zp	·	·	·	·
F_	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·

Z 256 možných bajtů je 179 legálních instrukcí, 2 jsou vyhrazené pseudoinstrukce, které vkládá hardware (`0xFD RESET`, `0xFE IRQ`), a 75 slotů zbývá volných.

> **Poznámka**
>
> Struktura matice není náhodná. Vodorovné pruhy odpovídají skupinám `gg`: horní čtvrtina (řádky 0–3) jsou bezoperandové instrukce, druhá (4–7) přesuny mezi registry a pamětí, třetí (8–B) aritmetika a logika nad **A**, poslední (C–F) řízení toku a operace typu čti-uprav-zapiš. Uvnitř skupin 01 a 10 je sloupec dán adresním režimem — proto v nich sousedí tytéž mnemoniky po osmicích.

PŘÍLOHA B

Instrukce po skupinách

Úplný seznam všech 179 legálních opkódů, seřazený podle hodnoty bajtu a rozdělený podle skupiny `gg`. Sloupec „Příznaky“ vypisuje pouze příznaky, které daný tvar **definuje**; co tam není, zůstane beze změny.

Skupina 00 — bezoperandové instrukce

Jeden bajt, žádný operand. Rozsah 0x00–0x3F.

Opkód	Instrukce	Zápis	Bajtů	T	Příznaky
`0x01`	`NOP`	—	1	3	*žádné*
`0x02`	`HLT`	—	1	3	*žádné*
`0x03`	`OUT`	—	1	3	*žádné*
`0x04`	`CLC`	—	1	3	`C`
`0x05`	`SEC`	—	1	3	`C`
`0x06`	`EI`	—	1	3	`I`
`0x07`	`DI`	—	1	3	`I`
`0x08`	`INC`	—	1	3	`Z N`
`0x09`	`DEC`	—	1	3	`Z N`
`0x0A`	`SHL`	—	1	3	`C Z N`
`0x0B`	`SHR`	—	1	3	`C Z N`
`0x0C`	`ROL`	—	1	3	`C Z N`
`0x0D`	`ROR`	—	1	3	`C Z N`
`0x0E`	`NOT`	—	1	3	`Z N`
`0x0F`	`RND`	—	1	3	*žádné*
`0x10`	`INX`	—	1	4	`Z N`
`0x11`	`DEX`	—	1	4	`Z N`
`0x12`	`INY`	—	1	4	`Z N`
`0x13`	`DEY`	—	1	4	`Z N`
`0x18`	`TAX`	—	1	3	`Z N`
`0x19`	`TXA`	—	1	3	`Z N`

Opkód	Instrukce	Zápis	Bajtů	T	Příznaky
`0x1A`	`TAY`	—	1	3	`Z N`
`0x1B`	`TYA`	—	1	3	`Z N`
`0x1C`	`TAB`	—	1	3	`Z N`
`0x1D`	`TBA`	—	1	3	`Z N`
`0x1E`	`TXYS`	—	1	4	*žádné*
`0x1F`	`TSXY`	—	1	4	*žádné*
`0x20`	`PUSH`	—	1	4	*žádné*
`0x21`	`POP`	—	1	4	`Z N`
`0x22`	`PHX`	—	1	4	*žádné*
`0x23`	`PLX`	—	1	4	`Z N`
`0x24`	`PHY`	—	1	4	*žádné*
`0x25`	`PLY`	—	1	4	`Z N`
`0x26`	`PHF`	—	1	4	*žádné*
`0x27`	`PLF`	—	1	4	`C Z N V I`
`0x28`	`MUL`	—	1	4	`C Z N`
`0x29`	`DIV`	—	1	4	`C Z N`
`0x2A`	`PHB`	—	1	4	*žádné*
`0x2B`	`PLB`	—	1	4	`Z N`

Skupina 01 — přesuny mezi registry a pamětí

Rozsah 0x40–0x7F. Opkód je `01 | ooo | mmm`, takže tytéž mnemoniky leží v souvislých osmicích.

Opkód	Instrukce	Zápis	Bajtů	T	Příznaky
`0x40`	`LDA`	`#n`	2	5	`Z N`
`0x41`	`LDA`	`addr8`	2	6	`Z N`
`0x42`	`LDA`	`addr16`	3	9	`Z N`
`0x43`	`LDA`	`addr16,X`	3	11	`Z N`
`0x44`	`LDA`	`addr16,Y`	3	11	`Z N`
`0x45`	`LDA`	`(addr8)`	2	10	`Z N`
`0x46`	`LDA`	`addr8,X`	2	7	`Z N`
`0x47`	`LDA`	`(addr8),Y`	2	13	`Z N`

Opkód	Instrukce	Zápis	Bajtů	T	Příznaky
0x48	LDX	#n	2	5	Z N
0x49	LDX	addr8	2	6	Z N
0x4A	LDX	addr16	3	9	Z N
0x4C	LDX	addr16,Y	3	11	Z N
0x50	LDY	#n	2	5	Z N
0x51	LDY	addr8	2	6	Z N
0x52	LDY	addr16	3	9	Z N
0x53	LDY	addr16,X	3	11	Z N
0x56	LDY	addr8,X	2	7	Z N
0x58	CPX	#n	2	7	C Z N
0x59	CPX	addr8	2	8	C Z N
0x5A	CPX	addr16	3	11	C Z N
0x61	STA	addr8	2	6	*žádné*
0x62	STA	addr16	3	9	*žádné*
0x63	STA	addr16,X	3	11	*žádné*
0x64	STA	addr16,Y	3	11	*žádné*
0x65	STA	(addr8)	2	10	*žádné*
0x66	STA	addr8,X	2	7	*žádné*
0x67	STA	(addr8),Y	2	13	*žádné*
0x69	STX	addr8	2	6	*žádné*
0x6A	STX	addr16	3	9	*žádné*
0x71	STY	addr8	2	6	*žádné*
0x72	STY	addr16	3	9	*žádné*
0x78	CPY	#n	2	7	C Z N
0x79	CPY	addr8	2	8	C Z N
0x7A	CPY	addr16	3	11	C Z N

Skupina 10 — aritmetika a logika nad A

Rozsah 0x80–0xBF, kódování `10 | ooo | mmm`.

Opkód	Instrukce	Zápis	Bajtů	T	Příznaky
0x80	ADD	#n	2	6	C Z N V

Opkód	Instrukce	Zápis	Bajtů	T	Příznaky
0x81	ADD	addr8	2	7	C Z N V
0x82	ADD	addr16	3	10	C Z N V
0x83	ADD	addr16,X	3	12	C Z N V
0x84	ADD	addr16,Y	3	12	C Z N V
0x85	ADD	(addr8)	2	11	C Z N V
0x86	ADD	addr8,X	2	8	C Z N V
0x87	ADD	(addr8),Y	2	14	C Z N V
0x88	ADC	#n	2	6	C Z N V
0x89	ADC	addr8	2	7	C Z N V
0x8A	ADC	addr16	3	10	C Z N V
0x8B	ADC	addr16,X	3	12	C Z N V
0x8C	ADC	addr16,Y	3	12	C Z N V
0x8D	ADC	(addr8)	2	11	C Z N V
0x8E	ADC	addr8,X	2	8	C Z N V
0x8F	ADC	(addr8),Y	2	14	C Z N V
0x90	SUB	#n	2	6	C Z N V
0x91	SUB	addr8	2	7	C Z N V
0x92	SUB	addr16	3	10	C Z N V
0x93	SUB	addr16,X	3	12	C Z N V
0x94	SUB	addr16,Y	3	12	C Z N V
0x95	SUB	(addr8)	2	11	C Z N V
0x96	SUB	addr8,X	2	8	C Z N V
0x97	SUB	(addr8),Y	2	14	C Z N V
0x98	SBC	#n	2	6	C Z N V
0x99	SBC	addr8	2	7	C Z N V
0x9A	SBC	addr16	3	10	C Z N V
0x9B	SBC	addr16,X	3	12	C Z N V
0x9C	SBC	addr16,Y	3	12	C Z N V
0x9D	SBC	(addr8)	2	11	C Z N V
0x9E	SBC	addr8,X	2	8	C Z N V
0x9F	SBC	(addr8),Y	2	14	C Z N V

Opkód	Instrukce	Zápis	Bajtů	T	Příznaky
0xA0	AND	#n	2	6	Z N
0xA1	AND	addr8	2	7	Z N
0xA2	AND	addr16	3	10	Z N
0xA3	AND	addr16,X	3	12	Z N
0xA4	AND	addr16,Y	3	12	Z N
0xA5	AND	(addr8)	2	11	Z N
0xA6	AND	addr8,X	2	8	Z N
0xA7	AND	(addr8),Y	2	14	Z N
0xA8	OR	#n	2	6	Z N
0xA9	OR	addr8	2	7	Z N
0xAA	OR	addr16	3	10	Z N
0xAB	OR	addr16,X	3	12	Z N
0xAC	OR	addr16,Y	3	12	Z N
0xAD	OR	(addr8)	2	11	Z N
0xAE	OR	addr8,X	2	8	Z N
0xAF	OR	(addr8),Y	2	14	Z N
0xB0	XOR	#n	2	6	Z N
0xB1	XOR	addr8	2	7	Z N
0xB2	XOR	addr16	3	10	Z N
0xB3	XOR	addr16,X	3	12	Z N
0xB4	XOR	addr16,Y	3	12	Z N
0xB5	XOR	(addr8)	2	11	Z N
0xB6	XOR	addr8,X	2	8	Z N
0xB7	XOR	(addr8),Y	2	14	Z N
0xB8	CMP	#n	2	6	C Z N
0xB9	CMP	addr8	2	7	C Z N
0xBA	CMP	addr16	3	10	C Z N
0xBB	CMP	addr16,X	3	12	C Z N
0xBC	CMP	addr16,Y	3	12	C Z N
0xBD	CMP	(addr8)	2	11	C Z N
0xBE	CMP	addr8,X	2	8	C Z N

Opkód	Instrukce	Zápis	Bajtů	T	Příznaky
`0xBF`	`CMP`	`(addr8),Y`	2	14	`C Z N`

Skupina 11 — řízení toku a čti-uprav-zapiš

Rozsah 0xC0–0xFF. Jediná skupina, která systematická není — její instrukce buď operand nemají, nebo mají netypický.

Opkód	Instrukce	Zápis	Bajtů	T	Příznaky
`0xC0`	`JNZ`	`addr16`	3	7	*žádné*
`0xC1`	`JZ`	`addr16`	3	7	*žádné*
`0xC2`	`JNC`	`addr16`	3	7	*žádné*
`0xC3`	`JC`	`addr16`	3	7	*žádné*
`0xC4`	`JP`	`addr16`	3	7	*žádné*
`0xC5`	`JN`	`addr16`	3	7	*žádné*
`0xC6`	`JNV`	`addr16`	3	7	*žádné*
`0xC7`	`JV`	`addr16`	3	7	*žádné*
`0xC8`	`JMP`	`addr16`	3	7	*žádné*
`0xC9`	`JMP`	`(addr8)`	2	9	*žádné*
`0xCA`	`CALL`	`addr16`	3	11	*žádné*
`0xCB`	`RET`	—	1	6	*žádné*
`0xCC`	`RTI`	—	1	8	`C Z N V I`
`0xCD`	`BRK`	—	1	12	`I`
`0xCE`	`CALL`	`(addr8)`	2	13	*žádné*
`0xD0`	`INC`	`addr8`	2	7	`Z N`
`0xD1`	`INC`	`addr16`	3	10	`Z N`
`0xD2`	`DEC`	`addr8`	2	7	`Z N`
`0xD3`	`DEC`	`addr16`	3	10	`Z N`
`0xD4`	`INW`	`addr8`	2	10	*žádné*
`0xD5`	`INW`	`addr16`	3	13	*žádné*
`0xD6`	`DEW`	`addr8`	2	10	*žádné*
`0xD7`	`DEW`	`addr16`	3	13	*žádné*
`0xD8`	`SHL`	`addr8`	2	7	`C Z N`
`0xD9`	`SHL`	`addr16`	3	10	`C Z N`

Opkód	Instrukce	Zápis	Bajtů	T	Příznaky
`0xDA`	`SHR`	`addr8`	2	7	`C Z N`
`0xDB`	`SHR`	`addr16`	3	10	`C Z N`
`0xDC`	`ROL`	`addr8`	2	7	`C Z N`
`0xDD`	`ROL`	`addr16`	3	10	`C Z N`
`0xDE`	`ROR`	`addr8`	2	7	`C Z N`
`0xDF`	`ROR`	`addr16`	3	10	`C Z N`
`0xE0`	`JGT`	`addr16`	3	7	*žádné*
`0xE1`	`JLE`	`addr16`	3	7	*žádné*
`0xE2`	`JGES`	`addr16`	3	7	*žádné*
`0xE3`	`JLTS`	`addr16`	3	7	*žádné*
`0xE4`	`BIT`	`#n`	2	6	`Z N`
`0xE5`	`BIT`	`addr8`	2	7	`Z N`
`0xE6`	`BIT`	`addr16`	3	10	`Z N`
`0xE8`	`ADW`	`addr8,#n`	3	13	*žádné*
`0xE9`	`SBW`	`addr8,#n`	3	13	*žádné*
`0xEA`	`LXY`	`#nn`	3	8	*žádné*
`0xEB`	`SXY`	`addr8`	2	8	*žádné*

PŘÍLOHA C

Mikrokódový výpis

Kompletní obsah mikrokódové paměti: 181 mikroprogramů, seřazených podle opkódu. Jeden pár hranatých závorek je jeden T-stav, signály uvnitř jsou v tom T-stavu aktivní. Otazník za závorkou označuje krok, který se přeskočí, když neplatí podmínka.

První dva až šest kroků každého programu je fetch a vypadá u všech instrukcí stejně. Význam signálů je v příloze D.

> **Poznámka**
>
> Poslední dva záznamy (`0xFD` a `0xFE`) jsou pseudoinstrukce, které vkládá hardware při resetu a při přijetí přerušení. Nejde je načíst z paměti, a proto nemají fetch.

```
0x01  NOP  (3 T)
[PCM] [RO,II,CE] []
0x02  HLT  (3 T)
[PCM] [RO,II,CE] [HLT]
0x03  OUT  (3 T)
[PCM] [RO,II,CE] [AO,OI]
0x04  CLC  (3 T)
[PCM] [RO,II,CE] [CFC]
0x05  SEC  (3 T)
[PCM] [RO,II,CE] [CFS]
0x06  EI  (3 T)
[PCM] [RO,II,CE] [EIF]
0x07  DI  (3 T)
[PCM] [RO,II,CE] [DIF]
0x08  INC  (3 T)
[PCM] [RO,II,CE] [INC,EO,AI,FI]
0x09  DEC  (3 T)
[PCM] [RO,II,CE] [DEC,EO,AI,FI]
0x0A  SHL  (3 T)
[PCM] [RO,II,CE] [SHL,EO,AI,FI]
0x0B  SHR  (3 T)
[PCM] [RO,II,CE] [SHR,EO,AI,FI]
0x0C  ROL  (3 T)
[PCM] [RO,II,CE] [ROL,EO,AI,FI]
0x0D  ROR  (3 T)
[PCM] [RO,II,CE] [ROR,EO,AI,FI]
0x0E  NOT  (3 T)
[PCM] [RO,II,CE] [NOT,EO,AI,FI]
0x0F  RND  (3 T)
```

```
[PCM] [RO,II,CE] [RND]
```

0x10 INX (4 T)
```
[PCM] [RO,II,CE] [XO,TI] [ATM,INC,EO,XI,FI]
```

0x11 DEX (4 T)
```
[PCM] [RO,II,CE] [XO,TI] [ATM,DEC,EO,XI,FI]
```

0x12 INY (4 T)
```
[PCM] [RO,II,CE] [YO,TI] [ATM,INC,EO,YI,FI]
```

0x13 DEY (4 T)
```
[PCM] [RO,II,CE] [YO,TI] [ATM,DEC,EO,YI,FI]
```

0x18 TAX (3 T)
```
[PCM] [RO,II,CE] [AO,XI,FI]
```

0x19 TXA (3 T)
```
[PCM] [RO,II,CE] [XO,AI,FI]
```

0x1A TAY (3 T)
```
[PCM] [RO,II,CE] [AO,YI,FI]
```

0x1B TYA (3 T)
```
[PCM] [RO,II,CE] [YO,AI,FI]
```

0x1C TAB (3 T)
```
[PCM] [RO,II,CE] [AO,BI,FI]
```

0x1D TBA (3 T)
```
[PCM] [RO,II,CE] [BO,AI,FI]
```

0x1E TXYS (4 T)
```
[PCM] [RO,II,CE] [XO,SHI] [YO,SLI]
```

0x1F TSXY (4 T)
```
[PCM] [RO,II,CE] [SHO,XI] [SLO,YI]
```

0x20 PUSH (4 T)
```
[PCM] [RO,II,CE] [SD,SPM] [AO,RI]
```

0x21 POP (4 T)
```
[PCM] [RO,II,CE] [SPM] [RO,AI,FI,SE]
```

0x22 PHX (4 T)
```
[PCM] [RO,II,CE] [SD,SPM] [XO,RI]
```

0x23 PLX (4 T)
```
[PCM] [RO,II,CE] [SPM] [RO,XI,FI,SE]
```

0x24 PHY (4 T)
```
[PCM] [RO,II,CE] [SD,SPM] [YO,RI]
```

0x25 PLY (4 T)
```
[PCM] [RO,II,CE] [SPM] [RO,YI,FI,SE]
```

0x26 PHF (4 T)
```
[PCM] [RO,II,CE] [SD,SPM] [FLO,RI]
```

0x27 PLF (4 T)
```
[PCM] [RO,II,CE] [SPM] [RO,FLI,SE]
```

0x28 MUL (4 T)
```
[PCM] [RO,II,CE] [MUL,EO,AI,FI] [EXO,BI]
```

0x29 DIV (4 T)
```
[PCM] [RO,II,CE] [DIV,EO,AI,FI] [EXO,BI]
```

0x2A PHB (4 T)
```
[PCM] [RO,II,CE] [SD,SPM] [BO,RI]
```

0x2B PLB (4 T)
```
[PCM] [RO,II,CE] [SPM] [RO,BI,FI,SE]
```

0x40 LDA #n (5 T)
```
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [ORO,AI,FI]
```

0x41 LDA addr8 (6 T)

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

0x42 `LDA addr16` (9 T)
```
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [PCM] [RO,OHI,CE] [ORO,MI] [OHO,MIH] [RO,AI,FI]
```

0x43 `LDA addr16,X` (11 T)
```
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [PCM] [RO,OHI,CE] [XO,TI] [ATM,AOP,EO,MI,ACO]
[OHO,TI] [ATM,AHC,EO,MIH] [RO,AI,FI]
```

0x44 `LDA addr16,Y` (11 T)
```
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [PCM] [RO,OHI,CE] [YO,TI] [ATM,AOP,EO,MI,ACO]
[OHO,TI] [ATM,AHC,EO,MIH] [RO,AI,FI]
```

0x45 `LDA (addr8)` (10 T)
```
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [ORO,MI] [RO,TI] [MRI] [RO,MIH] [TO,MIL] [RO,AI,FI]
```

0x46 `LDA addr8,X` (7 T)
```
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [XO,TI] [ATM,AOP,EO,MI] [RO,AI,FI]
```

0x47 `LDA (addr8),Y` (13 T)
```
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [ORO,MI] [RO,ORI] [YO,TI] [MRI] [RO,OHI]
[ATM,AOP,EO,MI,ACO] [OHO,TI] [ATM,AHC,EO,MIH] [RO,AI,FI]
```

0x48 `LDX #n` (5 T)
```
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [ORO,XI,FI]
```

0x49 `LDX addr8` (6 T)
```
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [ORO,MI] [RO,XI,FI]
```

0x4A `LDX addr16` (9 T)
```
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [PCM] [RO,OHI,CE] [ORO,MI] [OHO,MIH] [RO,XI,FI]
```

0x4C `LDX addr16,Y` (11 T)
```
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [PCM] [RO,OHI,CE] [YO,TI] [ATM,AOP,EO,MI,ACO]
[OHO,TI] [ATM,AHC,EO,MIH] [RO,XI,FI]
```

0x50 `LDY #n` (5 T)
```
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [ORO,YI,FI]
```

0x51 `LDY addr8` (6 T)
```
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [ORO,MI] [RO,YI,FI]
```

0x52 `LDY addr16` (9 T)
```
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [PCM] [RO,OHI,CE] [ORO,MI] [OHO,MIH] [RO,YI,FI]
```

0x53 `LDY addr16,X` (11 T)
```
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [PCM] [RO,OHI,CE] [XO,TI] [ATM,AOP,EO,MI,ACO]
[OHO,TI] [ATM,AHC,EO,MIH] [RO,YI,FI]
```

0x56 `LDY addr8,X` (7 T)
```
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [XO,TI] [ATM,AOP,EO,MI] [RO,YI,FI]
```

0x58 `CPX #n` (7 T)
```
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [XO,TI] [ORO,BI] [ATM,CMP,EO,FI]
```

0x59 `CPX addr8` (8 T)
```
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [XO,TI] [ORO,MI] [RO,BI] [ATM,CMP,EO,FI]
```

0x5A `CPX addr16` (11 T)
```
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [PCM] [RO,OHI,CE] [XO,TI] [ORO,MI] [OHO,MIH] [RO,BI]
[ATM,CMP,EO,FI]
```

0x61 `STA addr8` (6 T)
```
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [ORO,MI] [AO,RI]
```

0x62 `STA addr16` (9 T)
```
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [PCM] [RO,OHI,CE] [ORO,MI] [OHO,MIH] [AO,RI]
```

0x63 `STA addr16,X` (11 T)
```
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [PCM] [RO,OHI,CE] [XO,TI] [ATM,AOP,EO,MI,ACO]
[OHO,TI] [ATM,AHC,EO,MIH] [AO,RI]
```

0x64 `STA addr16,Y` (11 T)
```
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [PCM] [RO,OHI,CE] [YO,TI] [ATM,AOP,EO,MI,ACO]
[OHO,TI] [ATM,AHC,EO,MIH] [AO,RI]
```

0x65 `STA (addr8)` (10 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [ORO,MI] [RO,TI] [MRI] [RO,MIH] [TO,MIL] [AO,RI]

0x66 `STA addr8,X` (7 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [XO,TI] [ATM,AOP,EO,MI] [AO,RI]

0x67 `STA (addr8),Y` (13 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [ORO,MI] [RO,ORI] [YO,TI] [MRI] [RO,OHI] [ATM,AOP,EO,MI,ACO] [OHO,TI] [ATM,AHC,EO,MIH] [AO,RI]

0x69 `STX addr8` (6 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [ORO,MI] [XO,RI]

0x6A `STX addr16` (9 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [PCM] [RO,OHI,CE] [ORO,MI] [OHO,MIH] [XO,RI]

0x71 `STY addr8` (6 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [ORO,MI] [YO,RI]

0x72 `STY addr16` (9 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [PCM] [RO,OHI,CE] [ORO,MI] [OHO,MIH] [YO,RI]

0x78 `CPY #n` (7 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [YO,TI] [ORO,BI] [ATM,CMP,EO,FI]

0x79 `CPY addr8` (8 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [YO,TI] [ORO,MI] [RO,BI] [ATM,CMP,EO,FI]

0x7A `CPY addr16` (11 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [PCM] [RO,OHI,CE] [YO,TI] [ORO,MI] [OHO,MIH] [RO,BI] [ATM,CMP,EO,FI]

0x80 `ADD #n` (6 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [ORO,BI] [EO,AI,FI]

0x81 `ADD addr8` (7 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [ORO,MI] [RO,BI] [EO,AI,FI]

0x82 `ADD addr16` (10 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [PCM] [RO,OHI,CE] [ORO,MI] [OHO,MIH] [RO,BI] [EO,AI,FI]

0x83 `ADD addr16,X` (12 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [PCM] [RO,OHI,CE] [XO,TI] [ATM,AOP,EO,MI,ACO] [OHO,TI] [ATM,AHC,EO,MIH] [RO,BI] [EO,AI,FI]

0x84 `ADD addr16,Y` (12 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [PCM] [RO,OHI,CE] [YO,TI] [ATM,AOP,EO,MI,ACO] [OHO,TI] [ATM,AHC,EO,MIH] [RO,BI] [EO,AI,FI]

0x85 `ADD (addr8)` (11 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [ORO,MI] [RO,TI] [MRI] [RO,MIH] [TO,MIL] [RO,BI] [EO,AI,FI]

0x86 `ADD addr8,X` (8 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [XO,TI] [ATM,AOP,EO,MI] [RO,BI] [EO,AI,FI]

0x87 `ADD (addr8),Y` (14 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [ORO,MI] [RO,ORI] [YO,TI] [MRI] [RO,OHI] [ATM,AOP,EO,MI,ACO] [OHO,TI] [ATM,AHC,EO,MIH] [RO,BI] [EO,AI,FI]

0x88 `ADC #n` (6 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [ORO,BI] [CI,EO,AI,FI]

0x89 `ADC addr8` (7 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [ORO,MI] [RO,BI] [CI,EO,AI,FI]

0x8A `ADC addr16` (10 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [PCM] [RO,OHI,CE] [ORO,MI] [OHO,MIH] [RO,BI] [CI,EO,AI,FI]

0x8B `ADC addr16,X` (12 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [PCM] [RO,OHI,CE] [XO,TI] [ATM,AOP,EO,MI,ACO] [OHO,TI] [ATM,AHC,EO,MIH] [RO,BI] [CI,EO,AI,FI]

0x8C ADC addr16,Y (12 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [PCM] [RO,OHI,CE] [YO,TI] [ATM,AOP,EO,MI,ACO] [OHO,TI] [ATM,AHC,EO,MIH] [RO,BI] [CI,EO,AI,FI]

0x8D ADC (addr8) (11 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [ORO,MI] [RO,TI] [MRI] [RO,MIH] [TO,MIL] [RO,BI] [CI,EO,AI,FI]

0x8E ADC addr8,X (8 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [XO,TI] [ATM,AOP,EO,MI] [RO,BI] [CI,EO,AI,FI]

0x8F ADC (addr8),Y (14 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [ORO,MI] [RO,ORI] [YO,TI] [MRI] [RO,OHI] [ATM,AOP,EO,MI,ACO] [OHO,TI] [ATM,AHC,EO,MIH] [RO,BI] [CI,EO,AI,FI]

0x90 SUB #n (6 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [ORO,BI] [SU,EO,AI,FI]

0x91 SUB addr8 (7 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [ORO,MI] [RO,BI] [SU,EO,AI,FI]

0x92 SUB addr16 (10 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [PCM] [RO,OHI,CE] [ORO,MI] [OHO,MIH] [RO,BI] [SU,EO,AI,FI]

0x93 SUB addr16,X (12 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [PCM] [RO,OHI,CE] [XO,TI] [ATM,AOP,EO,MI,ACO] [OHO,TI] [ATM,AHC,EO,MIH] [RO,BI] [SU,EO,AI,FI]

0x94 SUB addr16,Y (12 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [PCM] [RO,OHI,CE] [YO,TI] [ATM,AOP,EO,MI,ACO] [OHO,TI] [ATM,AHC,EO,MIH] [RO,BI] [SU,EO,AI,FI]

0x95 SUB (addr8) (11 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [ORO,MI] [RO,TI] [MRI] [RO,MIH] [TO,MIL] [RO,BI] [SU,EO,AI,FI]

0x96 SUB addr8,X (8 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [XO,TI] [ATM,AOP,EO,MI] [RO,BI] [SU,EO,AI,FI]

0x97 SUB (addr8),Y (14 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [ORO,MI] [RO,ORI] [YO,TI] [MRI] [RO,OHI] [ATM,AOP,EO,MI,ACO] [OHO,TI] [ATM,AHC,EO,MIH] [RO,BI] [SU,EO,AI,FI]

0x98 SBC #n (6 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [ORO,BI] [SU,CI,EO,AI,FI]

0x99 SBC addr8 (7 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [ORO,MI] [RO,BI] [SU,CI,EO,AI,FI]

0x9A SBC addr16 (10 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [PCM] [RO,OHI,CE] [ORO,MI] [OHO,MIH] [RO,BI] [SU,CI,EO,AI,FI]

0x9B SBC addr16,X (12 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [PCM] [RO,OHI,CE] [XO,TI] [ATM,AOP,EO,MI,ACO] [OHO,TI] [ATM,AHC,EO,MIH] [RO,BI] [SU,CI,EO,AI,FI]

0x9C SBC addr16,Y (12 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [PCM] [RO,OHI,CE] [YO,TI] [ATM,AOP,EO,MI,ACO] [OHO,TI] [ATM,AHC,EO,MIH] [RO,BI] [SU,CI,EO,AI,FI]

0x9D SBC (addr8) (11 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [ORO,MI] [RO,TI] [MRI] [RO,MIH] [TO,MIL] [RO,BI] [SU,CI,EO,AI,FI]

0x9E SBC addr8,X (8 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [XO,TI] [ATM,AOP,EO,MI] [RO,BI] [SU,CI,EO,AI,FI]

0x9F SBC (addr8),Y (14 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [ORO,MI] [RO,ORI] [YO,TI] [MRI] [RO,OHI] [ATM,AOP,EO,MI,ACO] [OHO,TI] [ATM,AHC,EO,MIH] [RO,BI] [SU,CI,EO,AI,FI]

0xA0 AND #n (6 T)

[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [ORO,BI] [AND,EO,AI,FI]

0xA1 AND addr8 (7 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [ORO,MI] [RO,BI] [AND,EO,AI,FI]

0xA2 AND addr16 (10 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [PCM] [RO,OHI,CE] [ORO,MI] [OHO,MIH] [RO,BI] [AND,EO,AI,FI]

0xA3 AND addr16,X (12 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [PCM] [RO,OHI,CE] [XO,TI] [ATM,AOP,EO,MI,ACO] [OHO,TI] [ATM,AHC,EO,MIH] [RO,BI] [AND,EO,AI,FI]

0xA4 AND addr16,Y (12 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [PCM] [RO,OHI,CE] [YO,TI] [ATM,AOP,EO,MI,ACO] [OHO,TI] [ATM,AHC,EO,MIH] [RO,BI] [AND,EO,AI,FI]

0xA5 AND (addr8) (11 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [ORO,MI] [RO,TI] [MRI] [RO,MIH] [TO,MIL] [RO,BI] [AND,EO,AI,FI]

0xA6 AND addr8,X (8 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [XO,TI] [ATM,AOP,EO,MI] [RO,BI] [AND,EO,AI,FI]

0xA7 AND (addr8),Y (14 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [ORO,MI] [RO,ORI] [YO,TI] [MRI] [RO,OHI] [ATM,AOP,EO,MI,ACO] [OHO,TI] [ATM,AHC,EO,MIH] [RO,BI] [AND,EO,AI,FI]

0xA8 OR #n (6 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [ORO,BI] [OR,EO,AI,FI]

0xA9 OR addr8 (7 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [ORO,MI] [RO,BI] [OR,EO,AI,FI]

0xAA OR addr16 (10 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [PCM] [RO,OHI,CE] [ORO,MI] [OHO,MIH] [RO,BI] [OR,EO,AI,FI]

0xAB OR addr16,X (12 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [PCM] [RO,OHI,CE] [XO,TI] [ATM,AOP,EO,MI,ACO] [OHO,TI] [ATM,AHC,EO,MIH] [RO,BI] [OR,EO,AI,FI]

0xAC OR addr16,Y (12 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [PCM] [RO,OHI,CE] [YO,TI] [ATM,AOP,EO,MI,ACO] [OHO,TI] [ATM,AHC,EO,MIH] [RO,BI] [OR,EO,AI,FI]

0xAD OR (addr8) (11 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [ORO,MI] [RO,TI] [MRI] [RO,MIH] [TO,MIL] [RO,BI] [OR,EO,AI,FI]

0xAE OR addr8,X (8 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [XO,TI] [ATM,AOP,EO,MI] [RO,BI] [OR,EO,AI,FI]

0xAF OR (addr8),Y (14 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [ORO,MI] [RO,ORI] [YO,TI] [MRI] [RO,OHI] [ATM,AOP,EO,MI,ACO] [OHO,TI] [ATM,AHC,EO,MIH] [RO,BI] [OR,EO,AI,FI]

0xB0 XOR #n (6 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [ORO,BI] [XOR,EO,AI,FI]

0xB1 XOR addr8 (7 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [ORO,MI] [RO,BI] [XOR,EO,AI,FI]

0xB2 XOR addr16 (10 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [PCM] [RO,OHI,CE] [ORO,MI] [OHO,MIH] [RO,BI] [XOR,EO,AI,FI]

0xB3 XOR addr16,X (12 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [PCM] [RO,OHI,CE] [XO,TI] [ATM,AOP,EO,MI,ACO] [OHO,TI] [ATM,AHC,EO,MIH] [RO,BI] [XOR,EO,AI,FI]

0xB4 XOR addr16,Y (12 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [PCM] [RO,OHI,CE] [YO,TI] [ATM,AOP,EO,MI,ACO] [OHO,TI] [ATM,AHC,EO,MIH] [RO,BI] [XOR,EO,AI,FI]

0xB5 `XOR (addr8)` (11 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [ORO,MI] [RO,TI] [MRI] [RO,MIH] [TO,MIL] [RO,BI] [XOR,EO,AI,FI]

0xB6 `XOR addr8,X` (8 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [XO,TI] [ATM,AOP,EO,MI] [RO,BI] [XOR,EO,AI,FI]

0xB7 `XOR (addr8),Y` (14 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [ORO,MI] [RO,ORI] [YO,TI] [MRI] [RO,OHI] [ATM,AOP,EO,MI,ACO] [OHO,TI] [ATM,AHC,EO,MIH] [RO,BI] [XOR,EO,AI,FI]

0xB8 `CMP #n` (6 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [ORO,BI] [CMP,EO,FI]

0xB9 `CMP addr8` (7 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [ORO,MI] [RO,BI] [CMP,EO,FI]

0xBA `CMP addr16` (10 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [PCM] [RO,OHI,CE] [ORO,MI] [OHO,MIH] [RO,BI] [CMP,EO,FI]

0xBB `CMP addr16,X` (12 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [PCM] [RO,OHI,CE] [XO,TI] [ATM,AOP,EO,MI,ACO] [OHO,TI] [ATM,AHC,EO,MIH] [RO,BI] [CMP,EO,FI]

0xBC `CMP addr16,Y` (12 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [PCM] [RO,OHI,CE] [YO,TI] [ATM,AOP,EO,MI,ACO] [OHO,TI] [ATM,AHC,EO,MIH] [RO,BI] [CMP,EO,FI]

0xBD `CMP (addr8)` (11 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [ORO,MI] [RO,TI] [MRI] [RO,MIH] [TO,MIL] [RO,BI] [CMP,EO,FI]

0xBE `CMP addr8,X` (8 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [XO,TI] [ATM,AOP,EO,MI] [RO,BI] [CMP,EO,FI]

0xBF `CMP (addr8),Y` (14 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [ORO,MI] [RO,ORI] [YO,TI] [MRI] [RO,OHI] [ATM,AOP,EO,MI,ACO] [OHO,TI] [ATM,AHC,EO,MIH] [RO,BI] [CMP,EO,FI]

0xC0 `JNZ addr16` (7 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [PCM] [RO,OHI,CE] [JPW]?

0xC1 `JZ addr16` (7 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [PCM] [RO,OHI,CE] [JPW]?

0xC2 `JNC addr16` (7 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [PCM] [RO,OHI,CE] [JPW]?

0xC3 `JC addr16` (7 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [PCM] [RO,OHI,CE] [JPW]?

0xC4 `JP addr16` (7 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [PCM] [RO,OHI,CE] [JPW]?

0xC5 `JN addr16` (7 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [PCM] [RO,OHI,CE] [JPW]?

0xC6 `JNV addr16` (7 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [PCM] [RO,OHI,CE] [JPW]?

0xC7 `JV addr16` (7 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [PCM] [RO,OHI,CE] [JPW]?

0xC8 `JMP addr16` (7 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [PCM] [RO,OHI,CE] [JPW]

0xC9 `JMP (addr8)` (9 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [ORO,MI] [RO,TI] [MRI] [RO,JPH] [TO,JPL]

0xCA `CALL addr16` (11 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [PCM] [RO,OHI,CE] [SD,SPM] [CHO,RI] [SD,SPM] [CLO,RI] [JPW]

0xCB `RET` (6 T)

[PCM] [RO,II,CE] [SPM] [RO,JPL,SE] [SPM] [RO,JPH,SE]

0xCC RTI (8 T)
[PCM] [RO,II,CE] [SPM] [RO,FLI,SE] [SPM] [RO,JPL,SE] [SPM] [RO,JPH,SE]

0xCD BRK (12 T)
[PCM] [RO,II,CE] [SD,SPM] [CHO,RI] [SD,SPM] [CLO,RI] [SD,SPM] [FLO,RI,DIF] [VBR] [RO,JPL] [MRI] [RO,JPH]

0xCE CALL (addr8) (13 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [SD,SPM] [CHO,RI] [SD,SPM] [CLO,RI] [ORO,MI] [RO,TI] [MRI] [RO,JPH] [TO,JPL]

0xD0 INC addr8 (7 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [ORO,MI] [RO,TI] [ATM,INC,EO,RI,FI]

0xD1 INC addr16 (10 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [PCM] [RO,OHI,CE] [ORO,MI] [OHO,MIH] [RO,TI] [ATM,INC,EO,RI,FI]

0xD2 DEC addr8 (7 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [ORO,MI] [RO,TI] [ATM,DEC,EO,RI,FI]

0xD3 DEC addr16 (10 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [PCM] [RO,OHI,CE] [ORO,MI] [OHO,MIH] [RO,TI] [ATM,DEC,EO,RI,FI]

0xD4 INW addr8 (10 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [ORO,MI] [RO,TI] [ATM,INC,EO,RI,ACO] [MRI] [RO,TI] [ATM,AHC,EO,RI]

0xD5 INW addr16 (13 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [PCM] [RO,OHI,CE] [ORO,MI] [OHO,MIH] [RO,TI] [ATM,INC,EO,RI,ACO] [MRI] [RO,TI] [ATM,AHC,EO,RI]

0xD6 DEW addr8 (10 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [ORO,MI] [RO,TI] [ATM,DEC,EO,RI,ACO] [MRI] [RO,TI] [ATM,AHB,EO,RI]

0xD7 DEW addr16 (13 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [PCM] [RO,OHI,CE] [ORO,MI] [OHO,MIH] [RO,TI] [ATM,DEC,EO,RI,ACO] [MRI] [RO,TI] [ATM,AHB,EO,RI]

0xD8 SHL addr8 (7 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [ORO,MI] [RO,TI] [ATM,SHL,EO,RI,FI]

0xD9 SHL addr16 (10 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [PCM] [RO,OHI,CE] [ORO,MI] [OHO,MIH] [RO,TI] [ATM,SHL,EO,RI,FI]

0xDA SHR addr8 (7 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [ORO,MI] [RO,TI] [ATM,SHR,EO,RI,FI]

0xDB SHR addr16 (10 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [PCM] [RO,OHI,CE] [ORO,MI] [OHO,MIH] [RO,TI] [ATM,SHR,EO,RI,FI]

0xDC ROL addr8 (7 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [ORO,MI] [RO,TI] [ATM,ROL,EO,RI,FI]

0xDD ROL addr16 (10 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [PCM] [RO,OHI,CE] [ORO,MI] [OHO,MIH] [RO,TI] [ATM,ROL,EO,RI,FI]

0xDE ROR addr8 (7 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [ORO,MI] [RO,TI] [ATM,ROR,EO,RI,FI]

0xDF ROR addr16 (10 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [PCM] [RO,OHI,CE] [ORO,MI] [OHO,MIH] [RO,TI] [ATM,ROR,EO,RI,FI]

0xE0 JGT addr16 (7 T)
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [PCM] [RO,OHI,CE] [JPW]?

0xE1 JLE addr16 (7 T)

```
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [PCM] [RO,OHI,CE] [JPW]?
```

0xE2 JGES addr16 (7 T)
```
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [PCM] [RO,OHI,CE] [JPW]?
```

0xE3 JLTS addr16 (7 T)
```
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [PCM] [RO,OHI,CE] [JPW]?
```

0xE4 BIT #n (6 T)
```
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [ORO,BI] [AND,EO,FI]
```

0xE5 BIT addr8 (7 T)
```
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [ORO,MI] [RO,BI] [AND,EO,FI]
```

0xE6 BIT addr16 (10 T)
```
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [PCM] [RO,OHI,CE] [ORO,MI] [OHO,MIH] [RO,BI]
[AND,EO,FI]
```

0xE8 ADW addr8,#n (13 T)
```
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [PCM] [RO,OHI,CE] [ORO,MI] [OHO,BI] [RO,TI]
[ATM,EO,RI,ACO] [MRI] [RO,TI] [ATM,AHC,EO,RI]
```

0xE9 SBW addr8,#n (13 T)
```
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [PCM] [RO,OHI,CE] [ORO,MI] [OHO,BI] [RO,TI]
[ATM,SU,EO,RI,ACO] [MRI] [RO,TI] [ATM,AHB,EO,RI]
```

0xEA LXY #nn (8 T)
```
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [PCM] [RO,OHI,CE] [ORO,YI] [OHO,XI]
```

0xEB SXY addr8 (8 T)
```
[PCM] [RO,II,CE] [PCM] [RO,ORI,CE] [ORO,MI] [YO,RI] [MRI] [XO,RI]
```

0xFD RESET (4 T)
```
[VRS] [RO,JPL] [MRI] [RO,JPH]
```

0xFE IRQ (10 T)
```
[SD,SPM] [CHO,RI] [SD,SPM] [CLO,RI] [SD,SPM] [FLO,RI,DIF] [VIR] [RO,JPL] [MRI] [RO,JPH]
```

PŘÍLOHA D

Řídicí signály

Všech 70 řídicích signálů, rozdělených do šesti vzájemně vylučujících skupin. To dělení není jen dokumentační pomůcka — je to formát mikroslova, ze kterého se generuje paměť pro FPGA, a **pořadí členů uvnitř skupiny je součástí kódování**.

Konvence jmen: signál končící na `O` typicky něco dává na datovou sběrnici, signál končící na `I` si to ze sběrnice bere. V jednom T-stavu smí být aktivní nejvýš jeden zdroj sběrnice; cílů může být víc.

Zdroje datové sběrnice

Signál	Popis
`RO`	čtení paměti — bajt z adresy v MAR na datovou sběrnici
`AO`	registr A na sběrnici
`BO`	registr B na sběrnici
`XO`	registr X na sběrnici
`YO`	registr Y na sběrnici
`TO`	registr TMP na sběrnici
`ORO`	operandový registr (dolní bajt) na sběrnici
`OHO`	operandový registr (horní bajt) na sběrnici
`CLO`	dolní polovina PC na sběrnici (uložení při CALL/IRQ)
`CHO`	horní polovina PC na sběrnici (uložení při CALL/IRQ)
`SLO`	dolní polovina SP na sběrnici (TSXY)
`SHO`	horní polovina SP na sběrnici (TSXY)
`FLO`	příznaky na sběrnici — C/Z/N/V/I zabalené do stavového bajtu
`EO`	výsledek ALU na sběrnici
`EXO`	sekundární výsledek ALU na sběrnici (horní bajt součinu, zbytek po dělení)

Cíle datové sběrnice

Signál	Popis
`MI`	MAR := 0x00:sběrnice — adresa v nulté stránce (horní polovina se vynuluje)
`MIL`	dolní polovina MAR := sběrnice
`MIH`	horní polovina MAR := sběrnice
`RI`	zápis do paměti — bajt ze sběrnice na adresu v MAR
`II`	instrukční registr := sběrnice
`AI`	registr A := sběrnice
`BI`	registr B := sběrnice
`XI`	registr X := sběrnice
`YI`	registr Y := sběrnice
`TI`	registr TMP := sběrnice
`ORI`	operandový registr (dolní bajt) := sběrnice
`OHI`	operandový registr (horní bajt) := sběrnice
`OI`	výstupní registr OUT := sběrnice
`JPL`	dolní polovina PC := sběrnice (návrat z RET/RTI)
`JPH`	horní polovina PC := sběrnice (návrat z RET/RTI)
`SLI`	dolní polovina SP := sběrnice (TXYS)
`SHI`	horní polovina SP := sběrnice (TXYS)
`FLI`	příznaky := sběrnice — obnovení C/Z/N/V/I ze stavového bajtu

Adresní cesta (MAR)

Signál	Popis
`PCM`	MAR := PC (šestnáctibitová adresní cesta)
`SPM`	MAR := SP (šestnáctibitová adresní cesta)
`MRI`	MAR := MAR + 1 — druhý bajt ukazatele nebo vektoru
`VRS`	MAR := 0xFFFC (báze vektoru RESET)
`VIR`	MAR := 0xFFFE (báze vektoru IRQ)
`VBR`	MAR := 0xFFFA (báze vektoru BRK)

Režimy ALU

Signál	Popis
`SU`	režim ALU: odečítání
`AND`	režim ALU: logický součin
`OR`	režim ALU: logický součet
`XOR`	režim ALU: nonekvivalence
`NOT`	režim ALU: negace bitů
`INC`	režim ALU: zvýšení o jedna
`DEC`	režim ALU: snížení o jedna
`SHL`	režim ALU: posun vlevo
`SHR`	režim ALU: posun vpravo
`ROL`	režim ALU: rotace vlevo přes příznak C
`ROR`	režim ALU: rotace vpravo přes příznak C
`CMP`	režim ALU: porovnání — nastaví jen příznaky, výsledek se zahodí
`MUL`	režim ALU: násobení — dolní bajt na výstup, horní přes EXO
`DIV`	režim ALU: dělení — podíl na výstup, zbytek přes EXO
`AHC`	ALU := TMP + addrCarry — přičtení přenosu k hornímu bajtu adresy
`AHB`	ALU := TMP − addrCarry — odečtení výpůjčky od horního bajtu adresy

Vedlejší efekty

Signál	Popis
`CFC`	vynuluj příznak C (CLC)
`CFS`	nastav příznak C (SEC)
`EIF`	povol přerušení — nastav příznak I (EI)
`DIF`	zakaž přerušení — vynuluj příznak I (DI)
`HLT`	zastav procesor, dokud nepřijde povolené přerušení
`RND`	náhodný bajt do A (xorshift32 se zadatelným semínkem)

Samostatné bity mikroslova

Signál	Popis
`JPW`	PC := OPH:OP (šestnáctibitová adresní cesta — absolutní skok)
`CE`	zvýšení programového čítače: PC := PC + 1

Signál	Popis
`SD`	snížení zásobníkového ukazatele (šestnáctibitově)
`SE`	zvýšení zásobníkového ukazatele (šestnáctibitově)
`CI`	přenos do ALU z příznaku C (dělá z ADD instrukci ADC, ze SUB instrukci SBC)
`ATM`	multiplexor ALU — první vstup z TMP místo z A
`AOP`	multiplexor ALU — druhý vstup z operandového registru místo z B
`ACO`	ulož přenos či výpůjčku z ALU do záklopky addrCarry (příznaky se nemění)
`FI`	zápis příznaků — s EO příznaky z ALU, jinak Z/N podle bajtu na sběrnici

PŘÍLOHA E

Paměťová mapa a registry periferií

Adresní prostor

Adresa	Velikost	Oblast
`0x0000–0x00FF`	256 B	Nultá stránka — rychlý přístup a soubor ukazatelů
`0x0100–0x01FF`	256 B	Zásobník
`0x0200–0x7FFF`	32256 B	Program a data
`0x8000–0x83E7`	1000 B	Znakový buffer (40×25)
`0x83E8–0x83FF`	24 B	rezervováno
`0x8400–0x8BFF`	2048 B	Glyph RAM (8 bajtů na znak)
`0x8C00–0x8FE7`	1000 B	Barevná RAM (atribut na buňku)
`0x8FE8–0x8FFF`	24 B	rezervováno
`0x9000–0xAF3F`	8000 B	Bitmapový framebuffer
`0xAF40–0xAFDF`	160 B	rezervováno
`0xAFE0–0xAFFF`	32 B	Atributy spritů (8 × 4 bajty)
`0xB000–0xBFFF`	4096 B	Vzory spritů (32 × 128 bajtů)
`0xC000–0xFEFF`	16128 B	Datová RAM — velká pole, halda
`0xFF00–0xFF7F`	128 B	Registry periferií (MMIO)
`0xFF80–0xFFF9`	122 B	Systémová RAM — scratch pro ISR, domov boot ROM
`0xFFFA–0xFFFF`	6 B	Vektory BRK / RESET / IRQ

Součet všech řádků je přesně 64 KB — řádky označené jako rezervované jsou díry, které dnes nikdo nepoužívá.

Vektory

Adresa	Vektor	Čte se
0xFFFA	BRK	při provedení instrukce `BRK`
0xFFFC	RESET	po zapnutí a po resetu
0xFFFE	IRQ	při přijetí hardwarového přerušení

Všechny jsou dvoubajtové, dolním bajtem napřed.

Registry periferií

Blok 0xFF00–0xFF7F, osm zařízení po šestnácti bajtech. Číslo zařízení jsou bity 6 až 4 adresy.

Klávesnice

Adresa	Registr
0xFF00	KBD_DATA
0xFF01	KBD_STATUS

Terminál

Adresa	Registr
0xFF10	TERM_DATA
0xFF11	TERM_STATUS

Zvuk

Adresa	Registr
0xFF20	AUD_CTRL
0xFF21	AUD_FREQ_LO
0xFF22	AUD_FREQ_HI
0xFF23	AUD_WAVE
0xFF24	AUD_AD
0xFF25	AUD_SR

Časovač a počitadla

Adresa	Registr
0xFF30	TMR_DIV
0xFF31	TMR_CNT
0xFF32	CYC0
0xFF33	CYC1
0xFF34	CYC2
0xFF35	CYC3
0xFF36	UPT_LO
0xFF37	UPT_HI
0xFF38	CLK_TPMS_LO

Adresa	Registr
0xFF39	CLK_TPMS_HI
0xFF3A	CLK_MS_LO
0xFF3B	CLK_MS_HI

Řízení přerušení

Adresa	Registr
0xFF40	IRQ_ENABLE
0xFF41	IRQ_STATUS

Video

Adresa	Registr
0xFF50	VID_CTRL
0xFF51	VID_STATUS
0xFF52	VID_CURX
0xFF53	VID_CURY
0xFF54	VID_BORDER
0xFF55	VID_SCROLLX
0xFF56	VID_SCROLLY
0xFF57	VID_MCOLOR
0xFF58	SPR_ENABLE

Úložiště a sériová linka

Adresa	Registr
0xFF60	DSK_CMD
0xFF61	DSK_STATUS
0xFF62	DSK_BLK
0xFF63	DSK_ADDR_LO
0xFF64	DSK_ADDR_HI
0xFF65	SER_DATA
0xFF66	SER_STATUS
0xFF67	SER_CTRL
0xFF68	DSK_DISK

Gamepady a blitter

Adresa	Registr
0xFF70	PAD1
0xFF71	PAD2
0xFF72	BLT_SRC_LO
0xFF73	BLT_SRC_HI
0xFF74	BLT_DST_LO
0xFF75	BLT_DST_HI
0xFF76	BLT_LEN_LO
0xFF77	BLT_LEN_HI
0xFF78	BLT_FILL
0xFF79	BLT_CTRL
0xFF7A	BLT_STATUS

Kanály zvukového zařízení

Tři kanály opakují tutéž pětici registrů:

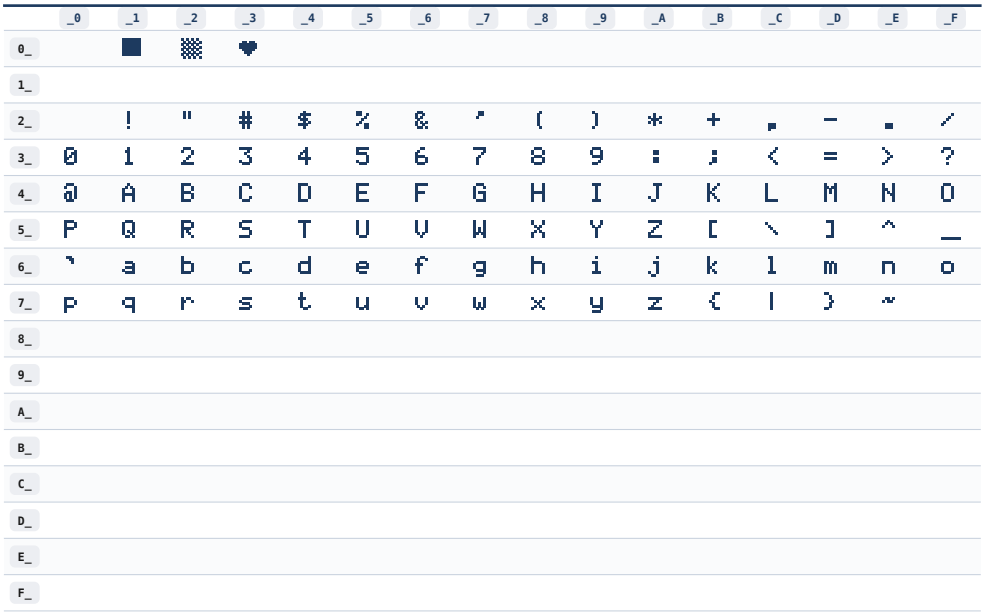
Adresa	Registr
0xFF21	AUD0_FREQ_LO
0xFF22	AUD0_FREQ_HI
0xFF23	AUD0_WAVE
0xFF24	AUD0_AD
0xFF25	AUD0_SR
0xFF26	AUD1_FREQ_LO
0xFF27	AUD1_FREQ_HI
0xFF28	AUD1_WAVE
0xFF29	AUD1_AD
0xFF2A	AUD1_SR
0xFF2B	AUD2_FREQ_LO
0xFF2C	AUD2_FREQ_HI
0xFF2D	AUD2_WAVE
0xFF2E	AUD2_AD
0xFF2F	AUD2_SR

PŘÍLOHA F

Znaková sada a paleta

Zabudované písmo

Paměť glyfů má 256 slotů. Zabudované písmo z nich zaplní 97 — tisknutelnou část ASCII (0x20–0x7E) a tři symboly na 0x01–0x03. Řádek je horní půlbajt kódu, sloupec dolní.



Obrázek 14. Zabudované písmo. Prázdná pole jsou volné sloty, ne mezery.

Celá horní polovina (0x80–0xFF) je **prázdná a k dispozici**. Je to nejjednodušší místo, kam si program uloží vlastní symboly, aniž by přišel o písmo — dlaždice, ikony, části hracího pole.

Glyfy nejsou v ROM — leží v paměti na 0x8400–0x8BFF, osm bajtů na znak, jeden bajt na řádek, nejvyšší bit vlevo. Program je smí kdykoli přepsat.

> **Poznámka**

Při resetu i při nahrání programu se glyfy naplní tímto písmem — **pokud** nahraný obraz nepoložil do oblasti glyfů žádný nenulový bajt. Program s vlastním písmem tedy vyhrává a nemusí nic vypínat.

Tisknutelné znaky

Rozsah 0x20–0x7F odpovídá ASCII.

Opkód	Znak	Opkód	Znak	Opkód	Znak	Opkód	Znak
0x20		0x38	8	0x50	P	0x68	h
0x21	!	0x39	9	0x51	Q	0x69	i
0x22	"	0x3A	:	0x52	R	0x6A	j
0x23	#	0x3B	;	0x53	S	0x6B	k
0x24	$	0x3C	<	0x54	T	0x6C	l
0x25	%	0x3D	=	0x55	U	0x6D	m
0x26	&	0x3E	>	0x56	V	0x6E	n
0x27	'	0x3F	?	0x57	W	0x6F	o
0x28	(	0x40	@	0x58	X	0x70	p
0x29	)	0x41	A	0x59	Y	0x71	q
0x2A	*	0x42	B	0x5A	Z	0x72	r
0x2B	+	0x43	C	0x5B	[	0x73	s
0x2C	,	0x44	D	0x5C	\	0x74	t
0x2D	-	0x45	E	0x5D	]	0x75	u
0x2E	.	0x46	F	0x5E	^	0x76	v
0x2F	/	0x47	G	0x5F	_	0x77	w
0x30	0	0x48	H	0x60	`	0x78	x
0x31	1	0x49	I	0x61	a	0x79	y
0x32	2	0x4A	J	0x62	b	0x7A	z
0x33	3	0x4B	K	0x63	c	0x7B	{
0x34	4	0x4C	L	0x64	d	0x7C	|
0x35	5	0x4D	M	0x65	e	0x7D	}
0x36	6	0x4E	N	0x66	f	0x7E	~
0x37	7	0x4F	O	0x67	g	0x7F	

Obrázek 15. Tisknutelná část znakové sady s kódy.

Řídicí kódy terminálu

Kód	Zápis	Účinek
0x08	`\b`	smaže předchozí znak
0x09	`\t`	tabulátor po osmi sloupcích
0x0A	`\n`	nový řádek

Kód	Zápis	Účinek
0x0C	`\f`	smaže obrazovku terminálu
0x0D	`\r`	návrat na začátek řádku
0x00	`\0`	ukončovač řetězce

Paleta

Index		Barva	Index		Barva
0		#000000	8		#8b5429
1		#ffffff	9		#574200
2		#883932	10		#b86962
3		#67b6bd	11		#505050
4		#8b3f96	12		#787878
5		#55a049	13		#94e089
6		#40318d	14		#7869c4
7		#bfce72	15		#9f9f9f

Obrázek 16. Šestnáctibarevná paleta, sdílená barevnou RAM, registrem `VID_MCOLOR` a okrajem `VID_BORDER`.

Index se do barevné RAM zapisuje po půlbajtech: dolní je popředí, horní pozadí. Hodnota 0x00 znamená výchozí vzhled, ne černou na černé.

PŘÍLOHA G

Chyby a pasti

Chyby vznikají ve třech různých okamžicích a je užitečné je od sebe odlišit: při překladu, při běhu, a nakonec ty, které se nikde neohlásí a jen tiše udělají něco jiného, než člověk čekal.

Chyby překladu

Hlášení nese číslo řádku, a u kódu vzniklého z makra i kontext rozvinutí (`Line 12 (in macro DELAY expanded at line 30)`).

Struktura a symboly

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

Operandy a adresní režimy

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

Hlášení
`BIT supports only immediate, zero-page or absolute operands`
`LXY requires a 16-bit immediate: LXY #value`
`LXY takes the full 16-bit value (no < or > byte selector)`
`SXY requires a zero-page address: SXY zp`
`Missing value after #` · `Empty indirect operand ()`
`Invalid operand for XYZ`

Direktivy, rozsahy a literály

Hlášení
`EQU directive requires a name` / `… requires a value`
`EQU name cannot be a local label`
`EQU value must be a constant expression (labels are not allowed): …`
`EQU value … out of 16-bit range`
`.ORG requires an address` / `.ORG address must be a constant expression: …`
`.ORG address … out of range (0x0000-0xFFFF)`
`.VECTOR requires: .VECTOR RESET|IRQ|BRK, target`
`Unknown vector: … (expected RESET, IRQ or BRK)`
`DATA requires at least one value` / `.WORD requires at least one value`
`Value … out of byte range (-128 to 255)`
`Address … is out of range (0x0000-0xFFFF)`
`Program overflows past 0xFFFF (at 0x….)`
`Malformed character literal: …` / `Unknown escape sequence: \…`
`Character literal must be a single character: …`
`Unterminated quote in value list` / `Malformed string literal: …`

Makroprocesor

Hlášení
`Unknown preprocessor directive: %xyz`
`Invalid %ifdef - expected %ifdef NAME` (obdobně `%ifndef`)
`Undefined identifier in %if: NAME (define it with %define or test with %ifdef)`
`%define expansion produced a preprocessor directive: %xyz`
`%include is not available in this environment`

Chyby za běhu

Procesor se zastaví jen ve dvou případech a oba se ohlásí i s adresou.

Situace	Co se stane
ilegální opkód	zastavení, hlášení s adresou; 0x00 a 0xFF mají vlastní diagnostiku
kód nebo data v bloku periferií	odmítne už assembler — za běhu k tomu nedojde

Dvě nejčastější varianty ilegálního opkódu:

Bajt	Obvyklá příčina
0x00	program skočil do nulami vyplněné oblasti — špatná adresa skoku nebo poškozený ukazatel
0xFF	program přetekl za svůj konec — chybí `HLT` nebo `RET`

Chyby, které se neohlásí

Tohle je nejdůležitější část přílohy. Následující situace jsou **legální** — procesor udělá přesně, co je napsané — a přesto skoro vždycky znamenají chybu.

Co	Proč to překvapí
`LDA 42` místo `LDA #42`	obojí je legální; první čte z adresy 42
`TAB` a pak jakákoli operace ALU	operand přepíše **B** hodnota je pryč
`TBA` po `MUL` / `DIV` s `CMP` mezi tím	totéž — sekundární výsledek se ztratí
znaménkový skok po `CMP`	`CMP` nenastavuje `V`, `JGES` / `JLTS` čtou starou hodnotu
test nulovosti po `INW` / `DEW`	operace nad slovem nenastavují žádný příznak
test nulovosti po `RND`	`RND` nenastavuje žádný příznak
`Z` a `N` po dělení nulou	`DIV` s nulovým dělitelem definuje jen `C`
`LDA 0xF0,X` s velkým `X`	`zp,X` se zabalí uvnitř nulté stránky
dvojí čtení stavového registru	první odečet smazal chybový bit
zápis do plné fronty terminálu	bajt se zahodí, ohlásí se jen sticky bitem
zápis bitu brány, který už je nastavený	tón se nespustí znovu

Co	Proč to překvapí
překrytí proměnných v nulté stránce	nikdo nehlídá; projeví se jako poškozená data
přetečení zásobníku	nikdo nehlídá; přepíše nultou stránku
zpožďovací smyčka počítaná na průchody	na desce běží dvanáctkrát rychleji

Past

Poslední tři řádky mají společné to, že se neprojeví v místě, kde vznikly. Nástroj, který na ně zabírá, není čtení kódu, ale **watchpoint**: procesor umí zastavit na zápis nebo čtení konkrétní adresy, a to i uvnitř DMA přenosu.

PŘÍLOHA H

Nástroje

Spuštění programu z příkazové řádky

```
npm run asm <soubor.asm> [přepínače]
```

Přepínač	Účinek
`--debug`	podrobný výpis průběhu
`--max-cycles N`	strop počtu cyklů (výchozí 10 000)
`--show-memory`	výpis paměti na konci
`--show-output`	jen výstup programu
`--seed N`	semínko pro `RND` — reprodukovatelný běh
`--input S`	skriptovaný vstup z klávesnice (zná `\n`, `\r`, `\e`)
`--serial-in S`	skriptovaná data na sériové lince, dávkovaná podle fronty
`--disk F`	obraz disku pro slot 0 (po běhu se aktualizuje)
`--card F`	obraz celé karty, 16 × 64 KB
`--boot`	bootovací režim: vyrob z programu bootovací disk a spusť ho přes boot ROM
`--make-boot-disk F`	zapiš bootovací disk do souboru a skonči (nespouští)
`--boot-name S`	jméno programu v hlavičce boot sektoru
`--boot-data-disk N`	datový disk zapsaný do hlavičky
`--include-dir D`	výchozí adresář pro `%include`

Návratové kódy: 1 chyba překladu, 2 zastavení za běhu, 3 vyčerpaný strop cyklů.

> **Poznámka**
>
> Přepínač `--seed` je to, co dělá běh s instrukcí `RND` reprodukovatelným. Bez něj se generátor inicializuje jinak při každém spuštění a výstup se nedá porovnávat.

Konfigurační direktivy

Program může nastavit prostředí, ve kterém se otevře, komentářem na začátku souboru:

```
; @config view: schematic
; @config memoryTab: memory
; @config speed: 50
; @config autorun: true
```

Direktiva	Hodnoty	Význam
`view`	`blocks`, `schematic`	blokové nebo detailní schéma procesoru
`memoryTab`	`memory`, `stack`	která záložka paměti se ukáže
`speed`	1 až 2 000 000	výchozí rychlost hodin v Hz
`autorun`	`true`	program se po otevření sám spustí

Direktivy jsou obyčejné komentáře, takže překlad nijak neovlivní a program běží stejně i mimo emulátor.

Ladicí nástroje

Nástroj	K čemu
krokování po instrukcích	zastavení na hranici instrukce
krokování po T-stavech	jeden mikrokrok — vidět, které signály jsou aktivní
návrat v čase	krok zpět; v režimu T-stavů je přesnou inverzí kroku vpřed
body přerušení	zastavení na řádku zdrojáku
watchpointy	zastavení na čtení nebo zápis adresy či na hodnotu registru
disassembler	zpětný překlad paměti na instrukce

Past

Návrat v čase má hranici. Nad 10 kHz („turbo“) se stav ukládá jen na každé 64. hranici instrukce, takže krok zpět skáče po 64 instrukcích. Ruční

krokování ukládá vždy každou hranici, takže při ladění je granularita plná.

PŘÍLOHA I

Slovníček pojmů

Česko-anglický slovníček termínů použitých v této knize. Anglické tvary jsou ty, které se vyskytují ve zdrojových kódech, ve specifikaci `docs/ISA3.md` a v anglickém vydání této knihy.

Česky	Anglicky	Význam
adresní režim	addressing mode	pravidlo, podle kterého se z instrukce určí, kde leží operand
adresní cesta	address path	šestnáctibitová cesta pro tři přenosy: PC→MAR, SP→MAR, skok
akumulátor	accumulator	registr A — cíl většiny aritmetiky
bezoperandová instrukce	implied	instrukce bez operandu, dlouhá jeden bajt
blitter	blitter	jednotka pro kopírování a vyplňování bloků paměti
brána	gate	bit, který spouští a zastavuje tón zvukového kanálu
čti-uprav-zapiš	read-modify-write, RMW	operace, která bajt přečte z paměti, změní a zapíše zpět
datová sběrnice	data bus	osmibitová cesta, po které se pohybují všechny bajty
efektivní adresa	effective address	adresa, na kterou instrukce nakonec sáhne

Česky	Anglicky	Význam
fetch	fetch	úvodní kroky instrukce, ve kterých se načte její opkód a operandy
glyf	glyph	obrazová podoba jednoho znaku, 8 × 8 bodů
hranový	edge-triggered	reaguje na změnu stavu, ne na stav samotný
mikrokód	microcode	posloupnost řídicích signálů, která tvoří jednu instrukci
mikroslovo	microword	jeden T-stav mikrokódu zakódovaný do bitů
nultá stránka	zero page	prvních 256 bajtů paměti; rychlý přístup a jediné místo pro ukazatele
obálka	envelope	křivka hlasitosti tónu v čase (ADSR)
opkód	opcode	bajt, který určuje, kterou instrukci procesor provede
pevná řádová čárka	fixed point	zápis desetinných čísel v celočíselných registrech
podmíněný skok	conditional jump	skok provedený jen při splnění podmínky nad příznaky
příznak	flag	jednobitový výsledek operace — C, Z, N, V, I
přerušení	interrupt	událost, která přeruší běžící program a předá řízení obsluze
pseudoinstrukce	pseudo-opcode	instrukce vložená hardwarem, kterou nelze načíst z paměti

Česky	Anglicky	Význam
rámec	stack frame	data uložená na zásobník při volání nebo přerušení
řídicí signál	control signal	jedno hradlo v datové cestě; mikrokód je seznam sepnutých
scancode	scancode	kód fyzické klávesy — na úrovni ISA se nevyskytuje
sprite	sprite	samostatný obrazový objekt kreslený nad hrací plochou
T-stav	T-state	jeden takt hodin; nejmenší jednotka času
úrovňový	level-sensitive	ukazuje současný stav; čtení ho nemění
ukazatel	pointer	dvoubajtová adresa uložená v nulté stránce
vektor	vector	dvoubajtová adresa na konci paměti, odkud si procesor bere cíl skoku
výpůjčka	borrow	význam příznaku C po odečítání: „podteklo“
zásobník	stack	oblast paměti, do které se ukládá při volání a přerušení
zdrojová stránka	source page	stránka paměti, ze které displej čte obraz
žurnál	journal	záznam změn, který umožňuje návrat v čase

PŘÍLOHA J

Rejstřík instrukcí

Abecední rejstřík všech mnemonik včetně aliasů skoků, s číslem stránky, na které začíná jejich heslo ve slovníku.