Postavíme si Arkanoid

Hra v assembleru, etapa po etapě

Od prázdné obrazovky k YOU WIN! — celá hra, které rozumíš do posledního bajtu.

Jan Müller

2026

Obsah

Předmluva

Kniha „Počítač do dlaně“ končila případovou studií: podívali jsme se z výšky na hotovou hru a ukázali si, jak v ní spolupracuje všechno, co procesor SAP-3/16 umí. Byl to pohled z letadla — viděl jsi celou krajinu, ale žádný jednotlivý kámen. Tahle knížka jde na zem. Vezme jednu skutečnou, hratelnou hru a **postaví ji s tebou od nuly** — řádek po řádku, bajt po bajtu, až do chvíle, kdy se na obrazovce rozsvítí YOU WIN!.

Ta hra je **Arkanoid**: pálka, míček a sto osm cihel. Vybrali jsme ho schválně. Je dost malý, aby se celý vešel do jedné knížky — hotový program má 978 řádků a 2 361 bajtů — a přitom dost velký, aby v něm bylo všechno, co dělá hru hrou: herní smyčka tikající přesně 250krát za sekundu, hardwarové sprity klouzající pod rozlišení znakové mřížky, fyzika v pevné řádové čárce, blitter skládající obrazovku, tříhlasý zvuk s obálkami a klouzáním, skóre, životy, pauza i restart. Až knížku dočteš, nebude v tom programu jediný řádek, kterému bys nerozuměl — a co je důležitější, budeš umět napsat hru vlastní.

Co potřebuješ

Tahle knížka je pokračování. Předpokládá, že znáš stroj SAP-3/16 zhruba v rozsahu „Počítače do dlaně“: co jsou registry **A**, **B**, **X** a **Y**, jak fungují příznaky a podmíněné skoky, co dělá zásobník s `CALL` a `RET`, jak se čte klávesnice a gamepad, a že zařízení jako zvukový čip nebo obrazovka jsou jen adresy v paměti. Kde stavíme na něčem konkrétním, odkážeme na příslušnou kapitolu první knihy — a co první kniha nepokrývá (a pár takových věcí bude), vysvětlíme tady od začátku.

Dál potřebuješ jen emulátor. Ten běží v prohlížeči na adrese **sap-3.com** — program vložíš do editoru, spustíš a hraješ. Kdo má repozitář emulátoru u sebe, může každou ukázku spustit i z příkazové řádky: `npm run asm <soubor.asm>`.

Jak je knížka postavená

Hru nestavíme „shora“ výkladem hotového kódu, ale **po etapách** — tak, jak by ji psal skutečný programátor. Každá kapitola přidá jednu vrstvu: nejdřív tikající srdce, pak obrazovku, pak pohyb, pravidla a nakonec zvuk. A každá etapa je **úplný, spustitelný program**: v adresáři `book/arkanoid/examples/` najdeš jedenáct souborů `e01` až `e11` a kapitola vždy říká, který z nich právě stavíš. Pozná se to podle rámečku na začátku kapitoly:

> **MILNÍK ETAPY — `eNN-nazev.asm`**
>
> Co program v této etapě umí a co uvidíš, když ho spustíš.

Poslední milník, `e11-arkanoid.asm`, je hotová hra — instrukce po instrukci totožná s tou, kterou najdeš v emulátoru mezi ukázkami. Nic v této knížce není opsané z hlavy: každý výpis pochází ze skutečného souboru a každé chování je ověřené spuštěním.

Některé etapy obsahují kousek kódu, který v hotové hře nebude — blikání okrajem, provizorní konec, míček, který se po propadnutí prostě podává znovu. Takovému kódu říkáme **lešení**: postaví se, poslouží a zase se strhne. I to k poctivému stavění patří; vždycky přesně uvidíš, kdy lešení stavíme a kdy ho bouráme.

V textu potkáš stejné tři druhy poznámek jako v první knize — *poznámku* pro souvislosti, *tip* pro programátorskou praxi a *pozor* pro zrádná místa.

Tak si nasaď helmu. Jdeme na stavbu.

ČÁST I

Než padne první cihla

KAPITOLA 1

Hra na stole

Než napíšeme první instrukci, musíme vědět, co vlastně stavíme. Tahle kapitola rozloží Arkanoid na stůl jako hodinky: podíváme se na pravidla, rozřežeme hru na subsystémy a ukážeme si, jak takový program vypadá zvenčí — jak se čte a v jakém pořadí je poskládaný. Ještě se nepíše; zato se hodně plánuje. Přesně jako na skutečné stavbě.

Pravidla

Arkanoid (v původní podobě z roku 1976 známý jako Breakout) je jedna z nejstarších videoher vůbec — a pořád jedna z nejlepších škol herního programování. Pravidla se vejdou do čtyř vět:

Dole na obrazovce jezdí **pálka**, řídíš ji doleva a doprava. Nad ní poletuje **míček**, který se odráží od stěn, od pálky a od **cihel** naskládaných v horní části hřiště. Zasažená cihla zmizí a přičte bod; míček, který proletí pod pálkou, stojí jeden ze tří **životů**. Vyhraješ, když rozbiješ všech 108 cihel; prohraješ, když ti dojdou životy.

K tomu drobnosti, které dělají hru hrou: míček se od pálky neodráží vždy stejně — **místo dopadu určuje úhel**, takže zkušený hráč míří. Mezerník hru pozastaví. A o všem důležitém informuje řádek nahoře: SCORE a LIVES.

Rozklad na subsystémy

Programátor her nemyslí v instrukcích, ale v subsystémech. Náš Arkanoid jich má šest a každý dostane v knížce svou etapu (nebo dvě):

Subsystém	Co dělá	Etapa
čas	smyčka tikající 250krát za sekundu, stejně rychle na každém stroji	1
obrazovka	statická scéna: zdi, cihly, HUD — složí ji blitter	2–3
sprity	míček a pálka plující nad znakovou mřížkou	4
vstup a pohyb	gamepad, rozjezd a brzdění pálky, let míčku	5–7
pravidla	odrazy, rozbíjení cihel, skóre, životy, výhra a prohra	7–9
zvuk	tři hlasy: klik, boing, výbuch, zavrčení, fanfára	10–11

Tabulka 1. Šest subsystémů hry a etapy, ve kterých je postavíme.

Všimni si, že to je skoro stejný seznam jako u kterékoli jiné akční hry. Vyměň cihly za vetřelce a pálku za kanón a máš Space Invaders; přidej scrollování a máš Cave Runner z první knihy. Kdo umí postavit Arkanoid, umí postavit osmibitovou hru.

Jak se čte zdroják

Hotová hra je jeden soubor o 978 řádcích. To zní hrozivě, ale assemblerové programy mají ustálenou architekturu — a když ji znáš, čte se soubor jako kniha s obsahem. Náš zdroják má pět pater, shora dolů:

1. **Slovník adres.** Desítky řádků `EQU`, které dávají jména všem adresám a konstantám: `VID_BORDER` místo `0xFF54`, `BRICKS_N` místo `108`. Procesoru je to jedno — `EQU` nevytváří žádný kód — ale člověku to mění šifru v text. Až uvidíš `STA VID_BORDER`, nebudeš muset nic hledat.
2. **Makra.** Dva pomocníci (`BFILL`, `BCOPY`), kteří sbalí obsluhu blitteru do jednoho řádku.
3. **Init.** Kód, který jednorázově postaví svět: zapne obraz, vyčistí paměť, namaluje zdi a cihly, připraví sprity. Běží jednou po startu.
4. **Hlavní smyčka a podprogramy.** Srdce hry — smyčka, která se opakuje každý snímek, a pod ní podprogramy, které volá: `wait_frame`, `ball_step`, `print_score`, zvukové efekty…
5. **Data.** Úplně dole leží bajty, které nejsou kód: tvary vlastních znaků, texty nápisů, vzory spritů.

> **Tip**
>
> Tohle pořadí není náhoda ani konvence pro parádu. Slovník musí být první, aby jména platila ve zbytku souboru; data jsou poslední, aby se procesor nepokusil „spustit“ je cestou. Stejné patrování uvidíš skoro v každém assemblerovém programu — od her po operační systémy.

Kolik toho bude

Na závěr trocha čísel, ať víš, do čeho jdeš. Hotový Arkanoid má 2 361 bajtů kódu a dat — méně než tahle kapitola v sazbě. Poběží 250 snímků za sekundu a v každém snímku stihne přečíst vstup, spočítat fyziku míčku, ohlídat kolize a odbavit zvuk. Ve špičce (rozbitá cihla se skóre a zvukem) to je pár set instrukcí na snímek — uvidíš, že rozpočet je překvapivě volný, právě proto, že těžkou práci s obrazem odvádí hardware.

V příští kapitole položíme základ, na kterém stojí všechno ostatní: čas.

KAPITOLA 2

Kostra: smyčka a čas

> **MILNÍK ETAPY — `e01-kostra.asm`**
>
> Prázdná obrazovka s fialovým okrajem, který každou půlsekundu blikne domodra. Nevypadá to jako hra — ale uvnitř už tiká srdce, které ponese všechno ostatní: smyčka běžící přesně 250krát za sekundu, na jakémkoli stroji.

Každá akční hra na světě má uvnitř totéž: **herní smyčku**. Jeden průchod smyčkou = jeden snímek, a v každém snímku se stane totéž, pořád dokola — přečti vstup, posuň svět, překresli obraz. U našeho Arkanoidu bude smyčka nakonec vypadat takhle:

```
opakuj každý snímek:
    1. počkej, až odtiká čas snímku     (wait_frame)
    2. odbav zvuk                       (snd_frame)
    3. přečti gamepad, vyřiď pauzu
    4. posuň pálku podle vstupu
    5. posuň míček, vyřeš odrazy        (ball_step)
    6. překresli sprity
```

Body 2 až 6 teprve přijdou. Dnes postavíme bod 1 — a to je paradoxně ten nejzáludnější.

Proč se nepočítá v taktech

První nápad zní: „snímek = tolik a tolik instrukcí, prostě nechám smyčku běžet.“ Jenže pak hra poběží **rychlostí procesoru**. Na stroji taktovaném na 100 kHz se míček poplazí, na 2 MHz proletí obrazovkou dřív, než stačíš mrknout — a když si hráč přepne rychlost emulátoru uprostřed hry, změní se i rychlost hry. Přesně tohle potkalo spoustu skutečných her z osmdesátých let: na rychlejším počítači byly nehratelné.

Druhý nápad z první knihy — počítat takty přes čítač (kapitola 16) — má tutéž vadu: takty nejsou čas. Potřebujeme hodiny, které měří **skutečné milisekundy** bez ohledu na takt. A stroj je má.

CLK_MS: milisekundové hodiny

Registrový pár `CLK_MS` (`0xFF3A` / `0xFF3B`) je šestnáctibitový čítač, který tiká jednou za **reálnou milisekundu** — při každém taktu procesoru se hardware podívá, kolik T-

-stavů odpovídá milisekundě při současné rychlosti hodin, a podle toho čítač krokuje. Zrychlíš stroj, čítač tiká po více taktech; zpomalíš, po méně. Výsledek je vždycky stejný: 1 000 tiků za sekundu.

Čte se stejným trikem jako ostatní vícebajtové čítače (vzpomeň na poznámku v kapitole 16 první knihy): **přečtení dolního bajtu zamkne oba bajty** do stínové kopie, ze které pak přečteš horní. Hodnota se ti tak nikdy „nerozjede“ uprostřed čtení.

wait_frame: krokovač snímků

Teď ta hlavní myšlenka. Zvolíme délku snímku — u nás `FRAME_MS EQU 4`, tedy 4 milisekundy, 250 snímků za sekundu (proč tak rychle, vysvětlíme u spritů v kapitole 5) — a na začátku každého průchodu smyčkou počkáme, až ty 4 milisekundy od minulého snímku **reálně uplynou**. Tady je celý podprogram z milníku:

```
wait_frame:
.spin:  LDA CLK_MS_LO           ; uplynulé ms = CLK_MS - ANCHOR
(čtení LO zamyká)
        SUB ANCHOR
        STA EL
        LDA CLK_MS_HI
        SBC ANCHOR+1
        JNZ .done               ; uplynulo >= 256 ms — dávno přes
rozpočet
        LDA EL
        CMP #FRAME_MS           ; C = 1, dokud uplynulé < FRAME_MS
        JC .spin
.done:  LDA CLK_MS_LO           ; nová kotva v okamžiku uvolnění —
snímek
        STA ANCHOR              ;   přes rozpočet nenese žádný
dluh
        LDA CLK_MS_HI
        STA ANCHOR+1
        RET
```

Rozeber si ho po vrstvách. V nulté stránce leží **kotva** `ANCHOR` — šestnáctibitový snímek hodnoty `CLK_MS` z okamžiku, kdy skončil minulý snímek. Smyčka `.spin` pořád dokola počítá `uplynulé = CLK_MS - ANCHOR`: klasické šestnáctibitové odečítání `SUB` / `SBC`, jak ho známe z kapitoly 6 první knihy. Pak dvě podmínky. Když je horní bajt rozdílu nenulový, uplynulo přes 256 ms — to se stane třeba po pauze v ladicím režimu — a čekání okamžitě končí. Jinak porovnáme dolní bajt s rozpočtem: `CMP #FRAME_MS` nastaví

příznak přenosu, dokud je uplynulý čas **menší** (připomeň si: `CMP` počítá výpůjčku), takže `JC .spin` drží smyčku v běhu přesně do chvíle, kdy čtvrtá milisekunda odtiká.

A nakonec detail, který dělá z dobrého krokovače výborný: kotva se **znovu přečte z hodin** až v okamžiku uvolnění, místo aby se jen posunula o 4. Snímek, který se výjimečně protáhl přes rozpočet, tak nezanechá „dluh“, který by další snímky honily. Hra po zádrhelu prostě plynule pokračuje.

> **Poznámka**
>
> Tenhle vzor se jmenuje **ukotvené čekání** (elapsed-since-anchor) a potkáš ho všude, kde se softwarově drží tempo: v herních enginech, přehrávačích, komunikačních protokolech. Naivní varianta „kotva += 4“ se po každém zpoždění rozbíhá do dohánění; varianta s novou kotvou odpouští. Které chování chceš, je návrhové rozhodnutí — hra chce odpouštět.

Init: zapnout světla

Před smyčkou proběhne jednorázová příprava. Zatím je nejkratší v celé knížce:

```
start:  DI                      ; vstup budeme číst pollingem,
přerušení nechceme
        LDA #0x01
        STA VID_CTRL            ; obraz zapnout, režim 0 (text
40x25)
        LDA #4
        STA VID_BORDER          ; fialový rám skříně automatu
        LDA #25
        STA VID_CURY            ; kurzor odstavit z obrazu (řádek
25 z 0-24)
```

Tři maličkosti stojí za slovo. `DI` hned na prvním řádku je **rozhodnutí, ne opomenutí**: celá hra bude číst vstup pollingem z gamepadu (kapitola 6), takže přerušení nepotřebuje — a co nepotřebuješ, to vypni. Fialový okraj je poklona hernímu automatu: skříň kolem obrazovky. A hardwarový kurzor by nám blikal doprostřed hřiště, tak ho parkujeme na řádek 25 — první řádek, který už na obrazovce není.

Lešení: důkaz, že srdce bije

Prázdná obrazovka se smyčkou, která „určitě běží“, není žádný důkaz. Proto do první etapy stavíme kousek lešení — počítadlo snímků, které každých 125 průchodů přepne barvu okraje:

```
main:   CALL wait_frame

        ; --- lešení: každých BLINK_FR snímků přepni barvu okraje
---
        DEC BLINK_CNT           ; DEC v paměti nastavuje Z
        JNZ main
        LDA #BLINK_FR
        STA BLINK_CNT
        LDA BORDER_NOW
        XOR #0x02               ; 4 (fialová) <-> 6 (modrá)
        STA BORDER_NOW
        STA VID_BORDER
        JMP main
```

125 snímků × 4 ms = přesně půl sekundy na přepnutí. Spusť milník a dívej se: okraj bliká v sekundovém rytmu jako metronom — a bliká **stejně rychle**, ať emulátoru nastavíš 100 kHz nebo 2 MHz. Zkus to; je to nejlepší způsob, jak uvěřit, že `wait_frame` opravdu měří čas, a ne takty.

Dvě řemeslné drobnosti na závěr. `DEC BLINK_CNT` pracuje přímo s pamětí a rovnou nastavuje příznak nuly, takže odpočítadlo nepotřebuje žádný registr (vzor „počítej dolů“ z kapitoly 13 první knihy). A přepínání barvy přes `XOR #0x02` využívá toho, že fialová (4) a modrá (6) se liší jediným bitem — místo porovnávání a dvou větví stačí jedna instrukce.

V hotové hře po tomhle kódu nezbude ani stopa — proměnné `BLINK_CNT` a `BORDER_NOW` v příští etapě zbouráme i se smyčkou. Kostra ale zůstane: `DI`, zapnutý obraz, kotva a `wait_frame` přežijí beze změny až do finále. Teď na tu kostru pověsíme obrazovku.

ČÁST II

Obrazovka patří blitteru

KAPITOLA 3

Blitter a makra

> **MILNÍK ETAPY — `e02-obrazovka.asm`**
>
> Statická scéna: čistá plocha, šedá zeď přes celý druhý řádek a azurový nápis SCORE 000 … LIVES 3 nahoře. Blikací lešení z minulé etapy je stržené; celou obrazovku poskládal blitter během startu.

Obrazovka má 40 × 25 = 1 000 znakových buněk a pod nimi stejně velkou barevnou rovinu. Postavit scénu znamená popsat **dva tisíce bajtů** paměti — a to je první chvíle, kdy si spočítáme účet, než začneme psát. Smyčka „načti, zapiš, posuň, porovnej“ stojí kolem pěti instrukcí na bajt, tedy zhruba deset tisíc instrukcí jen za úklid obrazovky. Při startu hry by nás to trápit nemuselo… ale stejný úklid budeme chtít při každém restartu, a hlavně: proč psát smyčku, když je v počítači zabudovaný specialista?

Blitter v pěti řádcích

Blitter jsme poznali v kapitole 18 první knihy — hardwarový stěhovák, kterému řekneš **co** a on se postará **jak**. Ovládá se přes osm registrů v oblasti `0xFF72` – `0xFF79`: dvojice zdroj/cíl/délka, výplňový bajt a řídicí registr `BLT_CTRL`, kde zápis `1` spustí kopii a zápis `2` výplň. Celý přesun proběhne **uvnitř toho jediného zápisu** — z pohledu programu za nula taktů navíc.

Vymazat znakovou rovinu tedy znamená: nastav cíl `0x8000`, délku `1000`, výplňový bajt mezeru, zapiš `2`. Deset obyčejných `LDA` / `STA`. Jenže těch výplní a kopií budeme v celé hře dělat přes dvacet — a tady přichází ke slovu druhý nástroj z první knihy.

Makra BFILL a BCOPY

Kapitola 12 první knihy nás naučila makra: pojmenované kusy kódu, které preprocesor rozbalí na místě použití. Přesně to tu potřebujeme — obsluha blitteru je pokaždé stejná, liší se jen tři čísla:

```
%macro BFILL 3              ; BFILL cíl, délka, bajt
        LDA #<%1
        STA BLT_DST_LO
        LDA #>%1
```

```
        STA BLT_DST_HI
        LDA #<%2
        STA BLT_LEN_LO
        LDA #>%2
        STA BLT_LEN_HI
        LDA #%3
        STA BLT_FILL
        LDA #2
        STA BLT_CTRL
%endmacro
```

Operátory `#<` a `#>` vyříznou z šestnáctibitové hodnoty dolní a horní bajt — makro tak přijme rovnou adresu (`BFILL CHARS, 1000, CH_SPACE`) a samo ji rozloží do dvou registrů. `BCOPY` vypadá stejně, jen navíc nastavuje zdroj a spouští příkazem `1`.

> **Poznámka**
>
> Proč makro, a ne podprogram? Podprogram by musel dostat šest bajtů parametrů — přes nultou stránku nebo přes registry, kterých je málo — a stál by `CALL` / `RET` navíc. Makro parametry „zadrátuje“ přímo do instrukcí. Cena je velikost (každé použití = 24 bajtů kódu), ale u dvaceti použití je to pořád levnější než aparát na předávání parametrů. Řemeslné pravidlo: krátké a časté s konstantními parametry -> makro; dlouhé nebo s parametry za běhu -> podprogram.

Stavíme scénu

S makry je init čitelný jako seznam úkolů pro stěhováky:

```
        ; Statická obrazovka: vyčistit obě roviny, pak namalovat
pevné řádky.
        BFILL CHARS, 1000, CH_SPACE
        BFILL COLORS, 1000, 0x00
        BFILL CROW0, 40, INK_HUD    ; barva HUD přežije přepisy
číslic
        BFILL ROW1, 40, CH_WALL     ; znaky horní zdi + jejich
barva
        BFILL CROW1, 40, INK_WALL
```

Pět příkazů, 2 120 přenesených bajtů. Za pozornost stojí třetí řádek — a s ním trik, který hra použije ještě mnohokrát. Barevná rovina (`COLORS`, od `0x8C00`) je **nezávislá** na znakové: barva buňky zůstává, i když se znak v buňce přepíše. Nabarvíme tedy celý

nultý řádek azurovou **jednou při startu** — a od té chvíle může skóre přepisovat číslice, jak chce, vždycky budou azurové. Nikde ve hře pak neuvidíš kód, který by barvil text HUD; není potřeba.

Vlastní znaky: zeď a cihly

Vestavěné písmo zeď ani cihlu nemá. Naštěstí znaková sada žije v **glyph RAM** — od `0x8400` leží pro každý z 256 znaků osm bajtů, řádek po řádku, bit = pixel (kapitola 18 první knihy). Nakreslíme si tedy tři vlastní znaky a nahrajeme je na pozice 3, 4 a 5:

```
glyphs: ; znak 0x03 — zeď (plný blok)
        DATA 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF
        ; znak 0x04 — cihla, levá půlka (1 px spáry vlevo)
        DATA 0x00, 0x7F, 0x7F, 0x7F, 0x7F, 0x7F, 0x7F, 0x00
        ; znak 0x05 — cihla, pravá půlka (1 px spáry vpravo)
        DATA 0x00, 0xFE, 0xFE, 0xFE, 0xFE, 0xFE, 0xFE, 0x00
```

Přečti si ty bajty jako obrázky: `0xFF` je plná osmice pixelů; `0x7F` (`0111 1111`) má zhasnutý **nejlevější** bit a `0xFE` (`1111 1110`) **nejpravější**; nulové první a poslední řádky dělají vodorovné spáry. Jedna cihla hry bude **dvojice buněk** `[]` — levá půlka se spárou vlevo, pravá se spárou vpravo — takže mezi cihlami vzniknou mezery jako v maltě, ačkoli buňky sedí těsně vedle sebe.

Do glyph RAM je dopraví jediné `BCOPY glyphs, GLYPH3, 24` — tři znaky po osmi bajtech, zkopírované přímo z datového patra programu. A hned za ním putují na obrazovku texty HUD, uložené jako ASCII bajty:

```
        BCOPY glyphs, GLYPH3, 24
        BCOPY hud_score, HUD_SCORE, 9
        BCOPY hud_lives, HUD_LIVES, 7
```

> **Pozor**
>
> Glyfy nahráváme **za běhu programu**, ne jako součást obrazu paměti. Kdyby assembler zapsal bajty do glyph RAM už při překladu, emulátor by při startu viděl „nahranou znakovou sadu“ a potlačil vestavěné písmo — a nápisy HUD by se rozsypaly. Init s `BCOPY` je za tři instrukce pojistka, že se písmo a naše tři znaky vždycky potkají správně.

Zbytek obrazovky — boční zdi a cihlové pole — vyžaduje počítání adres, a to si zaslouží vlastní kapitolu. Smyčka `main` zatím zůstává prázdná: `CALL wait_frame` a skok zpátky. Kostra tiká, scéna stojí. Jde se zdít.

KAPITOLA 4

calc_addr a cihlové pole

> **MILNÍK ETAPY — `e03-hriste.asm`**
>
> Kompletní hřiště: šedé zdi kolem tří stran a šest barevných řádků cihel — červená, oranžová, žlutá, zelená, azurová, světle modrá. Sto osm cihel čeká na míček, který ještě neexistuje.

Blitter umí souvislé bloky. Jenže boční zdi jsou **sloupce** — buňka úplně vlevo a úplně vpravo na každém řádku — a ty v paměti souvislé nejsou: mezi levou a pravou zdí jednoho řádku leží 38 bajtů hřiště. Tady musí nastoupit procesor a s ním dovednost, kterou hra použije na tisíc místech: **spočítat adresu buňky**.

calc_addr: od souřadnic k adrese

Vzorec známe z první knihy: adresa = `0x8000 + y*40 + x`. Ale násobení na tomhle stroji má záludnost, kvůli které si podprogram rozebereme instrukci po instrukci:

```
calc_addr:
        LDA CELL_Y
        TAB
        LDA #40
        MUL                     ; A = dolní bajt y*40, B = horní
bajt
        STA PTR
        TBA                     ; vyzvedni horní bajt, dokud ho B
ještě drží
        ADD #0x80               ; + báze VRAM (přenos nehrozí:
horní <= 3)
        STA PTR_HI
        LDA PTR
        ADD CELL_X              ; C = přenos do horního bajtu
        STA PTR
        JNC .done
        INC PTR_HI
.done:  RET
```

`MUL` vynásobí **A** × **B** a šestnáctibitový výsledek rozdělí: dolní bajt do **A**, horní do **B**. Pro `y*40` s `y` až 24 vyjde nejvýše 960, takže horní bajt je 0 až 3 — proto můžeme bez obav přičíst `0x80` (bázi obrazové paměti) bez kontroly přenosu.

Ta záludnost je v pořadí. Vzpomeň si na varování z kapitoly 13 první knihy („Pamatuj na registr B“): **každá dvouoperandová operace ALU si nejdřív natáhne svůj operand do B** — i nevinné `ADD #0x80`. Kdybychom horní bajt násobení nevyzvedli instrukcí `TBA` **hned**, první `ADD` by nám ho přepsalo a hodnota by byla nenávratně pryč. Tohle je nejčastější chyba, kterou v programech pro SAP-3/16 uvidíš — a `calc_addr` je učebnicová ukázka, jak se jí vyhnout: `MUL`, okamžitě `TBA`, teprve pak počítej dál.

Zbytek je šestnáctibitové přičtení osmibitového `x`: dolní bajt `ADD CELL_X` a případný přenos se do horního bajtu promítne podmíněným `INC PTR_HI` — instrukce, která nastavuje jen Z a N, takže nic nerozbije.

Boční zdi a trik +0x0C

Se `calc_addr` v ruce jsou boční zdi obyčejná smyčka přes řádky 2 až 23. Za pozornost stojí, jak se u každé buňky obslouží **obě** roviny — znak i barva:

```
.side:  LDA #0
        STA CELL_X
        CALL calc_addr
        LDA #CH_WALL
        STA (PTR)
        LDA PTR_HI
        ADD #0x0C
        STA PTR_HI
        LDA #INK_WALL
        STA (PTR)
```

Barevná rovina leží od `0x8C00`, znaková od `0x8000` — přesně `0x0C00` od sebe. Adresy se tedy liší **jen horním bajtem**, a tak se spočtený ukazatel na znak promění v ukazatel na jeho barvu jediným `ADD #0x0C` na `PTR_HI`. Žádné druhé volání `calc_addr`, žádné násobení. V dalším průchodu smyčky si `calc_addr` ukazatel stejně postaví znovu, takže „zkažený“ `PTR` nikomu nevadí.

> **Tip**

> Tenhle vzor si zapamatuj obecněji: když dvě datové struktury leží v pevném odstupu, ukazatel mezi nimi převádíš aritmetikou, ne novým výpočtem. Hra ho použije ještě jednou — u cihel, kterým při zásahu zůstává barva (kapitola 9).

Cihlové pole: napiš jednou, klonuj pětkrát

Zbývá 108 cihel: 6 řádků, v každém 18 cihel po dvou buňkách, sloupce 2 až 37. Napsat `STA` na 216 buněk by bylo k uzoufání — a tady se krásně rozdělí práce mezi procesor a blitter. Procesor postaví **jeden** řádek, blitter ho **naklonuje**.

První řádek stavíme ukazatelem s post-indexem `(zp),Y` (kapitola 14 první knihy): `PTR` míří na začátek řádku a **Y** šlape po buňkách:

```
        LDY #0
.bpair: LDA #CH_BRKL
        STA (PTR),Y
        INY
        LDA #CH_BRKR
        STA (PTR),Y
        INY
        CPY #36
        JC .bpair
```

Každý průchod zapíše **pár** — levou a pravou půlku cihly — takže `Y` skáče po dvou a smyčka běží, dokud `CPY #36` hlásí „ještě ne“: porovnání indexu nastavuje výpůjčku C, dokud je **Y** menší, a `JC` na tom staví podmínku smyčky, aniž by sáhlo na **A**.

A teď ta odměna. Řádky 4 až 8 jsou **bajt po bajtu stejné** jako řádek 3 — tak proč je stavět?

```
        BCOPY BRICK_ROW3, BRICK_ROW4, 36
        BCOPY BRICK_ROW3, BRICK_ROW5, 36
        BCOPY BRICK_ROW3, BRICK_ROW6, 36
        BCOPY BRICK_ROW3, BRICK_ROW7, 36
        BCOPY BRICK_ROW3, BRICK_ROW8, 36
```

Pět kopií, hotovo. Vzor „postav vzorek, rozmnož ho DMA“ je jeden z nejužitečnějších v celém osmibitovém programování — potkáš ho u dlaždicových map, tabulek i fontů.

Duha

Barvy cihel jsou čisté `BFILL` do barevné roviny — jedna výplň na řádek, šest řádků, šest barev z palety (příloha D první knihy):

```
        BFILL CBRICK_R3, 36, 2      ; červená
        BFILL CBRICK_R4, 36, 8      ; oranžová
        BFILL CBRICK_R5, 36, 7      ; žlutá
        BFILL CBRICK_R6, 36, 5      ; zelená
        BFILL CBRICK_R7, 36, 3      ; azurová
        BFILL CBRICK_R8, 36, 14     ; světle modrá
```

Duha je věrná předloze z automatu — a zároveň to je podruhé, kdy se nám vyplatí nezávislost barevné roviny: až míček začne cihly mazat, bude přepisovat **jen znaky**. Barva zůstane v buňce ležet a nikoho nebude stát jedinou instrukci.

Spusť milník. Hřiště stojí, svítí barvami — a je mrtvé jako kulisa před představením. V příští kapitole na scénu pustíme herce: sprity.

ČÁST III

Vše, co se hýbe, je sprite

KAPITOLA 5

Sprity: míček a pálka

> **MILNÍK ETAPY — `e04-sprity.asm`**
>
> Na hřišti se objeví herci: bílý kotouček míčku posazený na dvoudílné šedé pálce uprostřed spodního okraje. Zatím stojí jako sochy — ožijí v příští etapě.

Míček a pálka se budou hýbat každý snímek. Kdybychom je kreslili do znakové mřížky, znamenal by každý pohyb „smaž starou buňku, nakresli novou“ — a hlavně by pohyb **skákal po celých buňkách**, po osmi pixelech. Herní automat s trhaným míčkem by nikoho neuživil. Přesně pro tenhle problém má stroj **sprity** (kapitola 18 první knihy): malé obrázky, které hardware skládá **nad** obraz, aniž by se obrazu dotkl. Pohyb spritu = zápis dvou bajtů. Žádné mazání, žádné překreslování — a jemnější rozlišení, než má znaková mřížka.

Tlusté pixely: souřadnice spritů

Sprity žijí v souřadnicích **bitmapového režimu 1**: 160 × 100 bodů na celou obrazovku. Jeden takový bod budeme po zbytek knížky nazývat **tlustý pixel**. Převodní tabulka je jednoduchá a budeme ji potřebovat pořád:

Jednotka	Vztah
textová buňka	4 × 4 tlusté pixely
obrazovka	160 × 100 tlustých px (40 × 25 buněk)
převod buňka ⟶ tlusté px	vynásob 4, tedy dva posuvy `SHL`
převod tlusté px ⟶ buňka	vyděl 4, tedy dva posuvy `SHR`

Tabulka 2. Tlustý pixel = 1/4 textové buňky. Mezi oběma prostory se převádí dvěma posuvy.

Že je buňka **přesně** 4 × 4 tlusté pixely, není náhoda — je to důvod, proč hra může míchat světy: pálka a míček žijí ve spritovém prostoru, cihly ve znakové mřížce, a kolize mezi nimi je převod dvěma posuvy (uvidíš v kapitole 9).

K souřadnicím ještě jedna zvláštnost z první knihy: hardware k oběma osám přičítá **posun +16**. Hodnota 16 znamená „přilepený k okraji obrazu“; menší hodnoty sprite postupně vysouvají ven. Díky tomu se sprite umí schovat za kterýkoli okraj, aniž by

souřadnice potřebovala záporná čísla. V kódu to uvidíš jako věčné `ADD #16` těsně před zápisem do atributů.

Vzory: obrázek jako pole půlbajtů

Obrázky spritů — **vzory** — leží v obyčejné video paměti od `0xB000`, 128 bajtů na vzor: 16 řádků po 8 bajtech, v každém bajtu **dva pixely** (horní půlbajt levý, dolní pravý). Půlbajt je přímo číslo barvy z palety a **nula znamená průhledno**. Náš míček je kotouček 4 × 4 v levém horním rohu vzoru 0:

```
pat_ball:  ; vzor 0, řádky 0-3 — kotouček 4x4, nasvícený zleva
shora
        DATA 0x01, 0x10, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00
        DATA 0x11, 0x1F, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00
        DATA 0x1F, 0xFF, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00
        DATA 0x0F, 0xF0, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00
```

Přečti si první dva bajty prvního řádku po půlbajtech: `0, 1, 1, 0` — dva bílé pixely (barva 1) s průhlednými rohy. Poslední řádek `0, F, F, 0` zrcadlí totéž světle šedou (15). Diagonála z jedniček a patnáctek dělá dojem kuličky nasvícené zleva shora. Kreslení spritů je přesně tohle: obrázek přepsaný do šestnáctkových číslic.

Pálka ze dvou spritů

Sprite má 16 tlustých pixelů na šířku — jenže pálka potřebuje 24 (šest buněk). Řešení je prosté: pálka jsou **dva sprity vedle sebe**. Sprite 1 nese levých 16 pixelů (se zakulaceným levým koncem), sprite 2 pravých 8 (se zakulaceným pravým); zbytek vzoru 2 je průhledný. Kdo se na hru dívá, vidí jednu lištu — kdo čte atributovou tabulku, vidí dva záznamy, jejichž X se vždycky liší přesně o 16.

Vlastní zapojení v initu je už jen logistika, samozřejmě přes blitter:

```
        BFILL SPR_PAT0, 384, 0x00
        BFILL SPR_ATTR, 32, 0x00
        BCOPY pat_ball, SPR_PAT0, 32
        BCOPY pat_pad_l, SPR_PAT1R1, 16
        BCOPY pat_pad_r, SPR_PAT2R1, 16
        LDA #1
        STA SPR1_PAT
        LDA #2
        STA SPR2_PAT
        LDA #PAD_SPR_Y          ; pálka svůj řádek nikdy neopouští
```

```
        STA SPR1_Y
        STA SPR2_Y
```

Nejdřív se vymažou vzory 0–2 i celá atributová tabulka (nula = průhledno a zároveň „vzor 0, souřadnice 0“ — čistý stůl). Pak se nakopírují jen ty řádky vzorů, kde opravdu něco je: 32 bajtů kotoučku, 2 × 16 bajtů pálky. Spritům 1 a 2 se přidělí vzory a **ypsilonová** souřadnice `PAD_SPR_Y` — pálka jezdí jen vodorovně, takže její Y se nastaví jednou za hru a víckrát se ho nikdo nedotkne.

Nakonec rozsvítíme herce a zaparkujeme je na startovní značky:

```
        LDA #0b111
        STA SPR_ENABLE          ; míček + obě půlky pálky zapnout
```

Parkování je v této etapě natvrdo (pálka na `PAD_START` = 68, střed obrazovky; míček o deset tlustých pixelů dál, aby seděl na jejím středu) — poslední lešení s konstantami, které příští etapa nahradí živou proměnnou.

> **Poznámka**
>
> V `SPR_ENABLE` má každý sprite svůj bit a při překryvu vyhrává **nižší číslo**. Míček je schválně sprite 0: až poletí přes pálku, chceme ho vidět celý. Pořadí spritů je návrhové rozhodnutí, ne detail — rozmysli si ho dřív, než začneš kreslit.

Spusť milník: scéna je kompletní, herci na značkách. Zbývá maličkost — aby poslouchali.

KAPITOLA 6

Pálka pod prsty

> MILNÍK ETAPY — `e05-palka.asm`
>
> První živý vstup: klávesami A/D (nebo šipkami) jezdí pálka — s míčkem na hřbetě — po spodním okraji. Rozjíždí se plynule, zastavuje ostře a od zdí se nenechá vytlačit.

Tahle kapitola je o **pocitu**. Kód, který jen posouvá pálku o konstantu doleva a doprava, je napsaný za minutu — a hraje se s ním mizerně: buď je pálka lenivá, nebo přestřeluje. Herní programátoři tomu říkají *game feel* a věnují mu překvapivé množství práce. Náš recept má tři ingredience: úrovňový gamepad, rozjezdovou rampu a zlomkové bajty.

Gamepad: číst stav, ne události

Klávesnice hlásí „byla stisknuta klávesa“ — jednou, a pak na ni zapomene. Hra ale potřebuje vědět „je klávesa **pořád** držená?“, šedesátkrát za sekundu. Od toho má stroj gamepad (kapitola 16 první knihy): `PAD1` a `PAD2` jsou **bitmapy právě držených tlačítek**, jeden bajt na hráče, a čtení je nemaže. Hra si na začátku každého snímku pořídí snímek vstupu:

```
        LDA PAD1
        OR PAD2
        STA PAD_NOW
```

To `OR` je malá zdvořilost: `PAD1` mapuje WASD, `PAD2` šipky, a jejich sloučením hra slyší na obojí — hráč si nemusí nic nastavovat. Test směru je pak maska: `AND #0x04` pro Vlevo, `AND #0x08` pro Vpravo.

Rampa: pad_accel

Držená klávesa nemá pálku rozjet naplno hned — příjemnější je krátký rozjezd. Rychlost pálky žije v proměnné `PAD_SPD` a rampu obstará miniaturní podprogram:

```
pad_accel:
        LDA PAD_SPD
        CMP #PAD_VMAX
```

```
        JNC .done               ; už na maximu (A >= VMAX)
        ADD #PAD_ACCEL
        CMP #PAD_VMAX
        JC .set
        LDA #PAD_VMAX           ; poslední krok dosedne přesně na
VMAX
.set:   STA PAD_SPD
.done:  RET
```

Každý snímek s drženou klávesou přičte `PAD_ACCEL` (0x10), dokud nenarazí na strop `PAD_VMAX` (0xC0) — s pojistkou, aby poslední krok dosedl **přesně** na maximum a nepřestřelil. Při puštění klávesy hlavní smyčka rychlost prostě vynuluje: `LDA #0, STA PAD_SPD`. Rozjezd plynulý, zastavení ostré — pálka nesmí „plavat“, protože hráč zastavuje pod padajícím míčkem a každý pixel klouzání navíc je ztracený život.

A teď to hlavní: **v jakých jednotkách** je vlastně rychlost 0xC0?

Zlomkové bajty: první setkání s Q8.8

Pálka se pohybuje 250krát za sekundu. Kdyby nejmenší krok byl celý tlustý pixel, byla by nejpomalejší možná rychlost 250 px/s — jedenapůlnásobek šířky obrazovky za sekundu! Potřebujeme rychlosti **menší než jeden pixel na snímek**. Celá čísla na to nestačí… a přesto si vystačíme s celými čísly.

Trik se jmenuje **pevná řádová čárka**, formát Q8.8: hodnotu vedeme ve dvou bajtech, horní bajt jsou celé pixely, dolní bajt **dvěstěpadesátšestiny** pixelu. Poloha pálky je pár `PAD_POS` (celé tlusté pixely) a `PAD_POS_LO` (zlomek); rychlost `PAD_SPD` je **jen zlomkový bajt** — 0xC0 znamená 192/256 = 0,75 tlustého pixelu na snímek, tedy asi 187 px/s. Krok doleva je pak obyčejné šestnáctibitové odečtení, jak ho známe z kapitoly 6 první knihy:

```
.left:  CALL pad_accel          ; rampuj PAD_SPD směrem k PAD_VMAX
        LDA PAD_POS_LO
        SUB PAD_SPD             ; krok doleva aktuální rychlostí
(Q8.8)
        STA PAD_POS_LO
        LDA PAD_POS
        SBC #0
        STA PAD_POS
```

`SUB` odečte zlomky, `SBC #0` propíše případnou výpůjčku do celých pixelů. Nic víc pevná čárka není: **dvoubajtové číslo, u kterého jsme se rozhodli, že dolní bajt**

jsou zlomky. Sčítání a odčítání fungují beze změny — o interpretaci vědí jen naše hlavy a komentáře.

> **Poznámka**
>
> Zápis Q8.8 čti „8 bitů celé části, 8 bitů zlomku“. První kniha se zlomkům vyhnula; ve skutečných hrách jsou všudypřítomné, protože plovoucí čárka na osmibitech prakticky neexistuje — je pomalá a bez hardwarové podpory. Q8.8 dává rozlišení 1/256 pixelu a stojí jediný bajt navíc. V příští kapitole na něm postavíme celou fyziku míčku, včetně záporných rychlostí.

Mantinely a kreslení

Po kroku se poloha přišpendlí k mezím `PAD_PMIN` (4 — přilepená k levé zdi) a `PAD_PMAX` (132 — pravý konec pálky přilepený k pravé zdi); při dojezdu na mez se vynuluje i zlomek, aby pálka „seděla“ na zdi bez chvění. A nakonec se poloha propíše do spritů:

```
.pdraw: LDA PAD_POS             ; oba sprity pálky sledují PAD_POS
        ADD #16                 ; hardwarový posun obrazu +16
        STA SPR1_X
        ADD #16                 ; pravá půlka sedí o 16 tlustých
px dál
        STA SPR2_X
```

Dvě `ADD #16` za sebou jsou dvě různé věci: první je hardwarový posun souřadnic (kapitola 5), druhé posouvá pravou půlku pálky o šířku spritu. A všimni si, co se **nekreslí**: zlomek. Sprite dostává jen celé pixely z `PAD_POS`; `PAD_POS_LO` žije čistě v aritmetice. Obraz se tak pohybuje po celých tlustých pixelech, ale **rychlost** má jemnost 1/256 — přesně to rozdělení práce, kvůli kterému Q8.8 existuje.

Míček zatím jezdí přilepený na středu pálky (`PAD_POS + 10`, tedy střed lišty minus polovina kotoučku) — poslední ochutnávka před tím, než dostane vlastní život. Spusť milník a chvíli si jen tak jezdi: rozjezd, ostrá brzda, doraz na zeď. Pocit je hotový. Teď fyzika.

KAPITOLA 7

Míček letí: fyzika Q8.8

> **MILNÍK ETAPY — `e06-micek.asm`**
>
> Míček ožívá: chvíli jede na pálce, pak vystřelí náhodným směrem vzhůru a odráží se od stropu i bočních zdí. Cihlami zatím prolétá jako duch a pálka ho neumí chytit — když propadne dolů, podává se znovu (lešení).

V minulé kapitole jsme si Q8.8 vyzkoušeli na rychlosti pálky. Míček po nás chce plnou verzi: pohyb ve **dvou osách** a hlavně rychlosti, které umí být **záporné**. Míček letící doleva má vodorovnou rychlost −0,125 pixelu na snímek — a my ji musíme uložit do bajtů, které záporná čísla neznají.

Znaménko zadarmo: dvojkový doplněk

Odpověď známe z první knihy (kapitola 1): **dvojkový doplněk**. Šestnáctibitová rychlost `VX_HI:VX_LO` je znaménkové číslo — horní bit horního bajtu nese znaménko. Kouzlo dvojkového doplňku je, že sčítačka o něm nemusí vědět: `poloha + rychlost` je pořád totéž `ADD` / `ADC`, ať je rychlost kladná nebo záporná. Přetečení a výpůjčky to zařídí samy.

Konstanty rychlostí si hra pojmenovala v hlavičce:

Konstanta	Q8.8	px/s	Význam
`VY_MAG`	`0x0030`	~47	svislá rychlost — vždy stejná, mění jen znaménko
`VX_LAUNCH`	`0x0020`	~31	boční rychlost výkopu po podání
`-VY_MAG`	`0xFFD0`	~−47	tatáž svislá rychlost vzhůru, dvojkový doplněk

Tabulka 3. Rychlosti míčku. Přepočet: hodnota/256 tlustých px na snímek, × 250 snímků/s.

Číslo `0xFFD0` si ověř ručně: doplněk `0x0030` je `0xFFFF - 0x0030 + 1 = 0xFFD0`. V kódu se záporné hodnoty často zapisují přímo (`LDA #0xD0` do dolního bajtu, `LDA #0xFF` do horního) — a když je potřeba otočit znaménko **za běhu**, použije se vzorec `0 − x` v šestnácti bitech:

```
        LDA #0                  ; ...a otoč VX (16bitový dvojkový
doplněk)
```

```
        SUB VX_LO
        STA VX_LO
        LDA #0
        SBC VX_HI
        STA VX_HI
```

Tohle je **odraz od svislé zdi**: rychlost v ose X změní znaménko, velikost zůstane. Fyzika kulečníku ve čtyřech instrukcích.

serve: míček na pálce

Po každém podání míček neletí hned — sedí na pálce a čeká, aby se hráč stačil nadechnout. Podprogram `serve` ho tam posadí a natáhne odpočet:

```
serve:  LDA PAD_POS
        ADD #10                 ; PAD_MID - polovina 4px míčku
        STA BX
        LDA #0
        STA BX_LO
        STA BY_LO
        LDA #BY_PAD
        STA BY
        LDA #SERVE_PAUSE
        STA SERVE_DLY
        RET
```

`SERVE_PAUSE` je 180 snímků — při 250 fps asi 0,72 sekundy. Hlavní smyčka pak každý snímek, dokud `SERVE_DLY` nedojde na nulu, opisuje X míčku z polohy pálky (míček na ní „jede“, přesně jako v minulé etapě) a odpočet snižuje. V nule přijde **výkop**:

```
        LDA #0xD0               ; VY := -VY_MAG (16bitový dvojkový
doplněk)
        STA VY_LO
        LDA #0xFF
        STA VY_HI
        RND
        AND #0x01
        JZ .lright
```

Svisle vždycky vzhůru; vodorovně rozhodne nejnižší bit náhodného čísla — doleva, nebo doprava, `VX_LAUNCH` s příslušným znaménkem. Instrukce `RND` je tatáž, kterou první kniha používala na házení kostkou; tady dává podání špetku nepředvídatelnosti.

A protože emulátor umí generátor **nasadit na pevné semínko** (`--seed`), je i tahle náhoda při ladění reprodukovatelná: stejné semínko, stejná hra.

ball_step: pohyb a odrazy

Srdce fyziky je podprogram `ball_step`, volaný jednou za snímek. Jeho osa X v plné kráse:

```
ball_step:
        ; --- osa X: 16bitový součet, odraz od bočních zdí ---
        LDA BX_LO
        ADD VX_LO
        STA BX_LO
        LDA BX
        ADC VX_HI
        STA BX
        CMP #BX_MIN
        JNC .xhigh              ; na levé zdi, nebo napravo od ní
        LDA #BX_MIN
        JMP .xwall
.xhigh: CMP #BX_MAX
        JC .xok                 ; ostře uvnitř
        JZ .xok                 ; přesně na zdi — ještě dobré
        LDA #BX_MAX
.xwall: STA BX                  ; přilep ke zdi...
        LDA #0
        STA BX_LO
        LDA #0                  ; ...a otoč VX (16bitový dvojkový
doplněk)
        SUB VX_LO
        STA VX_LO
        LDA #0
        SBC VX_HI
        STA VX_HI
.xok:
```

První čtyři instrukce jsou celý „pohyb“: Q8.8 součet polohy s rychlostí. Zbytek je kontrola mantinelů `BX_MIN` / `BX_MAX` (4 a 152 — čtyřpixelový míček mezi zdmi v buňkách 0 a 39) a při nárazu dvojice úkonů: **přilepit** polohu přesně na zeď (i se zlomkem — proto `STA BX_LO` s nulou) a **otočit** rychlost. Přilepení je důležité: kdybychom polohu nechali „za zdí“, mohl by míček při nešťastném zlomku uvíznout a odrážet se donekonečna.

Osa Y je stejná píseň s jedním rozdílem: odraz existuje jen nahoře (`BY_MIN`, pod horní zdí). Dole se místo odrazu kontroluje `BY_DEAD` — a v téhle etapě míček, který proletí, prostě dostane nové podání: `JMP serve`. Je to lešení; skutečný trest (životy) přijde v kapitole 10.

> **Pozor**
>
> Všimni si dvojice `JC .xok` / `JZ .xok` u pravé zdi. `CMP` porovnává jen celé pixely a „přesně na zdi" je platná poloha — teprve „za zdí" je náraz. Kdyby chybělo `JZ`, míček by se odrážel o pixel dřív a u pravé zdi by vznikla neviditelná bariéra. Hraniční podmínky jsou v herní fyzice polovina práce.

Kdo kreslí

`ball_step` polohu jen počítá. Kreslení zůstává v hlavní smyčce — po fyzice se celé pixely polohy opíšou do atributů spritu 0, s obligátním posunem:

```
.draw:  LDA BX                  ; sprite 0 ukazuje celočíselnou
polohu míčku
        ADD #16                 ; hardwarový posun obrazu +16
        STA SPR0_X
        LDA BY
        ADD #16
        STA SPR0_Y
        JMP main
```

Oddělení „spočítej" a „nakresli" není pedanterie: až budeme v kapitole 9 vracet míček po zásahu cihly zpátky, oceníš, že fyzika žije jen v nulté stránce a obraz je pouhý její otisk, pořízený jednou za snímek.

Spusť milník a nech ho běžet: podání, výkop, klikatá dráha mezi stropem a zdmi — a věčný návrat na pálku. Hypnotické, ale bez rizika to není hra. Čas naučit pálku chytat.

ČÁST IV

Pravidla hry

KAPITOLA 8

Odraz od pálky

> **MILNÍK ETAPY — `e07-odraz.asm`**
>
> Pálka chytá: míček se od ní odráží a úhel odrazu určuje místo dopadu — kraj pálky ho vystřelí do strany, střed skoro kolmo vzhůru. Vzniká nekonečná vybíjená; propadlý míček se pořád ještě jen podává znovu.

Cihly budeme v příští kapitole hledat **pohledem do obrazovky** — obraz je pro ně kolizní mapa. S pálkou to nejde a nemá to jít: pálka je sprite, ve znakové mřížce po ní není ani stopa, a hlavně její poloha žije v jemnějším rozlišení. Kolize s pálkou je proto **čistá aritmetika** ve světě tlustých pixelů: dva intervaly na ose X a otázka, jestli se protínají.

Kdy vůbec testovat

Test má smysl jen v úzkém okně. `ball_step` proto nejdřív levně vyloučí všechno, co pálku zajímat nemůže:

```
        LDA VY_HI
        AND #0x80
        JNZ .done               ; letí nahoru — chytit ho nemůže
        LDA BY
        CMP #BY_PAD
        JC .done                ; ještě nad pásmem pálky
        CMP #BY_LATE
        JNC .done               ; zapadl moc hluboko — je pryč
```

První test čte **znaménkový bit** svislé rychlosti: míček letící vzhůru se od pálky neodráží (prolétá kolem ní při výkopu). Další dva vymezují svislé **pásmo chytání** od `BY_PAD` (89 — míček sedí přesně na liště) po `BY_LATE` (94): míček výš je ještě ve hře, míček níž už pálku minul a padá do ztracena. Pásmo je široké 5 tlustých pixelů, víc než jeden krok míčku za snímek — takže mezi dvěma snímky nemůže celé pásmo **přeskočit**.

Průnik intervalů

Teprve teď přijde vlastní test: překrývá se míček (interval od `BX` do `BX+4`) s pálkou (od `PAD_POS` do `PAD_POS+24`)?

```
        LDA BX
        ADD #4                  ; pravý okraj míčku
        CMP PAD_POS
        JC .done                ; celý nalevo od pálky
        JZ .done                ; okraje se jen dotýkají
        LDA PAD_POS
        ADD #PAD_WIDTH
        STA ZT
        LDA BX
        CMP ZT
        JNC .done               ; celý napravo od pálky
```

Učebnicová definice průniku dvou intervalů: **ne**(A končí před B) a **ne**(A začíná za B). Dvě porovnání, čtyři podmíněné skoky — a zase pozor na hranice: dotyk okrajů (`JZ .done`) se nepočítá, míček musí pálku opravdu překrýt.

Míření: pálka jako raketa

Kdyby se míček od pálky odrážel jen zrcadlově, byla by hra loterie — hráč by neměl jak ovlivnit, kam poletí. Skutečný Arkanoid dělá to, čemu hráči říkají „míření pálkou“: **místo dopadu na pálce určuje úhel odrazu**. Dopad na střed vrací míček skoro kolmo, dopad na kraj ho vystřelí ostře do strany. Tady je celá ta muzika:

```
        ; chycen! — posaď na lištu, pošli nahoru, zamiř podle
místa dopadu
        LDA #BY_PAD
        STA BY
        ...
        LDA BX
        ADD #2                  ; t = střed míčku - levý okraj
pálky: 0..25
        SUB PAD_POS
        JNC .tpos
        LDA #0                  ; škrtl o levý roh — přimkni k
čelu pálky
.tpos:  STA ZT
        CMP #PAD_MID
        JC .aim_l
```

```
        SUB #PAD_MID            ; pravá půlka: velikost 0..13
        SHL
        SHL                     ; x4 -> boční rychlost Q8.8 (max
~51 px/s)
        CMP #VX_MIN
        JNC .vr
        LDA #VX_MIN             ; i dopad na střed dostane malý
drift
.vr:    STA VX_LO
        LDA #0
        STA VX_HI
        RET
```

Postup po krocích. Nejdřív se spočítá **parametr dopadu** `t`: vzdálenost středu míčku od levého okraje pálky, 0 až 25. Porovnáním s `PAD_MID` (12) se rozhodne, na které polovině pálky míček přistál. Velikost výchylky od středu se pak **čtyřmi vynásobí** dvěma posuvy `SHL` (kapitola 13 první knihy: posouvej místo násobení) — a to je rovnou nová vodorovná rychlost ve zlomkovém bajtu Q8.8. Dopad 13 pixelů od středu tak dává 52/256 pixelu na snímek, asi 51 px/s; dopad na střed skoro nulu.

Skoro — a právě proti „skoro nule“ stojí konstanta `VX_MIN`. Míček letící svisle je herní past: odráží se mezi stropem a pálkou po stejné dráze donekonečna a hráč s tím nic nenadělá. Proto každý odraz dostane aspoň minimální boční drift 6/256 px na snímek. Levá polovina pálky (`.aim_l`, v milníku hned pod ukázkou) počítá zrcadlově a výsledek otočí dvojkovým doplňkem na zápornou rychlost.

> **Poznámka**
>
> Součet obou polovin dává přes dvacet různých úhlů odrazu — z jediného bajtu `t` a čtyř aritmetických instrukcí. Žádné tabulky sinů, žádná goniometrie: osmibitové hry si úhly **vymýšlejí** tak, aby se dobře počítaly, a hráč rozdíl nepozná. Kdo někdy hledal v kódu starých automatovek „fyziku“, našel většinou právě takovéhle chytré podvody.

Za zmínku stojí ještě řádek `LDA #0 / .tpos:` nad výpočtem: když míček škrtne o levý **roh** pálky, vyjde `t` záporné (odečítání podteče) — a místo záporného míření se `t` přimkne k nule, jako by míček trefil úplný kraj. Rohy jsou v herní fyzice vždycky zvláštní případ; tenhle stojí jednu instrukci.

Konec smyčky, který není RET

Poslední drobnost té etapy je nenápadná, ale krásná. Ztracený míček dál končí novým podáním — jenže podívej se, jak se tam skáče:

```
.ytop:  LDA BY
        CMP #BY_DEAD
        JC .alive               ; nad dnem — letí dál
        JMP serve               ; propadl — nové podání (koncové
volání)
```

`ball_step` byl zavolán přes `CALL`, takže na zásobníku leží návratová adresa do hlavní smyčky. `JMP serve` — obyčejný skok, žádné `CALL` — nechá tu adresu ležet, `serve` doběhne a jeho `RET` se vrátí… rovnou do hlavní smyčky, jako by se vracel `ball_step` sám. Tomuhle se říká **koncové volání** (tail call): když je volání jiného podprogramu poslední akcí, skoč místo volání a ušetři jeden návrat i dva bajty na zásobníku (kapitola 11 první knihy vysvětluje, co přesně na tom zásobníku leží).

Spusť milník a zkus si zamířit: chytej míček krajem pálky a sleduj, jak se dráha láme. Najednou je to hra — jen pořád bez soupeře. Toho dodá příští kapitola: sto osm cihel.

KAPITOLA 9

Cihly, skóre a výhra

> **MILNÍK ETAPY — `e08-cihly.asm`**
>
> Konečně se boural: míček rozbíjí cihly po celých párech, skóre v HUD počítá do sto osmi a poslední cihla zbarví okraj dozelena (provizorní výhra). Prohra pořád neexistuje — propadlý míček se podává znovu.

Kde jsou vlastně cihly **uložené**? Nikde — a to je hlavní myšlenka téhle kapitoly. Hra si nevede žádné pole `cihla[6][18]`, žádnou kolizní mapu. Cihly existují jedině **na obrazovce**: buňka s znakem `CH_BRKL` nebo `CH_BRKR` je cihla, buňka s mezerou není. Stejný trik nesl Cave Runner v závěru první knihy a nese i našeho Arkanoida: **data hry jsou obraz**. Ušetří se paměť, ušetří se synchronizace („mapa říká cihla, obrazovka říká mezera“ se prostě nemůže stát) — a kolize je jedno čtení z paměti.

Ze světa spritů do mřížky

Míček žije v tlustých pixelech, cihly v buňkách. Převod známe z kapitoly 5 — buňka je 4 × 4 tlusté pixely, takže stačí dělit čtyřmi, a dělení mocninou dvou jsou posuvy:

```
.alive: LDA BX
        ADD #2                  ; střed míčku, tlusté px
        SHR
        SHR                     ; /4: tlusté px -> textová buňka
        STA CELL_X
        LDA BY
        ADD #2
        SHR
        SHR
        STA CELL_Y
        CALL calc_addr
        LDA (PTR)
        CMP #CH_BRKL
        JZ .hit_l
        CMP #CH_BRKR
        JZ .hit_r
```

`ADD #2` posune souřadnici z levého horního rohu na **střed** čtyřpixelového míčku — testujeme buňku pod středem, ne pod rohem. Pak `calc_addr` z kapitoly 4 a jediné `LDA (PTR)`: tři možnosti — levá půlka cihly, pravá půlka, nebo nic. Celá detekce kolize s hracím polem stojí dvanáct instrukcí.

> **Pozor**
>
> Testovat jen střed je vědomé zjednodušení — stejné, jaké měl skutečný osmibitový Breakout. Míček, který o cihlu jen škrtne rohem, může „protunelovat" bez zásahu. Poctivý test čtyř rohů by stál čtyřnásobek času každý snímek kvůli jevu, který hráč skoro nikdy neuvidí. Herní programování je umění vybrat si, co si můžeš dovolit zanedbat.

Partnerská buňka

Cihla jsou **dvě** buňky, ale zasažena je jen jedna. Kde leží ta druhá? Odpověď je zakódovaná přímo v tom, **kterou** půlku jsme trefili — a přesně proto má hra dva znaky cihel místo jednoho:

```
        ; --- zásah cihly: půlka glyfu prozradí, kde leží
partnerská buňka ---
.hit_l: LDA #CH_SPACE
        STA (PTR)
        INW PTR                 ; partner je o buňku napravo
        LDA #CH_SPACE
        STA (PTR)
        JMP .smash
.hit_r: LDA #CH_SPACE
        STA (PTR)
        DEW PTR                 ; partner je o buňku nalevo
        LDA #CH_SPACE
        STA (PTR)
```

Levá půlka -> partner napravo (`INW`, šestnáctibitový inkrement ukazatele), pravá půlka -> partner nalevo (`DEW`). Obě buňky se přepíšou mezerou — a to je vše. Barva? Ta zůstane v barevné rovině ležet, jak jsme si připravili v kapitole 4: prázdná buňka s červeným inkoustem vypadá prostě černá. Rozbíjení cihel se barevné roviny **vůbec nedotýká**.

> **Tip**

> Vzor „identita znaku nese informaci“ je levnější sourozenec datových struktur. Tady kóduje jediný bit (jsem levá, nebo pravá půlka?) — ale stejným způsobem umí znaková sada nést odolnost cihly, druh bonusu nebo průchodnost políčka. Obrazovka je pak zároveň mapa **i databáze**.

Odraz na místě

Po smazání cihly se míček musí odrazit. Jenže pozor: jeho nová poloha už je **uvnitř** zbořené cihly — kdybychom jen otočili rychlost, další snímek by ho zavedl do sousední buňky a mohl by „prokousat“ celou řadu naráz. Řešení: vrátit čas.

```
.smash: INC SCORE
        CALL print_score
        DEC BRICKS              ; DEC v paměti nastavuje Z —
podmínka výhry
        JZ .do_win
        LDA OLDY_LO             ; odraz na místě: vrať pohyb v ose
Y...
        STA BY_LO
        LDA OLDY
        STA BY
        LDA #0                  ; ...a otoč VY (cihly jsou
vodorovné plochy,
        SUB VY_LO               ;    ať jsme přiletěli odkudkoli)
        STA VY_LO
        LDA #0
        SBC VY_HI
        STA VY_HI
        RET
```

Dvojice `OLDY` / `OLDY_LO` je snímek svislé polohy **před** pohybem — `ball_step` si ho od téhle etapy ukládá na začátku výpočtu osy Y. Odraz od cihly tedy polohu obnoví a otočí svislou rychlost: míček se odrazí „na místě“, jako by do cihly nikdy nevstoupil. Vodorovná rychlost se nemění — cihly bereme jako vodorovné plochy, což je pro pole široké osmnácti cihel vedle sebe věrné přiblížení.

Skóre přes DIV

`INC SCORE` počítá, `print_score` maluje. Převod čísla 0–108 na tři číslice je dvojí dělení deseti — a `DIV` na tomhle stroji vrací podíl v **`A`** a **zbytek v `B`**, takže jedno dělení dá rovnou obě informace:

```
print_score:
        LDA #10
        TAB
        LDA SCORE
        DIV                     ; A = skóre/10, B = jednotky
        STA ZT
        TBA
        ADD #'0'
        STA SCORE_ONE
```

Zbytek (jednotky) se vyzvedne `TBA` — a zase platí pravidlo z kapitoly 4: **vyzvednout dřív, než ho další operace přepíše**. Přičtení ASCII kódu `'0'` udělá z číslice znak a zápis přímo do VRAM na pozici v HUD ho zobrazí; azurovou barvu má buňka od initu. Druhé kolo dělení oddělí stovky od desítek. Celé skóre: dvanáct instrukcí, žádná smyčka.

Výhra za jeden příznak

Kolik cihel zbývá, hlídá čítač `BRICKS` — a podmínka výhry je schovaná v jediném řádku, který jsi možná přehlédl: `DEC BRICKS` následované `JZ .do_win`. Instrukce `DEC` nad pamětí nastavuje příznak nuly podle výsledku, takže „sniž počet a zjisti, jestli došly“ je jedna operace. Žádné `LDA BRICKS / CMP #0`.

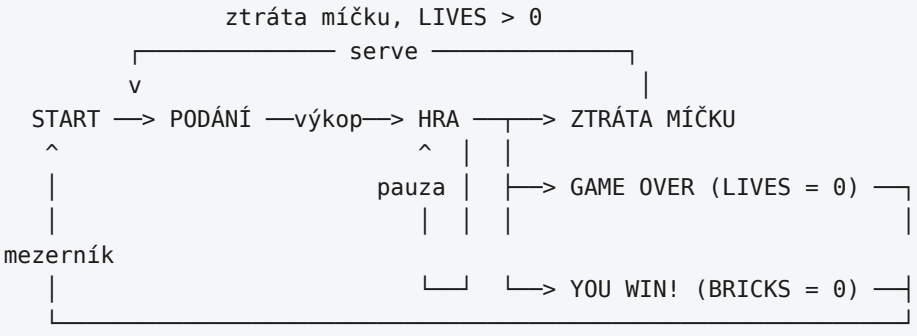
V tomhle milníku vede `.do_win` na poslední lešení knížky: zelený okraj a `HLT`. Skutečná obrazovka výhry potřebuje stavový automat — a ten si zaslouží celou příští kapitolu.

KAPITOLA 10

Životy, konec a pauza

> **MILNÍK ETAPY — `e09-stavy.asm`**
>
> Kompletní hra — jen němá. Tři životy s číslicí v HUD, obrazovky GAME OVER a YOU WIN! s restartem na mezerník a pauza, která umí zmrazit hru uprostřed letu míčku. Všechna lešení jsou stržena.

Hra, kterou jde jen hrát, ale ne prohrát, vyhrát ani přerušit, není hra — je to demo. Téhle etapě se v herním žargonu říká **stavový automat**: hra se v každém okamžiku nachází v jednom z několika stavů a mezi nimi přechází po pevných pravidlech.



Zajímavé je, **kde** ty stavy v programu bydlí. Žádná proměnná `STAV` neexistuje. PODÁNÍ je „`SERVE_DLY` je nenulové“, PAUZA je příznak `PAUSED`, a GAME OVER je prostě **místo v kódu**, kde program zrovna stojí — smyčka, která čeká na mezerník. Stavový automat nemusí být tabulka; u malých her je to půdorys programu.

Životy

Ztráta míčku konečně něco stojí. Místo lešení `JMP serve` je tu plná větev:

```
        ; --- míček ztracen pod pálkou ---
.lost:  DEC LIVES
        LDA LIVES
        ADD #'0'
        STA LIVES_DIG
```

```
        LDA LIVES
        JZ .do_over
        JMP serve               ; koncové volání — RET v serve
vrací do main
.do_over:
        JMP game_over
```

Sniž počet, přepiš číslici v HUD (jednociferné číslo — stačí `ADD #'0'`, žádné dělení), a rozhodni: zbývají životy -> nové podání (známé koncové volání), nezbývají -> konec hry.

Obrazovky konce a studený restart

`game_over` a `win` dělají totéž s jiným nápisem, tak si ukažme jen jeden:

```
game_over:
        LDA #0
        STA SPR_ENABLE          ; bez aktérů hřiště zamrzne
        BCOPY msg_over, MSG_OVER_AT, 9
        BFILL CMSG_OVER, 9, INK_OVER
        JMP wait_restart
```

Vypnout sprity znamená **zmrazit jeviště**: kulisy (zdi, zbylé cihly, skóre) zůstanou stát jako důkazní materiál, jen herci zmizí. Přes střed se blitterem otiskne nápis — řádek 12 je za hry zaručeně prázdný, cihly končí řádkem 8 a míček je sprite, takže se nemůže stát, že bychom nápisem něco přepsali. A pak se čeká:

```
wait_restart:
.wrel:  LDA PAD1                ; co se ještě drží, musí napřed
nahoru...
        OR PAD2
        AND #BTN_FIRE
        JNZ .wrel
.wprs:  LDA PAD1                ; ...a teprve čerstvý mezerník
spustí novou hru
        OR PAD2
        AND #BTN_FIRE
        JZ .wprs
        JMP start
```

Dvě smyčky za sebou vynucují **čerstvý stisk**: první čeká, až hráč mezerník pustí, druhá na nový stisk. Kdyby tam byla jen druhá, hráč, který zrovna pauzoval (a mezerník ještě drží), by obrazovku konce proletěl, ani by ji nezahlédl.

A restart? Jediný `JMP start`. Tady se zúročí investice z kapitol 3 a 4: init staví **celou** obrazovku od nuly blitterem, takže po něm není co uklízet — rozbité cihly, skóre, poloha pálky, všechno se prostě postaví znovu. Hry, které místo toho zkoušejí svět „opravit“ (vrátit cihly, vynulovat proměnné jednu po druhé), si koledují o zapomenutou proměnnou a nevysvětlitelné chyby po druhém restartu. **Studený start je nejspolehlivější úklid.**

Pauza: hrana místo úrovně

Mezerník má hru pozastavit. Jenže gamepad je **úrovňový** — hlásí „drží se“, ne „právě stiskl“ — a smyčka běží 250krát za sekundu. Kdybychom pauzu přepínali podle drženého bitu, přepne se 250krát za sekundu tam a zpět. Potřebujeme z úrovně vyrobit **hranu**, přesně jak radil tip v kapitole 16 první knihy: pamatuj si minulý stav.

```
        AND #BTN_FIRE
        JZ .fup
        LDA FIRE_PREV
        JNZ .fdn                ; drží se už od minulého snímku —
to není stisk
        LDA PAUSED
        XOR #0x01               ; čerstvý stisk: překlop stav
pauzy
        STA PAUSED
        CALL pause_note
.fdn:   LDA #1
        STA FIRE_PREV
        JMP .pchk
.fup:   LDA #0
        STA FIRE_PREV
.pchk:  LDA PAUSED
        JZ .input
        JMP main                ; zmrazeno — přeskoč vstup, fyziku
i kreslení
```

`FIRE_PREV` nese stav tlačítka z minulého snímku; jen kombinace „teď drženo **a** minule ne“ je stisk. Ten překlopí `PAUSED` operací `XOR #0x01` — zapnuto/vypnuto jednou instrukcí — a `pause_note` otiskne či smaže žlutý nápis PAUSED (zase blitter, zase řádek 12).

Samotné zmrazení je poslední řádek: dokud `PAUSED` platí, smyčka skočí rovnou na `main`. Vstup, fyzika ani kreslení neproběhnou — svět stojí — ale `wait_frame` běží dál, takže čas se měří pořád a po odpauzování hra naváže dalším snímkem, jako by se nic nestalo. Míček zamrzlý uprostřed letu je mimochodem skvělá příležitost prohlédnout si hru zblízka; zkus to.

Poslední jemnost najdeš až v initu, a stojí za ní pěkný příběh:

```
        LDA #1                  ; palbu ber při startu jako "už
držen
ou": stisk,
        STA FIRE_PREV           ;   který hru restartoval, nesmí
tu novou
        LDA #3                  ;   rovnou zapauzovat
        STA LIVES
```

Restart z obrazovky konce spouští stisk mezerníku — a hráč ho v okamžiku `JMP start` pořád drží. Kdyby nová hra začínala s `FIRE_PREV = 0`, viděla by v prvním snímku „čerstvý stisk“ a poslušně by se zapauzovala dřív, než se vůbec rozsvítí. Init proto lže: tvrdí, že mezerník už držený byl. Hranový detektor pak startovní stisk ignoruje. Takovéhle chyby — viditelné jen při přechodu mezi stavy — jsou u stavových automatů nejčastější; tahle je opravená jednou instrukcí a dvouřádkovým komentářem, proč tam je.

Hra je kompletní. Spusť milník a prohraj, vyhraj, pauzni, restartuj — všechno drží. Jediné, co chybí, je slyšet. Pojďme tomu stroji rozvázat jazyk.

ČÁST V

Zvuk a poslední šlif

KAPITOLA 11

Tři hlasy a ADSR

> **MILNÍK ETAPY — `e10-zvuk.asm`**
>
> Hra promluví: měkké kliknutí při odrazu od zdi, šumový výbuch při rozbití cihly a dlouhý tón, kterým se hlásí konec hry — hluboký pro prohru, jasný pro výhru.

Zvukový čip známe z kapitoly 17 první knihy: tři nezávislé hlasy, frekvence rovnou v hertzích, pět tvarů vlny a obálky ADSR, které tón tvarují samy. Tahle kapitola je o tom, jak se z těch součástek skládá **zvuk hry** — a začít musíme, kupodivu, plánováním.

Plán kanálů: kdo mluví kterým hlasem

Hlasy jsou tři a efektů víc. Naivní přístup — „každý efekt hraje na kanálu 0“ — má ošklivou vadu: nový zvuk usekne ten předchozí. Rozbiješ cihlu těsně po odrazu a klik zdi zmizí v půlce. Řešení je rozdělit hlasy jako role v kapele, jednou provždy:

Kanál	Role	Tvar vlny
0	odrazy: klik zdi (a od příští etapy boing pálky)	trojúhelník, obdélník
1	dlouhé tóny: konec hry (a zavrčení ztraceného míčku)	pila
2	perkuse: výbuch cihly	šum — natrvalo

Tabulka 4. Plán kanálů. Efekty, které se mohou potkat, nikdy nesdílí hlas.

Klíčová úvaha: **které zvuky se mohou zaznít současně?** Odraz od zdi a rozbitá cihla — klidně ve stejném snímku; proto různé kanály. Klik zdi a boing pálky — nikdy (míček nemůže trefit zeď a pálku naráz); proto smí sdílet kanál 0. Takhle se třemi hlasy obslouží pět efektů, aniž si kdy skočí do řeči.

Stín GATES a retrigger

Všechny gate bity žijí v jednom registru `AUD_CTRL` — ale my potřebujeme zapínat a vypínat **jednotlivé** kanály, každý ze svého efektu. Hra si proto vede **stín** registru v nulté stránce a do hardwaru zapisuje jen kopii:

```
; A = bit(y) gate. Gate spouští hrana, takže nota se přehraje
znovu shozením
```

```
; bitu a novým zvednutím — vždy snd_off před snd_on.
snd_on: OR GATES
        STA GATES
        STA AUD_CTRL
        RET

snd_off:XOR #0xFF
        AND GATES
        STA GATES
        STA AUD_CTRL
        RET
```

`snd_on` přiORuje bit ke stínu, `snd_off` ho maskou odAndí (`XOR #0xFF` vyrobí z bitu masku „všechno kromě něj“). A proč to věčné „napřed off, pak on“? Vzpomeň si: gate reaguje na **hranu**. Když kanál ještě hraje a my chceme tentýž zvuk spustit znovu — druhá cihla těsně po první — musí bit napřed spadnout a zase vstát, jinak se nic nestane. Dvojice `snd_off` / `snd_on` je zaručený retrigger.

> **Poznámka**
>
> Stínová kopie registru je obecný vzor pro **zapisovací** registry sdílené více pisateli: místo čtení hardwaru (které nemusí být možné nebo levné) se čte kopie v paměti a hardware jen přijímá. Potkáš ho u masek přerušení, konfiguračních registrů i tady u zvuku.

Efekt = nastav a zapomeň

A teď ta hlavní krása ADSR. Podívej se na celý klik zdi:

```
snd_wall:                       ; měkký trojúhelníkový klik —
odraz od zdi
        LDA #GATE_CH0
        CALL snd_off
        LDA #<600
        STA AUD_FREQ_LO
        LDA #>600
        STA AUD_FREQ_HI
        LDA #2
        STA AUD_WAVE            ; trojúhelník
        LDA #0x04               ; attack 0, decay 4 (~32 ms)
        STA AUD_AD
```

```
        LDA #0x02               ; sustain 0 — klik, zmizí sám
        STA AUD_SR
        LDA #GATE_CH0
        JMP snd_on
```

Sustain je **nula**: tón naskočí okamžitě (attack 0), za 32 ms doklesá na nic (decay 4) — a je po něm. Program se o zvuk už nikdy nestará: žádné počítadlo „za 8 snímků vypni“, žádný stav. **Obálka je časovač, který běží v hardwaru.** Volající udělá `CALL snd_wall` a jde si po svých; hra zůstává stejně jednoduchá jako předtím, jen zní.

Výbuch cihly je stejný vzor s jinými parametry — šumový hlas (nastavený jednou provždy v `snd_init`) a frekvence, která u šumu neurčuje tón, ale **takt generátoru šumu**: 2 400 Hz dává jasný, krátký sykot. Decay 3, sustain 0: perkuse. Voláme ho ze `.smash` v `ball_step` těsně před `INC SCORE` — jedna instrukce navíc a rozbíjení cihel dostane šťávu.

Závěrečný tón: sustain a busy-wait

Obrazovky konce chtějí opak kliknutí: tón, který **drží**, dokud ho nevypneme. Recept: plný sustain a nenulový release —

```
end_tone:
        LDA #3
        STA AUD_WAVE+5          ; pila na hlasu závěrečného tónu
        LDA #0x00               ; okamžitý attack, žádný decay
        STA AUD_AD+5
        LDA #0xF3               ; plný sustain, release 3 — zazní
a dozní
        STA AUD_SR+5
        LDA #GATE_CH1
        CALL snd_on
        LDY #30                 ; ~1 s znění (30 x 256 x ~8 T)
.ring:  LDX #0
.spin:  DEX
        JNZ .spin
        DEY
        JNZ .ring
        LDA #0                  ; pusť všechny hlasy — čekání je
potichu
        STA GATES
        STA AUD_CTRL
```

Zápisy s `+5` míří do registrů **kanálu 1** — pětice registrů každého kanálu leží 5 bajtů nad předchozí (kapitola 17 první knihy), takže `AUD_WAVE+5` je tvar vlny jedničky. Frekvenci nastavil volající: `game_over` hlubokých 220 Hz, `win` jasných 880 Hz — tentýž kód, dvě nálady.

Tón se nechá znít vteřinu obyčejnou zpožďovací smyčkou. V herní smyčce by byla hřích — ale tady **hra stojí**, na obrazovce svítí GAME OVER a čekat je přesně to, co chceme. Pak se všechny hlasy pustí (release je nechá elegantně doznít) a program pokračuje do známého čekání na čerstvý mezerník. Konec hry zní, ticho se vrací, kdo chce, hraje znovu.

Zbývají dva efekty, které jsem záměrně přeskočil: boing pálky a zavrčení ztraceného míčku. Oba totiž potřebují něco, co statická obálka neumí — **tón, který se hýbe**. To je poslední dílek stavby.

KAPITOLA 12

Klouzavé efekty

> **MILNÍK ETAPY — `e11-arkanoid.asm`**
>
> Finále. Odraz od pálky dostane stoupající „boing“, ztracený míček dlouhé klesající zavrčení — a tím je hra hotová: instrukce po instrukci totožná s Arkanoidem z ukázek emulátoru. Poslední milník je zároveň kompletní výpis v příloze A.

Obálka ADSR tvaruje **hlasitost** tónu, ale jeho výška stojí na místě. Jenže ty nejcharakterističtější herní zvuky — „boing“, padající siréna, vzlet rakety — jsou právě tóny, které **jedou po frekvenci**. Hardware nám s tím nepomůže; frekvenční registr se sám od sebe nezmění. Kdo ho tedy bude měnit? Ten samý mechanismus, který hýbe vším ostatním: herní smyčka, jednou za snímek.

Klouzání jako stav

Hra si ke zvuku přibere čtyři bajty stavu v nulté stránce:

Proměnná	Význam
`SND_DLY`	kolik snímků klouzání ještě poběží (0 = nic neklouže)
`SND_SLIDE`	znaménková 16bitová změna frekvence za snímek (dolní bajt + znaménko)
`SND_CH`	offset registrů klouzajícího kanálu (0 = kanál 0, 5 = kanál 1)
`SND_MASK`	gate bit téhož kanálu — kdo má být na konci vypnut

Tabulka 5. Stav klouzavého efektu. Klouže vždy nejvýš jeden zvuk — hře to stačí.

A do hlavní smyčky přibude hned za `wait_frame` jediné volání: `CALL snd_frame`. To je ta poslední změna smyčky v celé knížce — porovnej si její konečnou podobu s plánem z kapitoly 2.

```
snd_frame:
        LDA SND_DLY
        JZ .done
        DEC SND_DLY             ; DEC v paměti nastavuje Z
        JNZ .slide
        LDA SND_MASK            ; klouzání skončilo — pusť ten
```

```
hlas
        JMP snd_off
.slide: LDX SND_CH              ; blok registrů klouzajícího
kanálu
        LDA SND_SLIDE           ; 16bitový znaménkový součet: freq
+= slide
        ADD AUD_FREQ_LO,X
        STA AUD_FREQ_LO,X
        LDA SND_SLIDE+1         ; rozšíření znaménka (0x00 stoupá,
0xFF klesá)
        ADC AUD_FREQ_HI,X
        STA AUD_FREQ_HI,X
.done:  RET
```

Když nic neklouže, stojí to tři instrukce a návrat. Jinak: odpočítej snímek, a buď přičti `SND_SLIDE` k frekvenci, nebo — když odpočet právě došel — kanál vypni (gate dolů, release ho nechá doznít). Frekvence se čte i píše **z hardwarových registrů samotných**: frekvenční registry jsou na tomhle stroji čitelné, takže netřeba stínovat.

Za druhý pohled stojí adresování. `LDX SND_CH` a pak `ADD AUD_FREQ_LO,X` — indexovaný přístup, kde **X** není index v poli, ale **výhybka mezi kanály**: s X = 0 pracuje kód s kanálem 0, s X = 5 s kanálem 1. Pětibajtový odstup registrů kanálů, na který upozorňovala už první kniha, tu nese celou tíhu: jeden kód, kterýkoli hlas.

Boing a zavrčení

Efekty s klouzáním jen vyplní stav navíc. Boing pálky — obdélník na 700 Hz, který 20 snímků šplhá o 10 Hz nahoru (dohromady +200 Hz za 80 ms):

```
        LDA #20                 ; klouzej nahoru 20 snímků (~80
ms)
        STA SND_DLY
        LDA #10
        STA SND_SLIDE
        LDA #0                  ; kladná změna — znaménko doplní
nuly
        STA SND_SLIDE+1
        STA SND_CH              ; registry kanálu 0
        LDA #GATE_CH0
        STA SND_MASK
        JMP snd_on
```

Zavrčení ztraceného míčku je zrcadlo: pila na 500 Hz, sustain skoro plný, a 108 snímků klesání po −2 Hz — `SND_SLIDE` dostane `0xFE` s horním bajtem `0xFF`, dvojkový doplněk v akci naposledy. Trvá 0,43 s a **doznívá přesně během klidu podání**, kdy žádný jiný efekt vzniknout nemůže — proto si hra vystačí s jediným slotem pro klouzání a nemusí řešit, co kdyby klouzaly dva zvuky naráz. Není to náhoda, je to návrh: délky efektů jsou sladěné s rytmem hry.

> **Tip**
>
> Až budeš skládat vlastní efekty, začni od fyziky zvuku, ne od čísel: nárazy jsou krátké a bez sustainu, „pružné“ věci stoupají, ztráty a pády klesají, výbuchy jsou šum. Pak teprve laď hertze a snímky. Dobrý herní zvuk poznáš tak, že ho **nevnímáš** — jen hra najednou „sedí“.

Finále

A to je všechno. Vážně: přidej `CALL snd_pad` do větve chycení míčku, `CALL snd_lost` do větve ztráty, a máš před sebou `e11-arkanoid.asm` — 978 řádků, 2 361 bajtů, instrukce po instrukci tatáž hra, kterou najdeš v emulátoru mezi ukázkami (`examples/games/arkanoid.asm`; liší se jen jazykem komentářů). Spusť ji a dohraj až k YOU WIN!. Zasloužíš si to — a hra už taky.

Stavba je hotová, lešení odvezené. V poslední kapitole se projdeme kolem hotového domu: co jsme se vlastně naučili, co jsme zamlčeli a kam se vydat dál.

KAPITOLA 13

Co dál

Jedenáct etap, 978 řádků, 2 361 bajtů. Postavil jsi hru — a cestou sis osahal celý repertoár osmibitového herního programátora. Stojí za to si ho na chvíli přehlédnout vcelku, protože to je zároveň seznam věcí, které teď umíš použít jinde:

- **ukotvená herní smyčka**, která drží reálný čas na jakémkoli taktu (etapa 1),
- **dělba práce s hardwarem**: blitter na hromadné přesuny, sprity na pohyb, obálky na zvuk — procesoru zbývá logika (etapy 2, 4, 10),
- **vzor a klon**: postav jednou, rozmnož DMA (etapa 3),
- **pevná řádová čárka Q8.8** se znaménkem v dvojkovém doplňku (etapy 5–6),
- **obrazovka jako databáze**: kolizní mapa i stav hry v jedné paměti (etapa 8),
- **stavový automat** z příznaků a míst v kódu, hranová detekce vstupu, studený restart (etapa 9),
- **zvukový návrh**: plán kanálů, nastav-a-zapomeň, klouzání po snímcích (etapy 10–11).

Rozpočet, který zbyl

V kapitole 1 jsme slibovali, že rozpočet bude volný — teď to můžeme spočítat. Při taktu 2 MHz má snímek (4 ms) rozpočet 8 000 T-stavů. Nejdražší běžný snímek hry — pohyb pálky, `ball_step` s odrazem od cihly, `print_score`, zvuk — spotřebuje řádově stovky T-stavů; zbytek prosedí `wait_frame` v čekací smyčce. Jinými slovy: zbývá řádově **devadesát procent** stroje. To není plýtvání, to je prostor — přesně do něj se vejdou všechna rozšíření, která tě napadají.

Co jsme zamlčeli

Poctivost velí přiznat zjednodušení — obě jsme potkali cestou, obě mají i skutečné osmibitové předlohy:

- **Tunelování:** míček se testuje jen ve středové buňce, takže dokonale rohový zásah může cihlou proletět. Oprava (test čtyř rohů, nebo krokování dráhy po pixelech) stojí násobky času každý snímek za jev, který nastane jednou za sto her.
- **Pásmo chytání:** pálka chytá míček ve svislém okně 5 tlustých pixelů. Míček rychlejší než 5 px/snímek by mohl okno přeskočit — naše rychlosti jsou ale nejvýš 0,1875 px/snímek, takže rezerva je šestadvacetinásobná. Kdybys ale zrychloval, na tohle číslo si vzpomeň.

Nápady na přestavbu

Hotová hra je nejlepší staveniště. Pár nápadů, seřazených zhruba podle obtížnosti — všechny se obejdou bez nových konceptů, vystačíš s tím, co znáš z téhle knížky:

1. **Ladění obtížnosti.** Jedno odpoledne s konstantami: rychlost míčku, rozjezd pálky, délka podání. Uvidíš, jak křehká věc je herní pocit.
2. **Zrychlování.** Každých dvacet rozbitých cihel zvyš `VY` o kousek (pozor na pásmo chytání!). Stačí čítač a pár řádků ve `.smash`.
3. **Odolné cihly.** Druhá dvojice znaků (`0x06` / `0x07`) pro cihlu, která vydrží dva zásahy: první zásah ji přepíše na obyčejnou, druhý smaže. Obrazovka-databáze v akci.
4. **Bonusy.** Padající znak z rozbité cihly (stačí znaková mřížka, žádný sprite), který pálka chytá: širší pálka, pomalejší míček, život navíc.
5. **Kola.** Po YOU WIN! další úroveň s jiným vzorem cihel — rozlož pole cihel do datové tabulky a init ji nech číst. Dobrá lekce oddělení dat od kódu.
6. **Dva míčky.** Zdvojení stavu míčku a smyčka přes ně; sprite 3 se nabízí. Nejtěžší položka seznamu — a nejlepší test, jestli rozumíš `ball_step` do posledního řádku.

Kam dál

V ukázkách emulátoru na tebe čekají dvě hry, které začínají přesně tam, kde tahle knížka končí. **Space Invaders** (`examples/games/space-invaders.asm`) přidává animaci přepisováním glyfů, hudbu z datových tabulek a barvu, která se stěhuje s nepřáteli. **Turbo Run** (`examples/games/turbo-run.asm`) opouští znakovou mřížku úplně a kreslí celé hřiště v bitmapovém režimu. Obě jsou komentované stejně hustě jako náš Arkanoid — a obě už dokážeš číst.

Protože o tom celá tahle knížka byla: ne o jedné hře, ale o tom, že **žádný program není kouzlo**. Je to jen paměť, pár registrů a smyčka, která tiká — a ty teď víš, jak se z nich staví. Tak stav.

Přílohy

PŘÍLOHA A

Kompletní zdroják

Celý hotový Arkanoid — soubor `e11-arkanoid.asm`, poslední milník knížky — tak, jak ho přeloží assembler a spustí emulátor. Výpis je tu ve staré dobré tradici „opisovacích“ výpisů z časopisů osmdesátých let: kdo chce, může si hru přepsat ručně a u toho si ji zopakovat řádek po řádku. Pohodlnější cesta: soubor najdeš v repozitáři emulátoru v `book/arkanoid/examples/`, spustíš ho příkazem `npm run asm` — nebo prostě otevři ukázku Arkanoid na **sap-3.com**, je to tatáž hra.

```
; ============================================================================
; ARKANOID — etapa 11 (finále): kompletní hra
; ============================================================================
; A = doleva, D = doprava (gamepad hlásí držené klávesy, pohyb je plynulý).
; Mezerník (nebo Enter) přepíná pauzu; na obrazovce konce hry restartuje.
; Rozbij všech 108 cihel. 3 životy.
;
; Tento soubor je závěrečný milník knihy "Postavíme si Arkanoid" — kód je
; totožný s examples/games/arkanoid.asm, jen komentáře jsou české.
; Deterministické chování s pevným semínkem RND (CLI --seed).
; ============================================================================

; --- Paměťově mapovaná zařízení ---------------------------------------------
PAD1        EQU 0xFF70      ; gamepad hráče 1 (WASD) — bitmapa držených tlačítek
PAD2        EQU 0xFF71      ; gamepad hráče 2 (šipky) — bitmapa držených tlačítek
BTN_FIRE    EQU 0x10        ; bit 4 = Palba/A (mezerník na PAD1, Enter na PAD2)
; Zvuk: tři hlasy. AUD_CTRL je bitmapa gate (bit N spouští kanál N) a pojmenované
; registry níže patří kanálu 0; kanál 1 sedí o 5 bajtů výš a kanál 2 o 10,
; takže `AUD_WAVE+5` je registr tvaru vlny kanálu 1.
AUD_CTRL    EQU 0xFF20      ; bitmapa gate: bit 0 ch0, bit 1 ch1, bit 2 ch2
AUD_FREQ_LO EQU 0xFF21      ; frekvence v Hz, 16 bitů little-endian (lze i číst)
AUD_FREQ_HI EQU 0xFF22
AUD_WAVE    EQU 0xFF23      ; 0 sinus, 1 obdélník, 2 trojúhelník, 3 pila, 4 šum
AUD_AD      EQU 0xFF24      ; horní půlbajt attack, dolní decay
AUD_SR      EQU 0xFF25      ; horní půlbajt úroveň sustain, dolní release
GATE_CH0    EQU 0b001       ; odrazy (klik zdi, boing pálky)
GATE_CH1    EQU 0b010       ; zavrčení ztraceného míčku, závěrečný tón
GATE_CH2    EQU 0b100       ; šumové perkuse (rozbitá cihla)
CH1_REGS    EQU 5           ; offset registrů kanálu 1 (viz AUD_* výše)
VID_CTRL    EQU 0xFF50      ; bit 0 zapnuto, režim 0 = text 40x25
VID_BORDER  EQU 0xFF54      ; barva okraje, 16barevná paleta ve stylu C64 (&0x0F)
VID_CURY    EQU 0xFF53      ; řádek hardwarového kurzoru; >= 25 ho odstaví z obrazu
SPR_ENABLE  EQU 0xFF58      ; bit i zobrazí sprite i; při překryvu vyhrává sprite 0
CLK_MS_LO   EQU 0xFF3A      ; čítač reálných milisekund, dolní bajt (čtení zamkne
oba)
CLK_MS_HI   EQU 0xFF3B      ; horní bajt (ze zámku)
BLT_SRC_LO  EQU 0xFF72      ; Blitter DMA: dvojice zdroj, cíl, délka,
```

```
BLT_SRC_HI  EQU 0xFF73      ;   výplňový bajt — a zápis příkazu pak provede
BLT_DST_LO  EQU 0xFF74      ;   celý přesun uvnitř toho jediného zápisu
BLT_DST_HI  EQU 0xFF75
BLT_LEN_LO  EQU 0xFF76
BLT_LEN_HI  EQU 0xFF77
BLT_FILL    EQU 0xFF78
BLT_CTRL    EQU 0xFF79      ; zápis 1 = COPY, 2 = FILL

; --- Pomocná makra blitteru (každé se rozvine na ~10 zápisů, běží atomicky) ---
%macro BFILL 3              ; BFILL cíl, délka, bajt
        LDA #<%1
        STA BLT_DST_LO
        LDA #>%1
        STA BLT_DST_HI
        LDA #<%2
        STA BLT_LEN_LO
        LDA #>%2
        STA BLT_LEN_HI
        LDA #%3
        STA BLT_FILL
        LDA #2
        STA BLT_CTRL
%endmacro

%macro BCOPY 3              ; BCOPY zdroj, cíl, délka
        LDA #<%1
        STA BLT_SRC_LO
        LDA #>%1
        STA BLT_SRC_HI
        LDA #<%2
        STA BLT_DST_LO
        LDA #>%2
        STA BLT_DST_HI
        LDA #<%3
        STA BLT_LEN_LO
        LDA #>%3
        STA BLT_LEN_HI
        LDA #1
        STA BLT_CTRL
%endmacro

; --- Rozvržení obrazu ------------------------------------------------------
CHARS       EQU 0x8000      ; znakový buffer, 40x25
COLORS      EQU 0x8C00      ; barevná rovina (POZADÍ<<4 | PÍSMO; 0x00 = výchozí)
GLYPH3      EQU 0x8418      ; slot znaku 0x03 v glyph RAM (8 bajtů na znak)
ROW1        EQU 0x8028      ; řádek 1 VRAM (horní zeď)
HUD_SCORE   EQU 0x8002      ; text "SCORE 000": řádek 0, x=2
SCORE_HUN   EQU 0x8008      ; číslice skóre: řádek 0, x=8..10
SCORE_TEN   EQU 0x8009
SCORE_ONE   EQU 0x800A
HUD_LIVES   EQU 0x801C      ; text "LIVES 3": řádek 0, x=28
LIVES_DIG   EQU 0x8022      ; číslice životů: řádek 0, x=34
MSG_OVER_AT EQU 0x81EF      ; "GAME OVER" vystředěné na řádku 12
```

```
MSG_WIN_AT  EQU 0x81F0      ; "YOU WIN!"  vystředěné na řádku 12
MSG_PAUSE_AT EQU 0x81F1     ; "PAUSED"    vystředěné na řádku 12
CROW0       EQU 0x8C00      ; atributy řádku 0 (řádek HUD)
CROW1       EQU 0x8C28      ; atributy řádku 1 (horní zeď)
CMSG_OVER   EQU 0x8DEF      ; MSG_OVER_AT  + 0x0C00
CMSG_WIN    EQU 0x8DF0      ; MSG_WIN_AT   + 0x0C00
CMSG_PAUSE  EQU 0x8DF1      ; MSG_PAUSE_AT + 0x0C00
INK_HUD     EQU 3           ; azurový text HUD (číslice ji dědí také)
INK_WALL    EQU 12          ; středně šedé zdi
INK_OVER    EQU 10          ; světle červené "GAME OVER"
INK_WIN     EQU 13          ; světle zelené "YOU WIN!"
INK_PAUSE   EQU 7           ; žluté "PAUSED"

; Cihlové pole: řádky 3..8, sloupce 2..37 — adresy znakových řádků + barevných.
; Řádek 3 se staví ručně, řádky 4-8 jsou jeho klony přes Blitter COPY.
BRICK_ROW3  EQU 0x807A      ; 0x8000 + 3*40 + 2
BRICK_ROW4  EQU 0x80A2
BRICK_ROW5  EQU 0x80CA
BRICK_ROW6  EQU 0x80F2
BRICK_ROW7  EQU 0x811A
BRICK_ROW8  EQU 0x8142
CBRICK_R3   EQU 0x8C7A      ; BRICK_ROW3 + 0x0C00, jedna výplň barvy na řádek
CBRICK_R4   EQU 0x8CA2
CBRICK_R5   EQU 0x8CCA
CBRICK_R6   EQU 0x8CF2
CBRICK_R7   EQU 0x8D1A
CBRICK_R8   EQU 0x8D42

; --- Sprity (atributy i pattern RAM jsou obyčejná VRAM) ------------------------
; Souřadnice spritů nesou hardwarový posun +16 (16 = přilepený k okraji)
; v prostoru tlustých pixelů 160x100; tlustý pixel je 1/4 textové buňky.
SPR0_X      EQU 0xAFE0      ; míček
SPR0_Y      EQU 0xAFE1
SPR1_X      EQU 0xAFE4      ; pálka, levých 16 tlustých px
SPR1_Y      EQU 0xAFE5
SPR1_PAT    EQU 0xAFE6
SPR2_X      EQU 0xAFE8      ; pálka, pravých 8 tlustých px
SPR2_Y      EQU 0xAFE9
SPR2_PAT    EQU 0xAFEA
SPR_ATTR    EQU 0xAFE0      ; celá tabulka, pro úvodní vymazání
SPR_PAT0    EQU 0xB000      ; 128 bajtů na vzor: 16 řádků x 8 bajtů, 2 px/bajt
SPR_PAT1R1  EQU 0xB088      ; vzor 1, řádek 1 (tam začíná lišta pálky)
SPR_PAT2R1  EQU 0xB108      ; vzor 2, řádek 1
PAD_SPR_Y   EQU 108         ; atribut Y pálky: tlustý řádek 92 (textový 23) + 16

; --- Znaky (vlastní glyfy, nahrané při startu; míček/pálka jsou sprity) --------
CH_SPACE    EQU 0x20
CH_WALL     EQU 0x03
CH_BRKL     EQU 0x04        ; levá půlka cihly  (partner je na PTR+1)
CH_BRKR     EQU 0x05        ; pravá půlka cihly (partner je na PTR-1)

; --- Hřiště (prostor tlustých px: x 0..159, y 0..99; zdi = buňky 0/39/řádek 1)
PAD_WIDTH   EQU 24          ; 6 textových buněk
```

```
PAD_MID     EQU 12          ; offset středu míčku při dopadu přesně na střed pálky
PAD_VMAX    EQU 0xC0        ; max. rychlost pálky, Q8.8 px/snímek (0.75 = ~187 px/s)
PAD_ACCEL   EQU 0x10        ; přírůstek rychlosti za držený snímek (max za ~48 ms)
PAD_PMIN    EQU 4           ; mez PAD_POS: přilepená k levé zdi (buňka 1)
PAD_PMAX    EQU 132         ; mez PAD_POS: pravý okraj přilepený k buňce 39
PAD_START   EQU 68          ; na středu: (160 - PAD_WIDTH) / 2
BX_MIN      EQU 4           ; rozsah x míčku (4px míček mezi bočními zdmi)
BX_MAX      EQU 152
BY_MIN      EQU 8           ; pod horní zdí (řádek 1 = tlusté řádky 4..7)
BY_PAD      EQU 89          ; horní okraj míčku, když sedí na liště pálky
BY_LATE     EQU 94          ; níž už ho pálka nechytí
BY_DEAD     EQU 96          ; míček ztracen (textový řádek 24)
BRICKS_N    EQU 108         ; 6 řádků x 18 dvoubuňkových cihel

; --- Rychlost míčku (pevná čárka Q8.8, tlusté px/snímek) -----------------------
VY_MAG      EQU 0x30        ; svislá rychlost 0.1875/snímek = ~47 px/s (~11 buněk/s)
VX_LAUNCH   EQU 0x20        ; boční rychlost výkopu 0.125/snímek = ~31 px/s
VX_MIN      EQU 0x06        ; odraz nikdy strměji než ~6 px/s — žádné kolmice
SERVE_PAUSE EQU 180         ; snímky klidu po každém podání (~0.72 s)
; Snímky po 4 ms (250 fps). Hostitelský displej vzorkuje sprity vlastním,
; nesynchronizovaným tempem a stroj nemá vsync, na který by se zamkl — KAŽDÝ
; celosnímkový krok hry může padnout těsně před repaint, nebo těsně po něm,
; takže při snímkové frekvenci srovnatelné s displejem se drift hodin projeví
; rytmickým škubnutím celé hry. Při 250 fps posune vzorkovací skluz obraz jen
; o čtvrtkrok (pod pixel): tep tam pořád je, jen má amplitudu, kterou oko nevidí.
FRAME_MS    EQU 4

; --- Nultá stránka -------------------------------------------------------------
BX_LO       EQU 0x10        ; poloha míčku, Q8.8 (zlomek, pak celá část)
BX          EQU 0x11
BY_LO       EQU 0x12
BY          EQU 0x13
VX_LO       EQU 0x14        ; rychlost míčku, znaménková Q8.8
VX_HI       EQU 0x15
VY_LO       EQU 0x16
VY_HI       EQU 0x17
OLDY_LO     EQU 0x18        ; BY před pohybem — odraz od cihly se k němu vrací
OLDY        EQU 0x19
SERVE_DLY   EQU 0x1A
PAD_SPD     EQU 0x1B        ; velikost rychlosti pálky, zlomek Q8.8 (0..PAD_VMAX)
PAD_NOW     EQU 0x1C        ; snímek gamepadu v tomto snímku (PAD1 | PAD2)
SCORE       EQU 0x1D        ; zničené cihly, 0..108
LIVES       EQU 0x1E
BRICKS      EQU 0x1F        ; zbývající cihly
PTR         EQU 0x20        ; 16bitový ukazatel do VRAM (dolní 0x20, horní 0x21)
PTR_HI      EQU 0x21
CELL_X      EQU 0x22        ; vstup calc_addr
CELL_Y      EQU 0x23
ZT          EQU 0x24        ; pomocná
PAD_POS     EQU 0x25        ; levý okraj pálky v tlustých px (= čtvrtinách buňky)
PAD_POS_LO  EQU 0x2D        ; jeho zlomkový bajt Q8.8 (podpixelová rampa rychlosti)
SND_DLY     EQU 0x26        ; kolik snímků zbývá běžícímu klouzavému efektu
SND_SLIDE   EQU 0x27        ; 16bit znaménková změna výšky za snímek (dolní, pak
```

```
znaménko)
SND_CH      EQU 0x29        ; offset registrů klouzajícího kanálu (0 nebo 5)
SND_MASK    EQU 0x2A        ; bit gate klouzajícího kanálu
GATES       EQU 0x2B        ; stín AUD_CTRL — bity gate se nastavují/mažou po jednom
EL          EQU 0x2C        ; pomocná wait_frame: uplynulé ms, dolní bajt
ANCHOR      EQU 0x2E        ; snímek CLK_MS při posledním uvolnění snímku (16 bitů)
PAUSED      EQU 0x30        ; 1 = hra zmrazená (mezerník přepíná)
FIRE_PREV   EQU 0x31        ; bit palby minulý snímek — hranový detektor stisku

        .ORG 0x0200

; ============================================================================
; Init — celou statickou obrazovku poskládá na místo Blitter
; ============================================================================
start:  DI
        LDA #0
        STA SCORE
        STA PAD_SPD
        STA PAD_POS_LO
        STA PAUSED
        LDA #1                  ; palbu ber při startu jako "už drženou": stisk,
        STA FIRE_PREV           ;   který hru restartoval, nesmí tu novou
        CALL snd_init           ;   rovnou zapauzovat
        LDA #3
        STA LIVES
        LDA #BRICKS_N
        STA BRICKS
        LDA #PAD_START
        STA PAD_POS

        LDA #0x01
        STA VID_CTRL            ; obraz zapnout, režim 0 (text 40x25)
        LDA #4
        STA VID_BORDER          ; fialový rám skříně automatu
        LDA #25
        STA VID_CURY            ; kurzor odstavit z obrazu (řádek 25 z 0-24)

        ; Statická obrazovka: vyčistit obě roviny, pak namalovat pevné řádky.
        BFILL CHARS, 1000, CH_SPACE
        BFILL COLORS, 1000, 0x00
        BFILL CROW0, 40, INK_HUD    ; barva HUD přežije přepisy číslic
        BFILL ROW1, 40, CH_WALL     ; znaky horní zdi + jejich barva
        BFILL CROW1, 40, INK_WALL

        ; Glyfy zdi/cihel (znaky 3..5) a text HUD, zkopírované přímo
        ; z paměti programu. Glyfy se nahrávají až za BĚHU schválně:
        ; obraz s bajty v glyph RAM by potlačil vestavěné písmo.
        BCOPY glyphs, GLYPH3, 24
        BCOPY hud_score, HUD_SCORE, 9
        BCOPY hud_lives, HUD_LIVES, 7

        ; Boční zdi, sloupce 0 a 39, řádky 2..23. Po každém zápisu znaku se
        ; spočtený ukazatel promění v ukazatel do barev jediným ADD #0x0C
```

```
        ; na horním bajtu (calc_addr ho v dalším kole postaví znovu).
        LDA #2
        STA CELL_Y
.side:  LDA #0
        STA CELL_X
        CALL calc_addr
        LDA #CH_WALL
        STA (PTR)
        LDA PTR_HI
        ADD #0x0C
        STA PTR_HI
        LDA #INK_WALL
        STA (PTR)
        LDA #39
        STA CELL_X
        CALL calc_addr
        LDA #CH_WALL
        STA (PTR)
        LDA PTR_HI
        ADD #0x0C
        STA PTR_HI
        LDA #INK_WALL
        STA (PTR)
        INC CELL_Y
        LDA CELL_Y
        CMP #24
        JNZ .side

        ; Cihlové pole: řádek 3 se píše jako páry L/R přes post-indexovaný
        ; ukazatel (zp),Y, řádky 4-8 jsou klony řádku 3 přes Blitter.
        LDA #<BRICK_ROW3
        STA PTR
        LDA #>BRICK_ROW3
        STA PTR_HI
        LDY #0
.bpair: LDA #CH_BRKL
        STA (PTR),Y
        INY
        LDA #CH_BRKR
        STA (PTR),Y
        INY
        CPY #36
        JC .bpair
        BCOPY BRICK_ROW3, BRICK_ROW4, 36
        BCOPY BRICK_ROW3, BRICK_ROW5, 36
        BCOPY BRICK_ROW3, BRICK_ROW6, 36
        BCOPY BRICK_ROW3, BRICK_ROW7, 36
        BCOPY BRICK_ROW3, BRICK_ROW8, 36

        ; Duha cihel, jedna výplň barvy na řádek (klasický Arkanoid). Atributy
        ; přežívají pod znaky: rozbitá cihla nechá svou barvu na prázdné
        ; buňce, takže ball_step se barevné roviny vůbec nedotýká.
        BFILL CBRICK_R3, 36, 2      ; červená
```

```
        BFILL CBRICK_R4, 36, 8      ; oranžová
        BFILL CBRICK_R5, 36, 7      ; žlutá
        BFILL CBRICK_R6, 36, 5      ; zelená
        BFILL CBRICK_R7, 36, 3      ; azurová
        BFILL CBRICK_R8, 36, 14     ; světle modrá

        ; Sprity: vymazat vzory 0-2 i tabulku atributů, pak nakopírovat
        ; kotouček míčku (vzor 0, řádky 0-3) a lištu pálky (vzory 1+2,
        ; řádky 1-2 — zbytek už je po vymazání průhledný).
        BFILL SPR_PAT0, 384, 0x00
        BFILL SPR_ATTR, 32, 0x00
        BCOPY pat_ball, SPR_PAT0, 32
        BCOPY pat_pad_l, SPR_PAT1R1, 16
        BCOPY pat_pad_r, SPR_PAT2R1, 16
        LDA #1
        STA SPR1_PAT
        LDA #2
        STA SPR2_PAT
        LDA #PAD_SPR_Y          ; pálka svůj řádek nikdy neopouští
        STA SPR1_Y
        STA SPR2_Y
        LDA #0b111
        STA SPR_ENABLE          ; míček + obě půlky pálky zapnout

        CALL serve

        ; --- real-time krokování snímků: nastav lhůtu na "teď". wait_frame
        ; přepočítává délku snímku ze ŽIVÝCH hodin každý snímek, takže změna
        ; taktu uprostřed hry se projeví hned příštím snímkem.
        LDA CLK_MS_LO
        STA ANCHOR
        LDA CLK_MS_HI
        STA ANCHOR+1

        ; Vstup se čte pollingem (gamepad), přerušení se nezapínají.

; ============================================================================
; Hlavní smyčka — jeden průchod za real-time snímek (250 fps: viz FRAME_MS)
; ============================================================================
main:   CALL wait_frame
        CALL snd_frame

        LDA PAD1
        OR PAD2
        STA PAD_NOW

        ; --- Mezerník = pauza. Gamepad je úrovňový, takže přepínač potřebuje
        ; HRANU stisku: palba se počítá, jen když minulý snímek držená nebyla.
        ; Během pauzy běží jen wait_frame a snd_frame — obraz zamrzne na místě
        ; a rozehraný efekt sám dozní.
        AND #BTN_FIRE
        JZ .fup
        LDA FIRE_PREV
```

```
        JNZ .fdn                ; drží se už od minulého snímku — to není stisk
        LDA PAUSED
        XOR #0x01               ; čerstvý stisk: překlop stav pauzy
        STA PAUSED
        CALL pause_note
.fdn:   LDA #1
        STA FIRE_PREV
        JMP .pchk
.fup:   LDA #0
        STA FIRE_PREV
.pchk:  LDA PAUSED
        JZ .input
        JMP main                ; zmrazeno — přeskoč vstup, fyziku i kreslení

        ; --- vstup hráče: PAD_POS je pálka v tlustých px; PAD_SPD roste,
        ; dokud se drží směr, a při puštění spadne na 0 (ostré zastavení,
        ; žádné plavání). Sprity kreslí CELOU podbuněčnou polohu — už žádné
        ; přiskakování k textové mřížce. Řídí se WASD i šipkami.
.input: LDA PAD_NOW
        AND #0x04               ; drží se Vlevo?
        JNZ .left
        LDA PAD_NOW
        AND #0x08               ; drží se Vpravo?
        JNZ .right
        LDA #0                  ; ani jedno: zastav okamžitě
        STA PAD_SPD
        JMP .pdraw
.left:  CALL pad_accel          ; rampuj PAD_SPD směrem k PAD_VMAX
        LDA PAD_POS_LO
        SUB PAD_SPD             ; krok doleva aktuální rychlostí (Q8.8)
        STA PAD_POS_LO
        LDA PAD_POS
        SBC #0
        STA PAD_POS
        CMP #PAD_PMIN
        JNC .pdraw              ; pořád na zdi nebo napravo od ní
        LDA #PAD_PMIN           ; přilep ke zdi
        STA PAD_POS
        LDA #0
        STA PAD_POS_LO
        JMP .pdraw
.right: CALL pad_accel
        LDA PAD_POS_LO
        ADD PAD_SPD             ; krok doprava aktuální rychlostí (Q8.8)
        STA PAD_POS_LO
        LDA PAD_POS
        ADC #0
        STA PAD_POS
        CMP #PAD_PMAX
        JC .pdraw               ; pod maximem -> nech být
        JZ .pdraw               ; přesně na maximu -> nech být
        LDA #PAD_PMAX           ; za maximem -> přilep k mezi
        STA PAD_POS
```

```
        LDA #0
        STA PAD_POS_LO
.pdraw: LDA PAD_POS             ; oba sprity pálky sledují PAD_POS
        ADD #16                 ; hardwarový posun obrazu +16
        STA SPR1_X
        ADD #16                 ; pravá půlka sedí o 16 tlustých px dál
        STA SPR2_X

        ; --- míček: během podání jede na pálce, pak každý snímek fyzika
        ; Q8.8 ---
        LDA SERVE_DLY
        JZ .phys
        LDA PAD_POS             ; drž míček na středu pálky
        ADD #10                 ; PAD_MID - polovina 4px míčku
        STA BX
        LDA #0
        STA BX_LO
        DEC SERVE_DLY           ; DEC v paměti nastavuje Z
        JNZ .draw
        ; klid vypršel — výkop nahoru s náhodným bočním směrem
        LDA #0xD0               ; VY := -VY_MAG (16bitový dvojkový doplněk)
        STA VY_LO
        LDA #0xFF
        STA VY_HI
        RND
        AND #0x01
        JZ .lright
        LDA #0xE0               ; VX := -VX_LAUNCH
        STA VX_LO
        LDA #0xFF
        STA VX_HI
        JMP .draw
.lright:LDA #VX_LAUNCH          ; VX := +VX_LAUNCH
        STA VX_LO
        LDA #0
        STA VX_HI
        JMP .draw
.phys:  CALL ball_step
.draw:  LDA BX                  ; sprite 0 ukazuje celočíselnou polohu míčku
        ADD #16                 ; hardwarový posun obrazu +16
        STA SPR0_X
        LDA BY
        ADD #16
        STA SPR0_Y
        JMP main

; ============================================================================
; wait_frame — real-time omezovač snímků. Drž každý snímek >= FRAME_MS
; reálných milisekund, měřeno čítačem milisekund CLK_MS — ukotveno ke zdi,
; takže snímek trvá stejný REÁLNÝ čas při JAKÉMKOLI taktu (změna taktu
; uprostřed hry se přeladí okamžitě). Při taktech pomalejších než 4ms rozpočet
; hra prostě běží tempem, které logika stihne, přesně jako stará 16ms verze.
; Uplynulé-od-kotvy s novou kotvou při uvolnění: snímek přes rozpočet nenese
```

```
; žádný dluh a nikdy se nemůže přetočit do zaseknutí.
; ============================================================================
wait_frame:
.spin:  LDA CLK_MS_LO           ; uplynulé ms = CLK_MS - ANCHOR (čtení LO zamyká)
        SUB ANCHOR
        STA EL
        LDA CLK_MS_HI
        SBC ANCHOR+1
        JNZ .done               ; uplynulo >= 256 ms — dávno přes rozpočet
        LDA EL
        CMP #FRAME_MS           ; C = 1, dokud uplynulé < FRAME_MS
        JC .spin
.done:  LDA CLK_MS_LO           ; nová kotva v okamžiku uvolnění — snímek
        STA ANCHOR              ;   přes rozpočet nenese žádný dluh
        LDA CLK_MS_HI
        STA ANCHOR+1
        RET

; ============================================================================
; calc_addr — PTR := 0x8000 + CELL_Y*40 + CELL_X  (přepíše A, B)
; Horní bajt MUL je nutné vyzvednout z B *dřív*, než přijde jakákoli binární
; ALU operace: každá z nich (i ADD zp) si v mikrokódu přehraje B svým
; operandem. Přenos z dolního bajtu se přičte až potom podmíněným INC
; (nastavuje jen Z/N).
; ============================================================================
calc_addr:
        LDA CELL_Y
        TAB
        LDA #40
        MUL                     ; A = dolní bajt y*40, B = horní bajt
        STA PTR
        TBA                     ; vyzvedni horní bajt, dokud ho B ještě drží
        ADD #0x80               ; + báze VRAM (přenos nehrozí: horní <= 3)
        STA PTR_HI
        LDA PTR
        ADD CELL_X              ; C = přenos do horního bajtu
        STA PTR
        JNC .done
        INC PTR_HI
.done:  RET

; ============================================================================
; ball_step — pohyb Q8.8, pak řešení: zdi / smrt / cihly / pálka.
; Cihly odrážejí míček "na místě": BY se vrátí na hodnotu před pohybem, takže
; míček do cihly nikdy nezajede. Pálka je čistá aritmetika (žije ve spritovém
; prostoru, ne v mřížce znaků); cihly se čtou z obrazovky jako dřív.
; ============================================================================
ball_step:
        ; --- osa X: 16bitový součet, odraz od bočních zdí ---
        LDA BX_LO
        ADD VX_LO
        STA BX_LO
        LDA BX
```

```
        ADC VX_HI
        STA BX
        CMP #BX_MIN
        JNC .xhigh              ; na levé zdi, nebo napravo od ní
        LDA #BX_MIN
        JMP .xwall
.xhigh: CMP #BX_MAX
        JC .xok                 ; ostře uvnitř
        JZ .xok                 ; přesně na zdi — ještě dobré
        LDA #BX_MAX
.xwall: STA BX                  ; přilep ke zdi...
        LDA #0
        STA BX_LO
        LDA #0                  ; ...a otoč VX (16bitový dvojkový doplněk)
        SUB VX_LO
        STA VX_LO
        LDA #0
        SBC VX_HI
        STA VX_HI
        CALL snd_wall
.xok:
        ; --- osa Y: zapamatuj řádek před pohybem kvůli odrazu od cihel, pak přičti
---
        LDA BY_LO
        STA OLDY_LO
        LDA BY
        STA OLDY
        LDA BY_LO
        ADD VY_LO
        STA BY_LO
        LDA BY
        ADC VY_HI
        STA BY
        CMP #BY_MIN
        JNC .ytop               ; pod horní zdí — v pořádku
        LDA #BY_MIN             ; odraz od horní zdi
        STA BY
        LDA #0
        STA BY_LO
        LDA #VY_MAG             ; VY := +VY_MAG (rovnou zpátky dolů)
        STA VY_LO
        LDA #0
        STA VY_HI
        CALL snd_wall
.ytop:  LDA BY
        CMP #BY_DEAD
        JC .alive
        JMP .lost               ; pod řádkem pálky — míček ztracen

        ; --- cihla v buňce středu míčku (obrazovka je kolizní mapa) ---
.alive: LDA BX
        ADD #2                  ; střed míčku, tlusté px
        SHR
```

```
        SHR                     ; /4: tlusté px -> textová buňka
        STA CELL_X
        LDA BY
        ADD #2
        SHR
        SHR
        STA CELL_Y
        CALL calc_addr
        LDA (PTR)
        CMP #CH_BRKL
        JZ .hit_l
        CMP #CH_BRKR
        JZ .hit_r

        ; --- pálka: aritmetický test překryvu v prostoru tlustých px ---
        LDA VY_HI
        AND #0x80
        JNZ .done               ; letí nahoru — chytit ho nemůže
        LDA BY
        CMP #BY_PAD
        JC .done                ; ještě nad pásmem pálky
        CMP #BY_LATE
        JNC .done               ; zapadl moc hluboko — je pryč
        LDA BX
        ADD #4                  ; pravý okraj míčku
        CMP PAD_POS
        JC .done                ; celý nalevo od pálky
        JZ .done                ; okraje se jen dotýkají
        LDA PAD_POS
        ADD #PAD_WIDTH
        STA ZT
        LDA BX
        CMP ZT
        JNC .done               ; celý napravo od pálky
        ; chycen! — posaď na lištu, pošli nahoru, zamiř podle místa dopadu
        CALL snd_pad
        LDA #BY_PAD
        STA BY
        LDA #0
        STA BY_LO
        LDA #0xD0               ; VY := -VY_MAG (16bitový dvojkový doplněk)
        STA VY_LO
        LDA #0xFF
        STA VY_HI
        LDA BX
        ADD #2                  ; t = střed míčku - levý okraj pálky: 0..25
        SUB PAD_POS
        JNC .tpos
        LDA #0                  ; škrtl o levý roh — přimkni k čelu pálky
.tpos:  STA ZT
        CMP #PAD_MID
        JC .aim_l
        SUB #PAD_MID            ; pravá půlka: velikost 0..13
```

```
        SHL
        SHL                     ; x4 -> boční rychlost Q8.8 (max ~51 px/s)
        CMP #VX_MIN
        JNC .vr
        LDA #VX_MIN             ; i dopad na střed dostane malý drift
.vr:    STA VX_LO
        LDA #0
        STA VX_HI
        RET
.aim_l: LDA #PAD_MID            ; levá půlka: velikost 12-t = 1..12
        SUB ZT
        SHL
        SHL
        CMP #VX_MIN
        JNC .vl
        LDA #VX_MIN
.vl:    STA ZT                  ; otoč znaménko: míček letí doleva
        LDA #0
        SUB ZT
        STA VX_LO
        LDA #0xFF               ; velikost tu není nikdy 0, takže horní = 0xFF
        STA VX_HI
.done:  RET

        ; --- zásah cihly: půlka glyfu prozradí, kde leží partnerská buňka ---
.hit_l: LDA #CH_SPACE
        STA (PTR)
        INW PTR                 ; partner je o buňku napravo
        LDA #CH_SPACE
        STA (PTR)
        JMP .smash
.hit_r: LDA #CH_SPACE
        STA (PTR)
        DEW PTR                 ; partner je o buňku nalevo
        LDA #CH_SPACE
        STA (PTR)
.smash: CALL snd_brick
        INC SCORE
        CALL print_score
        DEC BRICKS              ; DEC v paměti nastavuje Z — podmínka výhry
        JZ .do_win
        LDA OLDY_LO             ; odraz na místě: vrať pohyb v ose Y...
        STA BY_LO
        LDA OLDY
        STA BY
        LDA #0                  ; ...a otoč VY (cihly jsou vodorovné plochy,
        SUB VY_LO               ;    ať jsme přiletěli odkudkoli)
        STA VY_LO
        LDA #0
        SBC VY_HI
        STA VY_HI
        RET
.do_win:
```

```
        JMP win

        ; --- míček ztracen pod pálkou ---
.lost:  CALL snd_lost
        DEC LIVES
        LDA LIVES
        ADD #'0'
        STA LIVES_DIG
        LDA LIVES
        JZ .do_over
        JMP serve               ; koncové volání — RET v serve vrací do main
.do_over:
        JMP game_over

; ============================================================================
; Zvukový engine — tři hlasy, hardwarové obálky. Každý efekt vlastní svůj
; kanál, takže rozbitá cihla už neusekne boing pálky:
;   ch0  odrazy: klik zdi, boing pálky
;   ch1  zavrčení ztraceného míčku a závěrečný tón
;   ch2  šumové perkuse: rozbitá cihla
; Efekt znamená "nastav obálku a výšku, pak gate": o dohasnutí tónu se stará
; tvar ADSR, takže se nemusí nic odpočítávat — kromě dvou efektů, které svou
; výšku záměrně KLOUŽOU (SND_DLY snímků po SND_SLIDE, přičítá snd_frame
; kanálu v SND_CH). Klouže vždy jen jeden efekt; boing a zavrčení se v
; provozu nemohou překrýt.
; ============================================================================
snd_init:
        LDA #0
        STA GATES
        STA AUD_CTRL
        STA SND_DLY
        STA SND_SLIDE
        STA SND_SLIDE+1
        LDA #4                  ; kanál 2 je šumový hlas natrvalo
        STA AUD_WAVE+10
        RET

; A = bit(y) gate. Gate spouští hrana, takže nota se přehraje znovu shozením
; bitu a novým zvednutím — vždy snd_off před snd_on.
snd_on: OR GATES
        STA GATES
        STA AUD_CTRL
        RET

snd_off:XOR #0xFF
        AND GATES
        STA GATES
        STA AUD_CTRL
        RET

snd_frame:
        LDA SND_DLY
        JZ .done
```

```
        DEC SND_DLY             ; DEC v paměti nastavuje Z
        JNZ .slide
        LDA SND_MASK            ; klouzání skončilo — pusť ten hlas
        JMP snd_off
.slide: LDX SND_CH              ; blok registrů klouzajícího kanálu
        LDA SND_SLIDE           ; 16bitový znaménkový součet: freq += slide
        ADD AUD_FREQ_LO,X
        STA AUD_FREQ_LO,X
        LDA SND_SLIDE+1         ; rozšíření znaménka (0x00 stoupá, 0xFF klesá)
        ADC AUD_FREQ_HI,X
        STA AUD_FREQ_HI,X
.done:  RET

snd_wall:                       ; měkký trojúhelníkový klik — odraz od zdi
        LDA #GATE_CH0
        CALL snd_off
        LDA #<600
        STA AUD_FREQ_LO
        LDA #>600
        STA AUD_FREQ_HI
        LDA #2
        STA AUD_WAVE            ; trojúhelník
        LDA #0x04               ; attack 0, decay 4 (~32 ms)
        STA AUD_AD
        LDA #0x02               ; sustain 0 — klik, zmizí sám
        STA AUD_SR
        LDA #GATE_CH0
        JMP snd_on

snd_pad:                        ; stoupající obdélníkový boing — odraz od pálky
        LDA #GATE_CH0
        CALL snd_off
        LDA #<700
        STA AUD_FREQ_LO
        LDA #>700
        STA AUD_FREQ_HI
        LDA #1
        STA AUD_WAVE            ; obdélník
        LDA #0x08               ; attack 0, decay 8 (~90 ms)
        STA AUD_AD
        LDA #0x53               ; sustain 5/15, release 3 — tělo pod klouzáním
        STA AUD_SR
        LDA #20                 ; klouzej nahoru 20 snímků (~80 ms)
        STA SND_DLY
        LDA #10
        STA SND_SLIDE
        LDA #0                  ; kladná změna — znaménko doplní nuly
        STA SND_SLIDE+1
        STA SND_CH              ; registry kanálu 0
        LDA #GATE_CH0
        STA SND_MASK
        JMP snd_on
```

```
snd_brick:                      ; šumový výbuch — rozbitá cihla
        LDA #GATE_CH2
        CALL snd_off
        LDA #<2400              ; rychlý takt LFSR = jasný, krátký sykot
        STA AUD_FREQ_LO+10
        LDA #>2400
        STA AUD_FREQ_HI+10
        LDA #0x03               ; attack 0, decay 3 (~16 ms)
        STA AUD_AD+10
        LDA #0x02               ; sustain 0, release 2 — čistá perkuse
        STA AUD_SR+10
        LDA #GATE_CH2
        JMP snd_on

snd_lost:                       ; dlouhé klesající pilové zavrčení — míček ztracen
        LDA #GATE_CH1
        CALL snd_off
        LDA #<500               ; (~0.4 s — dozní během klidu podání,
        STA AUD_FREQ_LO+5       ; kdy žádný jiný efekt vystřelit nemůže)
        LDA #>500
        STA AUD_FREQ_HI+5
        LDA #3
        STA AUD_WAVE+5          ; pila
        LDA #0x00               ; okamžitý attack, žádný decay
        STA AUD_AD+5
        LDA #0xC5               ; sustain 12/15, release 5 — drží a klouže
        STA AUD_SR+5
        LDA #108                ; klouzej dolů 108 snímků
        STA SND_DLY
        LDA #-2
        STA SND_SLIDE
        LDA #0xFF               ; záporná změna — znaménko doplní jedničky
        STA SND_SLIDE+1
        LDA #CH1_REGS
        STA SND_CH
        LDA #GATE_CH1
        STA SND_MASK
        JMP snd_on

; ============================================================================
; pause_note — ukaž/smaž žlutý nápis "PAUSED" podle příznaku PAUSED.
; Řádek 12 je za hry zaručeně prázdný (cihly končí řádkem 8, míček je
; sprite), takže smazání na mezery + výchozí barvu je vždy správně.
; ============================================================================
pause_note:
        LDA PAUSED
        JZ .clear
        BCOPY msg_pause, MSG_PAUSE_AT, 6
        BFILL CMSG_PAUSE, 6, INK_PAUSE
        RET
.clear: BFILL MSG_PAUSE_AT, 6, CH_SPACE
        BFILL CMSG_PAUSE, 6, 0x00
        RET
```

```
; ============================================================================
; pad_accel — rampuj PAD_SPD o PAD_ACCEL směrem k PAD_VMAX (drží se klávesa)
; ============================================================================
pad_accel:
        LDA PAD_SPD
        CMP #PAD_VMAX
        JNC .done               ; už na maximu (A >= VMAX)
        ADD #PAD_ACCEL
        CMP #PAD_VMAX
        JC .set
        LDA #PAD_VMAX           ; poslední krok dosedne přesně na VMAX
.set:   STA PAD_SPD
.done:  RET

; ============================================================================
; print_score — tři desítkové číslice přes DIV (A / B -> podíl A, zbytek B)
; ============================================================================
print_score:
        LDA #10
        TAB
        LDA SCORE
        DIV                     ; A = skóre/10, B = jednotky
        STA ZT
        TBA
        ADD #'0'
        STA SCORE_ONE
        LDA #10
        TAB
        LDA ZT
        DIV                     ; A = stovky, B = desítky
        STA ZT
        TBA
        ADD #'0'
        STA SCORE_TEN
        LDA ZT
        ADD #'0'
        STA SCORE_HUN
        RET

; ============================================================================
; serve — posaď míček na pálku; jede na ní, dokud klid podání nevyprší
; (samotný výkop udělá hlavní smyčka, až SERVE_DLY dojde na nulu)
; ============================================================================
serve:  LDA PAD_POS
        ADD #10                 ; PAD_MID - polovina 4px míčku
        STA BX
        LDA #0
        STA BX_LO
        STA BY_LO
        LDA #BY_PAD
        STA BY
        LDA #SERVE_PAUSE
```

```
        STA SERVE_DLY
        RET

; ============================================================================
; Obrazovky konce — schovej sprity, nápis, závěrečný tón, pak čekej na
; ČERSTVÝ stisk mezerníku (napřed puštění, aby držená klávesa obrazovku
; nepřeskočila) a restartuj studeným startem: JMP start všechno znovu
; postaví Blitterem. Tón dozní přes busy-wait a před čekáním se umlčí,
; takže obrazovka konce stojí potichu, dokud hráč nerestartuje.
; ============================================================================
game_over:
        LDA #0
        STA SPR_ENABLE          ; bez aktérů hřiště zamrzne
        BCOPY msg_over, MSG_OVER_AT, 9
        BFILL CMSG_OVER, 9, INK_OVER
        LDA #<220               ; hluboké zavrčení
        STA AUD_FREQ_LO+5
        LDA #>220
        STA AUD_FREQ_HI+5
        JMP end_tone

win:    LDA #0
        STA SPR_ENABLE
        BCOPY msg_win, MSG_WIN_AT, 8
        BFILL CMSG_WIN, 8, INK_WIN
        LDA #<880               ; jasný tón
        STA AUD_FREQ_LO+5
        LDA #>880
        STA AUD_FREQ_HI+5

end_tone:
        LDA #3
        STA AUD_WAVE+5          ; pila na hlasu zavrčení
        LDA #0x00               ; okamžitý attack, žádný decay
        STA AUD_AD+5
        LDA #0xF3               ; plný sustain, release 3 — zazní a dozní
        STA AUD_SR+5
        LDA #GATE_CH1
        CALL snd_on
        LDY #30                 ; ~1 s znění (30 x 256 x ~8 T)
.ring:  LDX #0
.spin:  DEX
        JNZ .spin
        DEY
        JNZ .ring
        LDA #0                  ; pusť všechny hlasy — čekání je potichu
        STA GATES
        STA AUD_CTRL
.wrel:  LDA PAD1                ; co se ještě drží, musí napřed nahoru...
        OR PAD2
        AND #BTN_FIRE
        JNZ .wrel
.wprs:  LDA PAD1                ; ...a teprve čerstvý mezerník spustí novou hru
```

```
        OR PAD2
        AND #BTN_FIRE
        JZ .wprs
        JMP start

; ============================================================================
; Data
; ============================================================================
glyphs: ; znak 0x03 — zeď (plný blok)
        DATA 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF
        ; znak 0x04 — cihla, levá půlka (1 px spáry vlevo)
        DATA 0x00, 0x7F, 0x7F, 0x7F, 0x7F, 0x7F, 0x7F, 0x00
        ; znak 0x05 — cihla, pravá půlka (1 px spáry vpravo)
        DATA 0x00, 0xFE, 0xFE, 0xFE, 0xFE, 0xFE, 0xFE, 0x00

hud_score: ; "SCORE 000"
        DATA 0x53, 0x43, 0x4F, 0x52, 0x45, 0x20, 0x30, 0x30, 0x30
hud_lives: ; "LIVES 3"
        DATA 0x4C, 0x49, 0x56, 0x45, 0x53, 0x20, 0x33
msg_over:  ; "GAME OVER"
        DATA 0x47, 0x41, 0x4D, 0x45, 0x20, 0x4F, 0x56, 0x45, 0x52
msg_win:   ; "YOU WIN!"
        DATA 0x59, 0x4F, 0x55, 0x20, 0x57, 0x49, 0x4E, 0x21
msg_pause:
        DATA "PAUSED"

; Řádky vzorů spritů: 8 bajtů na řádek, dva 4bitové pixely na bajt (horní
; půlbajt = levý pixel), půlbajt 0 průhledný. Paleta: 1 bílá, 15 světle
; šedá, 12 šedá.
pat_ball:  ; vzor 0, řádky 0-3 — kotouček 4x4, nasvícený zleva shora
        DATA 0x01, 0x10, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00
        DATA 0x11, 0x1F, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00
        DATA 0x1F, 0xFF, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00
        DATA 0x0F, 0xF0, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00
pat_pad_l: ; vzor 1, řádky 1-2 — lišta pálky, levých 16 px, kulatý levý konec
        DATA 0x0F, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF
        DATA 0x0C, 0xCC, 0xCC, 0xCC, 0xCC, 0xCC, 0xCC, 0xCC
pat_pad_r: ; vzor 2, řádky 1-2 — lišta pálky, pravých 8 px, kulatý pravý konec
        DATA 0xFF, 0xFF, 0xFF, 0xF0, 0x00, 0x00, 0x00, 0x00
        DATA 0xCC, 0xCC, 0xCC, 0xC0, 0x00, 0x00, 0x00, 0x00

        ; RESET vektor zapíše assembler sám (první instrukce = start)
        ; Vektor IRQ není: hra čte vstup pollingem, přerušení nepoužívá.
```

PŘÍLOHA B

Referenční karta

Všechno, na co při stavbě (a přestavbách) hry sáhneš, na čtyřech stranách. Úplné referenční tabulky celého stroje — instrukce, mapu paměti, signály, paletu — najdeš v přílohách „Počítače do dlaně“; tady je jen výřez, který používá Arkanoid.

Zařízení, na která hra sahá

Adresa	Jméno	Význam
`0xFF20`	`AUD_CTRL`	bitmapa gate: bit N spouští/pouští kanál N (reaguje na hranu)
`0xFF21` / `22`	`AUD_FREQ`	frekvence kanálu 0 v Hz, 16 bitů (čitelná — klouzání čte i píše)
`0xFF23`	`AUD_WAVE`	tvar vlny: 0 sinus, 1 obdélník, 2 trojúhelník, 3 pila, 4 šum
`0xFF24`	`AUD_AD`	obálka: horní půlbajt attack, dolní decay
`0xFF25`	`AUD_SR`	obálka: horní půlbajt úroveň sustain, dolní release
—	kanál N	tatáž pětice o `N*5` výš: kanál 1 od `0xFF26`, kanál 2 od `0xFF2B`
`0xFF3A` / `3B`	`CLK_MS`	reálné milisekundy, 16 bitů; čtení dolního bajtu zamkne oba
`0xFF50`	`VID_CTRL`	bit 0 = obraz zapnout (režim 0: text 40 × 25)
`0xFF53`	`VID_CURY`	řádek kurzoru; hodnota ≥ 25 kurzor schová
`0xFF54`	`VID_BORDER`	barva okraje (dolní 4 bity, paleta C64)
`0xFF58`	`SPR_ENABLE`	bit i zobrazí sprite i; při překryvu vyhrává nižší číslo
`0xFF70` / `71`	`PAD1` / `PAD2`	gamepad: bitmapa držených tlačítek, čtení nemaže
`0xFF72` – `77`	`BLT_SRC/DST/LEN`	blitter: zdroj, cíl, délka (vždy dolní + horní bajt)
`0xFF78`	`BLT_FILL`	výplňový bajt pro FILL
`0xFF79`	`BLT_CTRL`	zápis 1 = COPY (jako `memmove`), 2 = FILL; provede se okamžitě

Tabulka 6. Paměťově mapovaná zařízení použitá hrou.

Bity gamepadu: 0 Nahoru, 1 Dolů, 2 Vlevo (`0x04`), 3 Vpravo (`0x08`), 4 Palba (`0x10`, mezerník/Enter), 5 B (Shift), 6 Start, 7 Select. Hra čte `PAD1 OR PAD2`, takže poslouchá WASD i šipky zároveň.

Mapa video paměti

Od	Co	Poznámka
`0x8000`	znaková rovina 40 × 25	adresa buňky = `0x8000 + y*40 + x`
`0x8400`	glyph RAM	8 bajtů na znak; zeď = znak 3, cihly = znaky 4 a 5
`0x8C00`	barevná rovina 40 × 25	horní půlbajt pozadí, dolní písmo; `0x00` = výchozí; znak + `0x0C00`
`0xAFE0`	atributy spritů	4 bajty na sprite: X, Y, vzor, příznaky; souřadnice s posunem +16
`0xB000`	vzory spritů	128 bajtů na vzor: 16 řádků × 8 bajtů, 2 pixely na bajt, 0 = průhledná

Tabulka 7. Výřez video paměti. Tlustý pixel (souřadnice spritů) = 1/4 textové buňky.

Nultá stránka hry

Adresa	Jméno	Význam
`0x10` – `13`	`BX_LO BX BY_LO BY`	poloha míčku, Q8.8 v obou osách
`0x14` – `17`	`VX VY`	rychlost míčku, znaménková Q8.8
`0x18` / `19`	`OLDY_LO OLDY`	svislá poloha před pohybem (odraz od cihly „na místě“)
`0x1A`	`SERVE_DLY`	zbývající snímky klidu podání (nenulové = míček jede na pálce)
`0x1B`	`PAD_SPD`	rychlost pálky, zlomek Q8.8 (rampa 0… `0xC0`)
`0x1C`	`PAD_NOW`	snímek gamepadu v tomto snímku (`PAD1 OR PAD2`)
`0x1D` – `1F`	`SCORE LIVES BRICKS`	skóre, životy, zbývající cihly
`0x20` – `24`	`PTR CELL_X/Y ZT`	ukazatel do VRAM, vstup `calc_addr`, pomocná
`0x25` / `2D`	`PAD_POS(_LO)`	levý okraj pálky v tlustých px + zlomek Q8.8
`0x26` – `2A`	`SND_*`	klouzavý efekt: odpočet, změna/snímek, kanál, gate maska
`0x2B`	`GATES`	stín registru `AUD_CTRL`
`0x2C` / `2E`	`EL ANCHOR`	pomocné `wait_frame`: uplynulé ms, kotva (16 bitů)
`0x30` / `31`	`PAUSED FIRE_PREV`	příznak pauzy, hranový detektor palby

Tabulka 8. Rozvržení nulté stránky hry.

Q8.8 v kostce

Dvoubajtové číslo, dolní bajt = zlomek (1/256). Sčítá a odčítá se obyčejně po bajtech (`ADD` / `ADC`, `SUB` / `SBC`), záporné hodnoty v dvojkovém doplňku, negace = `0 - x` v šestnácti bitech. Při 250 snímcích za sekundu platí převod: **rychlost v px/s = hodnota × 250 / 256**, tedy zhruba hodnota × 0,98 — Q8.8 hodnota je skoro přesně rychlost v pixelech za sekundu.

Konstanta	Q8.8	px/s	Role
`VY_MAG`	`0x30`	~47	svislá rychlost míčku
`VX_LAUNCH`	`0x20`	~31	boční rychlost výkopu
`VX_MIN`	`0x06`	~6	nejmenší povolený boční drift
max. míření	`0x34`	~51	dopad na kraj pálky
`PAD_VMAX`	`0xC0`	~187	nejvyšší rychlost pálky
`PAD_ACCEL`	`0x10`	—	přírůstek rampy za snímek (max za 12 snímků)

Tabulka 9. Rychlostní konstanty hry na jednom místě.

Barvy hry

Z šestnáctibarevné palety (příloha D první knihy) hra používá: 2 červená, 3 azurová (HUD a cihly), 4 fialová (okraj), 5 zelená, 7 žlutá (PAUSED a cihly), 8 oranžová, 10 světle červená (GAME OVER), 12 šedá (zdi a tělo pálky), 13 světle zelená (YOU WIN!), 14 světle modrá, a ve vzorech spritů 1 bílá a 15 světle šedá.