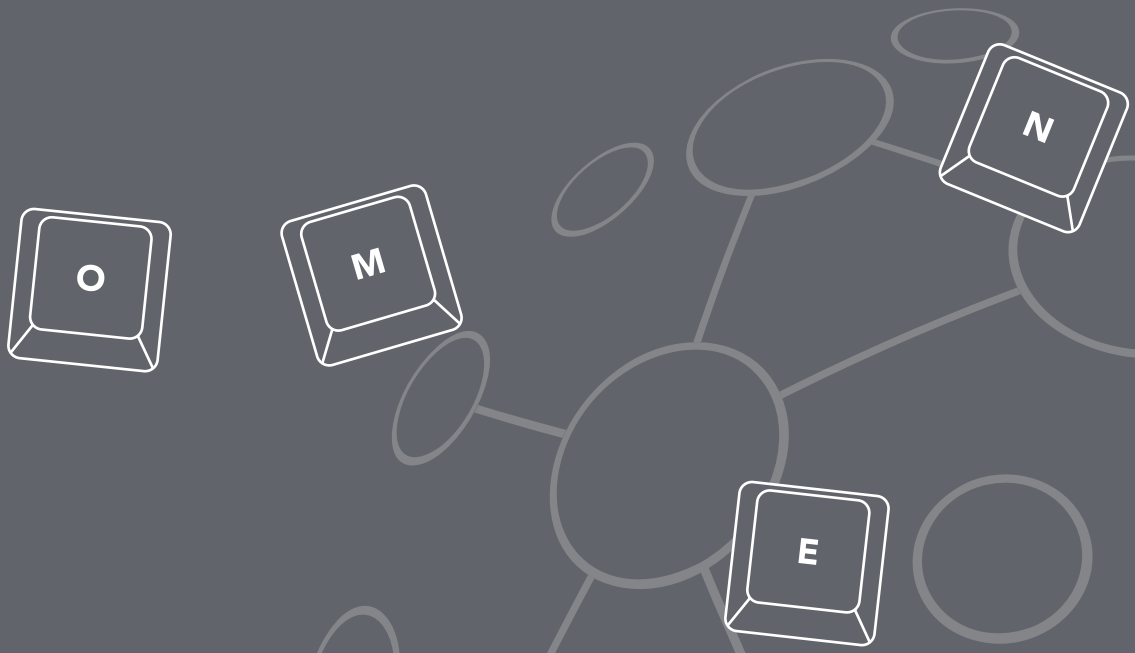
Martin Malý

Porty, bajty, osmibity

Počítače na koleni



Edice CZ.NIC

PORTY, BAJTY, OSMIBITY
Počítače na koleni

Martin Malý

Vydavatel:
CZ.NIC, z. s. p. o.
Milešovská 5, 130 00 Praha 3
Edice CZ.NIC
www.nic.cz

1. vydání, Praha 2019
Kniha vyšla jako 21. publikace v Edici CZ.NIC.
ISBN 978-80-88168-39-3

CZ.NIC | SPRÁVCE DOMÉNY CZ

— Martin Malý

Porty, bajty, osmibity

Počítače na koleni

— Edice CZ.NIC

Poděkování

Poděkování

Děkuji svému nakladateli za důvěru a péči, věnovanou této knize.

Děkuji všem, kteří mi s psaním knihy fandili. Dodávali mi sílu, když bylo psaní nekonečné.

Děkuji všem, co přečetli rukopis a přispěli poznámkami. Díky nim je kniha o něco jasnější a srozumitelnější.

Děkuji designérům Davidu Švejdovi a Jakubu Goldmannovi za návrh a vypracování grafických inzerátů ve stylu počítačových časopisů z 80. let.

Děkuji Míše, že se mnou proces psaní vydržela. Už podruhé.

Předmluva vydavatele

Předmluva vydavatele

Vážení čtenáři,

dostává se vám do rukou druhá kniha od Martina Malého, která je, dá se říct, volným pokračováním jeho publikace Hradla, volty, jednočipy. Jak napovídá již název této knihy – Porty, bajty, osmibity, autor popisuje své zkušenosti s osmibitovými mikrokontrolery a počítači a pokouší se je předat svým čtenářům.

Osobně jsem se s osmibitovými mikrokontrolery, konkrétně s PIC 16F84 a Atmel 8051, poprvé setkal v druhém ročníku na střední škole. Dodnes si pamatuji, jak jsme zápasili se vstupními a výstupními porty (a několik jich přitom poslali do křemíkového nebe), snažili se pochopit přímé a nepřímé adresování a nakonec i rozběhnout resetovatelné stopky s pamětí za pomoci sedmisegmentových displejů a externí paměti.

Většina úloh by se dnes dala přirovnat k dětskému hraní, ale nás to bavilo a pro mnohé to byla neocenitelná lekce, která nám poskytla základy pro naši další práci při vývoji hardwarů i softwarů. V dnešní době si samozřejmě můžete koupit Arduino a začít programovat v „céčku“ nebo jiném vyšším jazyce. Budete tak ale ochuzeni o úplné základy, nastavování jednotlivých bitů registrů a psaní kódu v assembleru. Ze své praxe mohu říct, že se vám to může někdy dost hodit.

Právě tato kniha vám může přiblížit svět osmibitových mikrokontrolerů a počítačů. Svět, ve kterém si musíte vše udělat a ošetřit sami, ale odměnou vám bude váš vlastní kus hardwaru, na kterém bude běžet váš program. A to je pocit k nezaplacení.

Přeji všem hezkou četbu a mnoho úspěchů při tvorbě vlastních elektronických aplikací.

Petr Bílek, CZ.NIC
Praha, 14. února 2019

Předmluva

Předmluva

Nevím jak vy, ale já relaxuju u her. Sednu si na gauč, nebo lehnu do postele, vezmu do ruky ovladač od XBoxu nebo PlayStationu, nahraju nějakou hru a hraju.

Po čase hraní, když už se nemusím soustředit na to, jaká tlačítka mám mačkat a co mám přesně dělat, se mi ve volných chvílích, když moje postava někam běží nebo jede nebo poslouchá nekonečně dlouhé nepřeskočitelné monology jiných postav, v hlavě rozběhnou úplně jiné myšlenky.

Většinou si říkám, že bych měl dodělat ten engine na psaní textových her, co mám rozdělaný už dva roky, pak chvilku přemýšlím nad tím, co tam ještě dodělat, chvilku si představuju, jak to bude hezké, až to bude hotové, pak si říkám, že bych měl taky dodělat ten kurz Unity, co mám rozkoukaný a zaplacený, no a konečně mi fantazie sklouzne k tomu, že bych vlastně chtěl mít ten svůj osmibit a pro něj si napsat pár her.

Abychom si rozuměli – vlastních osmibitů mám plné skříně: Spectra, Commodory, Atari, i nějaké podivnější a obskurnější značky a typy, ale já bych rád *vlastní*. Jako že víc vlastní. Můj. Mnou navržený a postavený.

Ano, v žebříčku kutilských projektů je myšlenka „navrhnout si a postavit vlastní osmibitový počítač a psát pro něj programy" někde hodně… no… jak by řekla televizní věštkyně Jolanda: *Je to hodně někde!* Obvykle kutilský projekt aspoň předstírá, že k něčemu bude, že bude plnit nějakou užitečnou funkci. Že třeba bude měřit teplotu a hlídat garáž a zalévat květiny. Ovšem vlastní osmibitový počítač, to je něco *tak moc samoúčelného*, jak vám na požádání vysvětlí každý racionálně uvažující technik.

Jenže když jste amatér, tak víte, že si prostě nemůžete pomoct. Vy máte ty věci fakt rádi! A chcete je dělat, i když racionálno říká: *Budeš jediný s takovým počítačem. No dobře, budete možná tři: Ty a dva blázni z internetu, co si ho postavěj' taky. Tolik času tím strávíš, tolik času spálíš programováním, a k čemu to bude? No ne, vážně: K čemu to bude?*

Je to úplně stejná otázka, jakou slyší každý hráč her. K čemu to je? A myslím, že i odpověď bude stejná: je to pro nás zábava, příjemná činnost, a smysl to má jako každý jiný podobný koníček. Děláme něco, co máme rádi, a baví nás to. Něco se při tom naučíme, získáme nějaké ty dovednosti, ale hlavně: je to mnohem lepší zábava, než civět na televizi.

Gordon Freeman v té mojí televizi právě prochází Ravenholmem. Když jsem hrál tuhle část Half-Life 2 poprvé, úplně jsem ji nenáviděl, nedokázal jsem ji projít, pořád mě něco někde zabíjelo. Dneska už to hraju tak zlehka, zkouším různé přístupy, jako „ani jeden výstřel" nebo „co nejrychleji k cíli" nebo „tuhle pasáž projdu extrémně opatrně". Jak říkám – relax pro hlavu, takže se tak na dvacet procent věnuju hře, a zbytek vědomé kapacity se, nebrzděn racionalitou, oddává radostným představám, v nichž tančí úplně amatérské počítače, ze kterých čouhají dráty a smrdí křemík,

a přemýšlí nad tím, jestli bude lepší implementovat BASIC, nebo FORTH, a jestli bude lepší procesor 6502 nebo Z80 – ten je sice známější, ale zase složitější. A co třeba 6809?

Výstup udělám na televizi, nebo na nějaký miniaturní displej? A jak to namapuju?

A co klávesnice? Tak, PS/2 jde vždycky, ale co nějaká víc *vintage*? Co třeba vzít starou od nějakého Commodora, na eBay jich jsou spousty. Nebo co třeba vzít nějakou starou českou Consuláckou? Nebo co si udělat vlastní membránovou, jako mělo JPR-1? Nebo vlastní z tlačítek? A na 3D tiskárně vytisknout hmatník?

Nebo konzoli! Normálně herní konzoli, čtyři tlačítka do kříže, dvě akční, mezi to displej… A emulátor si k tomu udělám, aby se snadno vyvíjely programy! Na 3D tiskárně kryt vytisknu. Anebo použiju joystickový modul. Jako procesor klidně Arduino Nano, udělám si na to plošný spoj…

Po chvíli hru vypnu a z ničeho nic kreslím plošný spoj a posílám ho do výroby. Hrozně se na to těším, bude to fajn, budu mít herní konzoli, kterou nemá nikdo jiný, a nebude pro ni jediná hra!

STOP!

Pokud vám to připadá jako ten největší nesmysl, přestaňte číst právě teď, knihu někomu věnujte, odkaz smažte, zapomeňte… Není to pro vás.

Ale pokud si říkáte „chci taky jeden plošný spoj, budeš mít navíc?" – jste moji čtenáři! Pojďte dál, řekneme si něco o starých procesorech, o osmibitových počítačích, o tom, jak se vlastně programují, jak fungují uvnitř, ukážeme si, jak sestavit počítač v duchu těch starých časů, ukážeme si i pár modernějších triků, jako jsou malé barevné displeje, a hlavně: pár kousků si postavíme! A budou to počítače jako byly kdysi: s výstupem na terminál, černobílé, s malou hroznou klávesnicí a pípátkem na zvuk! Pojďte, pojďte!

Konvence

V této knize se podržím následujících konvencí:

- Hexadecimální hodnoty budu v textu zapisovat ve tvaru ABCDh – tedy bez počáteční nuly a se suffixem „h". Tam, kde to bude mít speciální význam, např. u výpisu v jazyce C, použiju samozřejmě odpovídající zápis 0xABCD. U výpisu zdrojového kódu budu zase používat variantu s počáteční nulou.
- Pro negované signály budu používat úvodní lomítko, tedy např. /RESET. Nebudu používat ani nadtržení, ani suffix „B", ani prefix či suffix „n".

Obsah

1 Slavné domácí osmibity 80. let

1 Slavné domácí osmibity 80. let

1.1 Domácí počítač

Čím jiným začít tuto knížku, než vyprávěním o slavných hvězdách let dávno minulých, z dob, kdy každý domácí počítač měl jiný systém a kompatibilita byla nanejvýš mezi sousedními modely jedné řady jednoho výrobce. Pravděpodobně jste některý z těchto počítačů měli doma, a možná ho stále máte. Bylo to Spectrum? *Gumák,* nebo *Plusko*? Nebo *Atárko*? Nebo snad Commodore, dokonce s disketovkou? Nebo Sharp, či snad Sord?

Na začátku 70. let stále ještě platilo, že počítač = sálové zařízení. Nejnovější výkřiky techniky pak byly velké jako skříň. Jejich srdcem byly procesory, postavené ze samostatných modulů, většinou na jedné desce.

Procesor, coby základní řídicí jednotka, obsahuje jednak část, která řídí běh programu a vykonávání instrukcí (*řadič*), a jednak část, která zpracovává data (*aritmeticko-logická jednotka*, ve zkratce ALU, a k ní sada registrů, v nichž se data uchovávají při zpracování). K procesoru byla připojena paměť, v níž je uložený program a zpracovávaná data, a taky nějaké periferie, což byly všechny ty terminály, klávesnice, snímače děrných pásek a štítků, magnetopáskové jednotky, diskové jednotky…

Ve stejné době vznikly první mikroprocesory, tedy integrované obvody, které na jednom čipu (či několika málo čipech) obsahovaly řadič instrukcí, výkonnou jednotku, pracovní registry i aritmeticko-logickou jednotku. Poměrně rychle vznikly obvody 4004, 8008 a 8080 (vše Intel). Od vzniku procesoru 8080 se vlastně dá mluvit o začátku mikroprocesorové éry.

Intel nebyl samozřejmě sám, vznikaly i další procesory, a po poklesu cen a příchodu mikroprocesorů 6502 a Z80 začaly vznikat první počítače, které se vešly na stůl. Kromě profesionálních zařízení přišla i zařízení pro amatérské použití (Altair 8800) nebo dokonce plně amatérské konstrukce.

Amatér je samozřejmě slovo, které má hanlivý nádech, ale nezapomínejme, že jeho původ je ve slově *amare*, tedy milovat. Amatér je tedy vlastně člověk, který se oboru věnuje nikoli proto, že by ho živil, ale proto, že ho má rád. A v tomto významu prosím čtěte i výše zmíněné „amatérské konstrukce".

První amatérské počítače se prodávaly jako stavebnice: deska tištěných spojů, sada součástek, spájejte si doma, IKEA hadr. Přesně v tomto stylu začala vyrábět své počítače garážová firma Apple – model

Apple I byl právě taková stavebnice, kterou jste si za 500 USD koupili a spájeli doma (a pokud jste si ji schovali až dodneška, má hodnotu několik desítek tisíc dolarů).

Brzy ale přišly kompletní sestavené stroje, které jste si přinesli z obchodu, zapojili – a fungovaly. Nejslavnější jména té doby (hovoříme o konci 70. let) jsou Apple II, Commodore PET a TRS-80.

Obrázek 1: Apple I

Apple II se neprodával tak dobře jako další (například v jeho BASICu chyběla podpora pro desetinná čísla), ale prodával se delší dobu, takže úhrnem až do roku 1993, kdy se přestal vyrábět, prodala firma Apple přes čtyři miliony kusů tohoto počítače.

Commodore PET přišel s ikonickým designem plechové krabice se zabudovanou klávesnicí, kazetovým magnetofonem a černobílým displejem.

Obrázek 2: Commodore PET

Commodore oznámil poměrně brzy nové modely, a tak se různých modelů PETu prodal necelý milion.

Třetí slavný počítač té doby, TRS-80, používal procesor Z80, na rozdíl od předchozích dvou, které byly poháněné procesorem 6502. Zkratka TRS odkazuje na výrobce (Tandy) a distributora tohoto počítače (Radio Shack).

Obrázek 3: TRS-80

1.2 Druhá generace

Do druhé generace domácích počítačů promluvily i další firmy. V Atari vzali hardware svých herních konzolí, přidali k němu klávesnici, a tak vznikla slavná řada domácích osmibitů od Atari (400, 800, série XL, série XE). Commodore přepracovalo PET a vznikly počítače VIC-20 a velmi úspěšný Commodore 64.

V Británii sir Clive Sinclair, který věřil kalkulačkám a směřoval spíš k vědeckému použití vlastních vynálezů, vlastně jako vedlejší produkt uvedl na trh nejprve počítačovou stavebnici MK14 s méně známým procesorem SC/MP (INS8060) a později slavnou sérii počítačů ZX: ZX80, ZX81 a ZX Spectrum.

Atari a Sinclair patřily k nejrozšířenějším počítačům v tehdejší ČSSR. U Atari to byly především modely 800XL a 130XE, od Sinclaira pak ZX81, ale především Spectrum (i Spectrum Plus, vídané i pod značkou „Delta“). Méně bylo Commodorů, Specter 128, počítačů Amstrad, Sharp MZ800 nebo Sord m5.

O počítačích ZX asi nemá smysl v české knize pro českého čtenáře psát podrobněji. Přesto alespoň pár slov.

Počítač ZX80 přišel s jednoduchou konstrukcí: deska včetně membránové klávesnice, paměť, procesor a několik integrovaných obvodů, které se spolu s mikroprocesorem staraly o zobrazování textu na černobílé televizi. Sinclairovi konstruktéři vymysleli velmi důmyslný trik, kterým dokázali zobrazovat text na televizi s minimem potřebných součástek. Na druhou stranu za to zaplatil uživatel, protože procesor většinu času sloužil jako čítač adres pro zobrazování, a pouze ve chvílích, kdy se nic nezobrazovalo, mohl počítat uživatelské úlohy.

Následovník ZX81 přinesl zásadní změnu. Koncepce zůstala stejná, ale naprostá většina integrovaných obvodů byla nahrazena jedním na míru vytvořeným zákaznickým čipem ULA SCL (Sinclair Custom Logic). Počítač tak obsahoval pouhých pět integrovaných obvodů – procesor, paměť RAM (2 čipy), paměť ROM a SCL. Díky tomu byl ZX81 menší a levnější. Jinak zůstal, snad s výjimkou drobného rozdílu v přerušovacím systému, funkčně shodný s předchůdcem.

Ale protože konkurence oznamovala počítače s barevným výstupem, tak i u Sinclairů připravovali vlastní barevný stroj. Dostal název ZX Spectrum a vývojáři vyšli z toho, co dobře znali.

Použili stejnou koncepci klávesnice (jen přes ni dali gumovou masku s tlačítky, která vysloužila Spectru přezdívku „gumák“), použili stejný procesor, zvětšili paměť z 8 kB ROM na 16 kB ROM, v základu místo jednoho kilobajtu statické RAM bylo 16 kilobajtů dynamické RAM, a konečně přepracovali zákaznický obvod tak, že dokázal generovat barevný obraz. Tentokrát už bez přispění procesoru, takže procesor mohl pracovat bez ohledu na potřeby zobrazování. Pouze pokud přistupoval k paměti ve stejné paměťové oblasti, jako byl displej, tak byl mírně zpomalován.

Kromě tří modelů počítačů ZX, z nichž každý je legendou sám o sobě, přispěl Sinclair k boomu domácích počítačů i zprostředkovaně: Sinclairův zaměstnanec Chris Curry nebyl spokojen s tím, jak Sinclair vede (spíš nevede) vývoj osobních počítačů, a tak odešel a založil si vlastní firmu Acorn. Po prvním jednodeskovém Acorn System 1 přišel počítač Acorn Atom, a po něm slavný počítač Acorn Proton, známější jako BBC Micro (britská veřejnoprávní síť BBC totiž vybrala tento počítač jako výukový počítač pro svůj kurz programování a číslicové techniky).

Po BBC Micro přišla i jeho levnější varianta Acorn Electron, ovšem pak nastala krize v polovině 80. let, mnohé značky nepřežily a Acorn se udržel jen díky svému novému počítači Archimedes a vlastnímu procesoru, nazvanému Acorn RISC Machine, ve zkratce ARM. Ten je tu s námi dodnes, i když v notně vylepšené podobě.

V polovině 80. let skončila éra domácích osmibitů: Commodore ani Atari už nepředvedly významnější zlepšení osmibitové série a představily své šestnáctibitové stroje Amiga a ST. Sinclair přechod na vícebitovou architekturu urychlil tak moc, že jeho Sinclair QL předběhl dobu, zaznamenal neúspěch a Sinclair nakonec svou značku prodal konkurentům od Amstradu (kteří do té doby vyráběli počítače řady CPC).

Obrázek 4: Acorn Proton, alias BBC Micro

1.3 Další slavní…

Sinclairovi lidé, kteří od něho na začátku 80. let odešli, mají na svědomí nejen firmu Acorn, ale třeba i počítač Jupiter Ace.

Obrázek 5: Jupiter ACE (fotografie autora Factor-h pod licencí CC-BY-SA)

Tento počítač připomíná na první pohled ZX Spectrum. Uvnitř je ale podobný spíš ZX81. Nejzajímavější na něm je fakt, že na rozdíl od ostatních počítačů té doby neměl Jupiter zabudovaný programovací jazyk BASIC, ale používal jazyk FORTH. A zase: pokud máte doma tento počítač, má hodnotu několika desítek tisíc.

Zatímco Acorn jsme tu aspoň trošku znali, jméno Tangerine Computer Systems znělo povědomě snad jen těm největším fandům. Možná někdo někde zahlédl název ORIC, nebo fotografii…

Přitom první počítač této firmy, nazvaný ORIC-1 a uvedený v roce 1983, rozhodně měl šanci zaujmout. Dodával se ve verzi s 16 kB RAM nebo s 48 kB RAM, uvnitř ho poháněl osvědčený 6502A, taktovaný na 1 MHz. Na rozdíl od Spectra 48 obsahoval speciální zvukový čip – známý AY-3-8912 (později použitý ve Spectru 128). Samozřejmostí byl zabudovaný BASIC…

I v ORICu byl použit zákaznický obvod ULA, který se staral o zobrazování ve dvou grafických módech, LORES a HIRES.

V Tangerine připravili vylepšené verze Atmos, Stratos a Telestratos, ale nakonec i je dostihl konec osmibitové éry. Jako zajímavý moment dodám, že firma vyvezla zhruba tisíc počítačů Atmos do tehdejší Jugoslávie a že v Bulharsku vyráběli klon Atmosu pod označením Pravec 8D.

Firma Tandy spolu s prodejcem RadioShack pod značkou TRS pokračovala v úspěšné sérii domácích počítačů, zahájené modelem TRS-80, a po několika modelových sériích tohoto černobílého počítače přišla s barevnou verzí, nazvanou „TRS-80 Color Computer“, zkráceně CoCo. V tomto typu opustila procesor Z80 a místo něj stroje CoCo poháněl procesor Motorola 6809. Což tedy znamenalo hlavně programovou nekompatibilitu s předchozími stroji. Počítače CoCo se dočkaly tří modelů (CoCo, CoCo 2 a CoCo 3), ten poslední se vyráběl až do roku 1991. Časem se pro ně objevily i operační systémy FLEX9 a OS-9 se schopnostmi lehce přesahujícími tehdejší DOS.

Obrázek 6: TRS-80 Color Computer (autor fotografie: Bilby, CC-BY)

Firma Dragon vyráběla počítače Dragon 32 a Dragon 64, které byly koncepcí i schopnostmi velmi podobné počítači CoCo. Hlavní rozdíl byl ten, že byly britské, a tedy pracovaly se systémem PAL (nikoli NTSC). Britský tisk je označoval za „lepší než jejich britští konkurenti, ale celkem nic moc" a trochu nespravedlivě psal, že „je to levnější klon CoCo s lepší klávesnicí".

Obrázek 7: Dragon 32

1.4 MSX

Standard MSX byl logickou reakcí na spoustu platforem, co v té době existovala. Dneska máme v zásadě dvě hlavní platformy a tři systémy, přičemž ty platformy můžou poměrně bez problémů sdílet navzájem periferie, dokonce i některé komponenty, takže *piánko*. Ale představte si začátek 80. let, co výrobce, to vlastní nová geniální platforma, často nekompatibilní ani s předchozím modelem… Ti velcí, co se chytli, měli štěstí: Atari, Commodore, Sinclair. Ale co ti ostatní, kterým trh domácích počítačů ujížděl?

Na trhu profesionálních sestav bylo relativně dobře, tam se vždycky našel nějaký společný jmenovatel, třeba CP/M. Výrobci stačilo udělat stroj s disketovou mechanikou a BIOSem a systém se už nějak podařilo naportovat. Ale pro domácí počítače byly diskety v tu dobu zatím pořád drahé.

S řešením přišel Microsoft, firma, která v té době ještě nebyla nenáviděný symbol korporátní arogance a terč k diskusním jedovatostem pro každého juniorního vývojáře, ale především vývojář

na tehdejší dobu velmi kvalitních interpreterů BASICu pro všemožné platformy. (Ostatně právě tohle nakonec donutilo Gatese kývnout na poptávku IBM – kdyby PC nemělo systém, Gates by nemohl prodávat své programovací jazyky pro PC, tam viděl hlavní těžiště obchodního modelu MS+PC) No a právě Microsoft se tehdy spojil s několika (převážně japonskými) výrobci elektroniky a dohromady navrhli „standard MSX".

Vize byla jednoduchá: Každý počítač, který bude mít logo MSX, bude splňovat určitou minimální sadu požadavků, což zajistí, alespoň teoretickou, kompatibilitu software a části hardware. Počítače budou mít stejný procesor (Z80A) na stejné frekvenci (3,58 MHz), budou mít stejný grafický řadič (TMS9918, popř. ekvivalent pro PAL systém TMS9929), zvukový čip AY-3-8910, PIO 8255, na něm budou připojené standardizovaným způsobem některé interní periferie (např. přepínání paměti), a tak dál. Tedy v podstatě něco jako PC, kdyby PC bylo od začátku výsledkem dohody několika firem. A bylo na výrobcích, kolik použijí paměti (předepsáno bylo minimálně 16 kB RAM), jakou použijí klávesnici, jaké pouzdro zvolí, jaké příslušenství dodají… O základní programové vybavení se nemuseli starat: měli od Microsoftu MSX BASIC.

Obrázek 8: Počítač standardu MSX2 od Daewoo

Výhody tohoto přístupu jsou jasné a pro nás dnes už samozřejmé: výrobce mohl vyrábět třeba jen cartridge nebo software a měl jistotu, že budou fungovat na nejrůznějších modelech od nejrůznějších výrobců (ve skutečnosti měl spíš „naději" než „jistotu", ale dejme tomu). Zákazník si mohl koupit počítač MSX od libovolného výrobce podle svých osobních preferencí a věděl, že mu budou fungovat moduly a programy od jiných výrobců. Mohl upgradovat bez obav z toho, že na novém systému nebudou staré programy fungovat. A tak dál.

Jenže: přídavné moduly měly občas problémy při fungování s jinými. Nebyly mechanicky dobře vyřešené, takže těžké moduly občas vypadly z konektoru. Výrobci poměrně rychle začali nabízet modely s velkou pamětí, čemuž se přizpůsobili tvůrci software, a tak jste si některé hry na slabším modelu nespustili.

Ale to všechno byly jen prkotiny proti tomu největšímu problému: Počítače MSX vyráběla spousta výrobců, převážně japonských. Prodávaly se v Japonsku velmi dobře. Ale ve světě už moc ne. Ve Spojených státech a v Evropě, kde se rozhodovala bitva o domácí počítače, vládla trojice Commodore, Atari, Sinclair a MSX se tu nikdy významně nerozšířil (i když jedním z výrobců byl Philips) – snad jen s výjimkou Španělska a Nizozemí. Standard MSX se kupodivu ujal v arabských zemích a zemích Latinské Ameriky. No a nemalá část těchto počítačů, okolo milionu kusů, se prodala v – Sovětském Svazu! Tam je nakoupili do škol (embargo COCOM se na tuto kategorii strojů, na rozdíl třeba od strojů s procesorem MC68000, nevztahovalo).

Takže co tu máme? Kvalitně navržený osmibit, s velmi dobrým (na tu dobu) grafickým čipem, s procesorem, co „všichni známe", se spoustou paměti, standardizovaný, s velmi slušným zadrátovaným BASICem, spousta modelů, hmmm… jak by to mohlo neuspět?

Obrázek 9: Sony HitBit

A vidíte – mohlo! Postupně vznikly standardy MSX2, MSX2+ a MSX TurboR, které přidávaly práci s disketami, lepší grafické a zvukové možnosti, TurboR dokonce mnohonásobně rychlejší procesor R800, ale to už byla labutí píseň – Commodore navrhl Amigu, v Atari vzniklo ST a zbytek světa pomalu šel cestou „PC a kompatibilní". První stroje MSX vyráběly desítky výrobců, namátkou Canon, Daewoo, Fujitsu, Goldstar, Hitachi, JVC, Kawai, Panasonic, Philips, Samsung, Sanyo, Sharp, Sony, Toshiba nebo Yamaha. Všechno známé značky. Postupem času většina odpadla a u posledního standardu zůstala jen trojice Panasonic, Sanyo a Sony.

V roce 1986 vyšel v Amatérském Radiu (AR A, čísla 3 a 4) dvoudílný popis standardu MSX. Dneska už působí trošku archaicky, hlavně některé výrazy a formulace (vzpomeňte si: bylo tažení proti cizím slovům v elektronice, a tak se vymýšlely magnetoskopy, styky, styková rozhraní a podobné krkolomnosti). Z popisu funkce „STRING$ vydá řetězec na základě čísla“ dneska už fakt nejsem moudrý, ale upřímně: Zaplať pámbu za ty články!

1.5 A u nás?

Pokud jste v té době žili v ČSSR a měli jste domácí počítač, byl to pravděpodobně Sinclair, Commodore, Atari, Sord, Sharp – tedy zahraniční stroj. Ne že by se v ČSSR žádné domácí počítače nevyráběly. Vyráběly, ale nedaly se moc koupit. Kvůli nejrůznějším problémům se často nedostaly ani do škol nebo do zájmových klubů.

Československo tehdy vyrábělo hned několik strojů, které sem tak trochu spadají. Kromě školních jednodeskových počítačů PMI-80 a TEMS80-03 to byly především počítače PMD-85 a IQ-151. Ten druhý by se dal zařadit spíš do kategorie „neblaze proslulý“ a plně spadá do kategorie „chtěli jsme to udělat co nejlíp, ale dopadlo to jak vždycky“. Navržený byl jako modulární systém, ovšem zajímavý návrh zabily problémy se zdrojem (takže první verze fungovala maximálně se dvěma přídavnými kartami, následující verze naopak topila jak sporák) a naprosto příšerná klávesnice, o kterou jste si lámali prsty.

Naštěstí první jmenovaný, PMD-85, to alespoň trochu zachraňoval, a i když měl taky své mouchy, tak aspoň fungoval a netopil. Později vznikly další dva modely, s kvalitnější klávesnicí a dokonce i s barevným výstupem. A nejen to, vyráběly se i kompatibilní stroje, například počítač Maťo (což byla stavebnice s menší klávesnicí, ale do jisté míry kompatibilní) nebo počítače Consul ze Zbrojovky Brno (kde se pamětníci dodnes brání tomu, že by PMD zkopírovali).

Další klon PMD vyrábělo družstvo Didaktik Skalica pod názvy Didaktik Alfa a Didaktik Beta. Třetí do série, Didaktik Gama, byl taky klon, ovšem už ne PMD, ale naopak známého a rozšířeného ZX Spectra.

Tak trochu stranou stál tým Eduarda Smutného, českého konstruktéra počítačů, který po úspěšných průmyslových JPR-12 a JPR-1 (základ sestavy SAPI-1) zkonstruoval i obdobu Sinclairu ZX-81 pod názvem Ondra. Výrobce Ondry sliboval jednoduchý a dostupný počítač pro děti, bohužel výroba vázla a vázla, a když se rozjela, svět už byl dál.

1.6 Jednodeskové počítače

Jednodeskáč, tedy celým jménem „jednodeskový mikropočítač“, byl v 70. a 80. letech malý počítač, co se vešel na jednu desku. Dnes se na jednu desku vejde leccos, ale tehdy jste si moc vyskakovat

nemohli. Většinou se tam vešel procesor, trocha paměti (typicky 1 kB RAM, 1 kB PROM), klávesnice (měla většinou okolo 20-25 tlačítek: 16 znaků 0-9 a A-F pro zadávání hexadecimálních číslic a několik ovládacích), displej (ze sedmisegmentovek) a nějaké vstupně/výstupní porty, které byly buď dostupné pro další rozšiřování, nebo na nich byly různé LEDky, motorky a podobně.

V Československu se vyráběly dva známější jednodeskáče (a několik dalších, dnes už pozapomenutých): legendární PMI-80 a školní jednodeskový počítač TEMS 80-03A. Oba s procesorem 8080A a s plus mínus stejnou technickou výbavou. Několik jednodeskových počítačů vzniklo i péčí nadšenců (BOB85, SAVIA84), vznikaly i jako stavebnice (Petr)… Technicky vzato byl třeba i Ondra jednodeskový, ale pro nás budiž jednodeskáč počítač s omezenou klávesnicí a s displejem ze sedmisegmentovek.

Ve světě se prosadil asi nejvíc KIM-1, jednodeskový počítač od MOS Technology s procesorem 6502. Vznikla i „odlehčená" verze Junior Computer. Další zajímavý jednodeskáč byl třeba Cosmac Elf nebo Heathkit ET3400.

PMI-80

Začnu PMIčkem. Je totiž velmi dobře zdokumentované, hlavně díky sérii článků v Amatérském Radiu. PMI-80 mělo 1 kB RAM, 1 kB ROM (a mohli jste si druhý kB přidat), klávesnici s 25 tlačítky, devítimístný displej z kalkulačky (obojí připojené přes periferní obvod 8255) a primitivní rozhraní pro magnetofon.

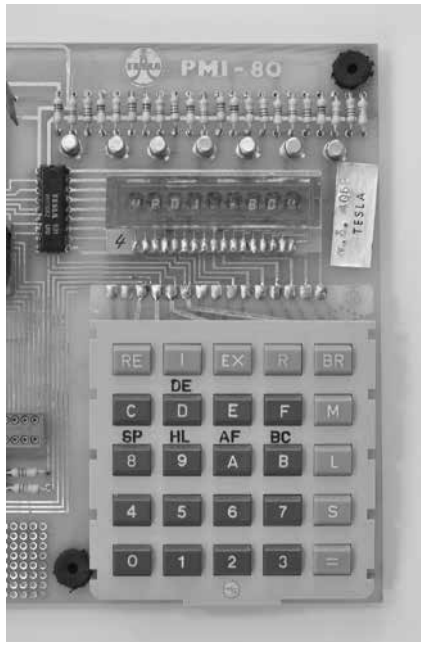
Ovládání PMI-80

Displej byl, jako u většiny jednodeskáčů, rozdělen na část adresní (vlevo) a část datovou (vpravo). Rozdělení nebylo nijak dané, byla to čistě konvence. Klávesnice obsahovala 25 tlačítek, fyzicky rozmístěných do matice 5×5.

Tlačítko RE celý systém zresetovalo, a na displeji se objevil nápis „PMI-80". Po stisknutí libovolné klávesy (tedy kromě RE a I) se na displeji úplně vlevo objeví znak, co má být asi otazník, který říká, že PMI je připravené přijímat pokyny.

Nejzásadnější věc u jednodeskáče byla funkce prohlížení a měnění obsahu paměti. Po stisku tlačítka M (Memory) se vlevo objevilo „velké M" (píšu to v uvozovkách, protože na sedmisegmentovém displeji to vypadá spíš jako obrácené velké U nebo ruské P), pak mezera a čtyři pozice adresy. Třeba: „M 1C00" Pomocí hexadecimální klávesnice jste zadali adresu (pokud jste udělali chybu, prostě jste psali dál, systém bral jako platné pouze čtyři poslední znaky) a tlačítkem „=" jste ji potvrdili. Na displeji se objevil za adresou znak „=" a za ním dva znaky – obsah paměti na dané adrese. Třeba „M 1C00=00". Teď jste mohli zadat nový obsah, zase pomocí hexadecimální klávesnice a zase platily pouze poslední dva znaky. Tlačítkem „=" jste potvrdili změnu a přesunuli se k další adrese.

Konec zadávání jste udělali tak, že jste buď stiskli klávesu, kterou program nečeká (třeba EX), nebo resetovali stroj tlačítkem RE. Tady reset neznamená vymazání paměti, takže to, co jste zadali, v ní zůstává.



Obrázek 10: klávesnice a displej PMI-80 (autor: Teslaton, CC-BY-SA)

Pomocí EX jste program spustili. Po stisku EX zase systém očekává zadání adresy, a po jejím potvrzení tlačítkem „=“ spouští program.

Sofistikovanější podobu měla funkce Break (BR). Zde jste nejprve zadali adresu, na které se má program přerušit a skočit do monitoru, a v druhém kroku spouštěcí adresu. Jakmile program narazil na danou adresu, odskočil zpět do monitoru, a vy jste si mohli zkontrolovat, zda je vše tak, jak má být.

K tomu vám dopomáhala třeba i funkce kontroly registrů – klávesa R. Po jejím stisknutí jste si vybrali, od jaké dvojice chcete prohlížet (AF, BC, DE, HL, SP – viz klávesnice). Zase můžete pomocí klávesnice obsah měnit, tlačítkem „=“ potvrzovat a přesunout se k další dvojici.

Poslední dvě funkce, S a L, fungovaly pro ukládání (save) a čtení (load) programů na pásek a z pásku.

BOB-85

Tuhle konstrukci postavil ing. Kratochvíl s procesorem 8085 a publikoval ji, jak jinak, v Amatérském Radiu, a to včetně dokumentace, výpisu monitoru, několika programů a zdrojových kódů.

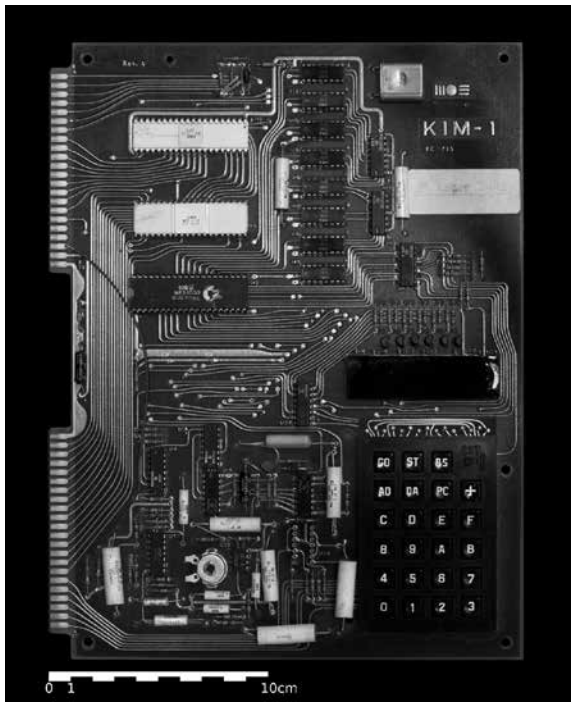
BOB-85 měl plus mínus stejné schopnosti jako PMI-80. Klávesnice měla 24 tlačítek (6×4) a displej pouhých 6 pozic.

Ovládání se od PMI-80 trochu lišilo. Nebyla tu možnost měnit obsah registrů nebo nastavovat breakpointy. Prohlížení paměti fungovalo podobně (jen místo klávesy M se jmenovala S MEM), místo klávesy „=“ byla klávesa „NEXT“ a při opravě dat nefungoval systém tak, že by obsah posouval, ale zadávali jste ho znovu. Navíc proti PMI tu je možnost postupovat pamětí nejen dopředu (NEXT), ale i dozadu (REC).

Klávesa EXEC vyvolala RESET stroje. Klávesa GO spouštěla program od dané adresy. VEK1, VEK2, a RST měly být nějaké přerušovací vektory do budoucna. A klávesa REC kromě už uvedené funkce vyvolávala ještě funkce pro zápis a čtení programů.

KIM-1

Pro nás trochu exotičtější stroj, s těmi jsme se v ČSSR moc nesetkávali. Koncepce opět stejná: procesor (6502), trocha pamětí, šest sedmisegmentovek, klávesnice 6×4, ale hlavně: vyvedený rozšiřující konektor, díky kterému se dalo ke KIM-1 snadno připojit leccos.



Obrázek 11: MOS KIM-1 (autor Rama, CC-BY-SA)

Displej držel stejnou konvenci, 4 pozice adresa, 2 data, ale ovládání bylo trošičku jiné. Tlačítkem AD jste zvolili možnost „zadám adresu“, tlačítkem DA jste začali přistupovat k datům. Roli tlačítka „=“, respektive „NEXT“, tu má tlačítko „+“.

Tlačítko GO spustilo program od adresy, která byla na displeji (zadaná po stisknutí AD). Tlačítko PC vyvolalo hodnotu Program Counteru (PC) na displej. RS je RESET, ST je STOP.

Poslední tlačítko nebylo tlačítko, ale přepínač SST – pokud byl aktivní, fungoval KIM-1 v režimu „Single Step“, tedy po stisknutí tlačítka GO vykonal přesně jednu instrukci a opět se zastavil.

ET3400

Tato počítačová stavebnice obsahovala procesor Motorola MC6800 a kromě klasických „jednodeskáčových“ propriet i nepájivé kontaktní pole. Co je na tomto stroji zajímavé je to, že si vystačí s šestnácti tlačítky a RESETem. Každé tlačítko mělo dvě funkce – kromě hexadecimálního znaku to byl i nějaký povel, a co bylo aktivní, to záleželo na kontextu.

Mimochodem, vřele doporučuju najít a přečíst si manuál k tomuto počítači, stejně jako manuál ke KIM-1! Není to jen „jak se to ovládá“, ale představují i procesor, popisují jeho funkce, učí ho programovat, a příručka k ET3400 začíná dokonce velmi podrobným konstrukčním návodem (IKEA faktor: 100!)

2 Slavné osmibitové mikroprocesory

2 Slavné osmibitové mikroprocesory

Takto se to učí ve školách: *Základní prvky každého číslicového počítače jsou: řadič, aritmeticko-logická jednotka (ALU), paměť, periferie. Řadič spolu s ALU tvoří procesor. Pokud je integrovaný do podoby jednoho nebo několika málo obvodů, hovoříme o mikroprocesoru.*

Traduje se, že prvním čipem takového druhu byl Intel 4004, a od něj je počítána „éra mikroprocesorů".

Obrázek 12: Intel 4004 (autor Thomas Nguyen, CC-BY-SA)

Mikroprocesor 4004 pracoval s daty o šířce 4 bity, což nevadilo, protože jeho zamýšlené použití bylo v kapesních kalkulátorech. Jeho drobnou úpravou vznikl procesor 4040. Ale kromě této vylepšené verze vznikl i mikroprocesor 8008 – jak už číslice napovídají, zvětšila se šířka sběrnice a zpracovávaných dat, a to na 8 bitů. Návrháři připravovali 8008 jako implementaci procesoru pro terminál Datapoint 2200 – původně měla 8008 nahradit několik desek s TTL obvody v těchto teminálech od společnosti CDC, ale vývoj vázl a původní kontrakt byl nakonec zrušen. Výsledná dohoda zněla tak, že CDC si nechá peníze a Intel svůj procesor.

2.1 8080 a 8085

Procesor 8008 se na trhu ani pořádně neohřál, a už přišel vylepšený typ 8080, záhy v rychlejší verzi 8080A. Tento procesor vlastně definoval celou rodinu procesorů 80xx od Intelu. Při přechodu od osmibitů k šestnáctibitům totiž Intel vzal právě tento procesor a jakoby jej „natáhl" na 16 bitů. Návrháři měli stále před očima nebohé vývojáře, kteří budou muset přepisovat osmibitové programy na šestnáct bitů, a tak se jim snažili přechod ulehčit. Vznikly tak známé „segmentové registry" a další radosti, které zná každý, kdo programoval v assembleru pro procesory řady x86. Tato architektura pak plynule přešla i do vyšších verzí, od procesoru 80386 zůstala uvnitř jako „virtuální 86", a svým způsobem je s námi vlastně dodneška…

Procesoru 8080, respektive vylepšené variantě 8085, se budu v této knize ještě věnovat intenzivně – naučíme se jej zapojovat, postavíme si vlastní jednodeskový počítač a projdeme si krátkým kurzem programování v assembleru. Ale to až v dalších kapitolách.

Procesor 8080 má v bývalém Československu velmi silné postavení – československá Tesla totiž vyráběla klon tohoto procesoru pod označením MHB8080A, a tak většina československých mikropočítačů používala právě tento procesor.

Přitom nebyla 8080 nějak moc přívětivá. Návrhář musel počítat s tím, že procesor potřebuje tři napájecí napětí, totiž +5 voltů, +12 voltů a -5 voltů, které navíc musely nabíhat v určeném pořadí (respektive alespoň zařídit, aby při výpadku napětí -5 V vypnulo napětí +12 V). Kromě toho potřeboval procesor ještě dva podpůrné obvody, 8224 a 8228, z nichž první se staral o hodiny a o signál RESET a druhý pomáhal procesoru generovat řídicí signály a obsluhovat přerušení.

I u Intelu si těchto omezení byli vědomi a po čase přišli s vylepšenou verzí 8085. Tento procesor si vystačil s jediným napájecím napětím +5 voltů, hodinové signály mu stačily obyčejné a obešel se bez podpůrných obvodů. Navíc měl vylepšený přerušovací systém a měl navíc i jednobitové sériové rozhraní, ovládané instrukcemi RIM a SIM (tyto dvě instrukce měl 8085 navíc; používaly operační kódy, které v původní 8080 nebyly použité).

Po 8085 už další osmibit od Intelu nepřišel (když nebudeme uvažovat jednočipové počítače 8048 / 8051 / 8052), místo toho následovala 16bitová rodina x86. Přesto procesor 8080 vyráběla spousta výrobců v nejrůznějších variantách.

2.2 Z80

Návrhář procesorů ve společnosti Intel, který se jmenoval Federico Faggin, po úspěchu procesoru 8080 odešel a založil si vlastní firmu Zilog. Ta na trh v roce 1976 uvedla vlastní procesor s označením Z80. Z80 byla vlastně přepracovaná a rozšířená 8080. Při rozšiřování nechali návrháři vše to, co měla 8080 – tedy instrukční sadu a sadu registrů – a vlastní vylepšení „nacpali“ do volného místa v instrukčním souboru. Čtyři kódy měly funkci „prefixu“ a v kombinaci s dalšími vybíraly instrukce z rozšířené instrukční sady. V ní byly dotaženy některé instrukce, které 8080 neměla (třeba bitové posuny), některé instrukce byly rozšířené tak, že mohly pracovat i s jinými registry, přibyly instrukce pro blokové operace (přesun, porovnání, přenos dat), a především přibyly dva indexové registry pro snazší práci s paměťovými strukturami a k nim odpovídající prefixy pro jejich použití. Pro programátorovo pohodlí přibyla druhá sada pracovních registrů, mezi nimiž se dá přepínat, byl rozšířen a vylepšen přerušovací systém a byla zabudovaná podpora refreshe pro dynamické paměti (co je refresh vysvětlím později v kapitole o pamětech).

Z80 si vystačila s jedním napájecím napětím, brzy přišly verze, schopné pracovat až s frekvencí 4 MHz (8080A pouze 2 MHz) a většina instrukcí se zpracovávala rychleji (potřebovala menší počet cyklů). V ČSSR se tento procesor bohužel nevyráběl; ze socialistických zemí ho vyráběla například NDR pod označením U880D a používali jej němečtí výrobci, například známý Robotron.

2.3 6800

Intel nebyl jediný, kdo vyráběl mikroprocesory. Nezávisle na něm svoje mikroprocesory představili i další výrobci – Texas Instruments, Signetics, National Semiconductor, RCA nebo třeba Motorola. Ačkoli i procesory jiných výrobců byly zajímavé, úspěšné byly především dvě větve. První byla ta od Intelu, 80xx, do níž můžeme započítat i Z80 od Zilogu. Druhá úspěšná větev pak byla ta s označením 68xx, právě od Motoroly.

Motorola byla v té době druhý největší výrobce integrovaných obvodů a polovodičů v USA (první byl Texas Instruments). Několik měsíců po procesoru 8080 představila i ona svůj mikroprocesor s označením 6800. Jeho návrh byl ovlivněn známým a úspěšným minipočítačem DEC PDP-11. Procesor 6800 pracoval na nižší frekvenci než 8080A, ale výkon měly oba procesory srovnatelné (6800 potřeboval menší počet cyklů na vykonání jedné instrukce). 6800 měl bohatší možnosti adresování, což bylo právě dědictví zmíněného „ideového předchůdce", počítače PDP-11. Na druhou stranu používal jeden adresní prostor pro paměť i periferie.

Po čase přišly další varianty, jako 6805, 6809 a 6811 – některé z nich jsou stále základem jednočipových mikrokontrolérů.

Podobně jako Intel se i Motorola připravovala na nástup šestnáctibitové techniky, ale na rozdíl od Intelu šli její návrháři na věc z druhého konce. Místo aby vzali návrh 680x a zvětšili jej na 16 bitů, navrhli plně 32bitový procesor 68000 (označuje se i jako 68k) a dali mu 16bitovou datovou sběrnici. Vyhnuli se tak potížím s reálnými módy, překládáním adres a podobnými radostmi, co zažívali vývojáři ve světě x86, a zároveň měli základ pro plně dvaatřicetibitové procesory, které přišly po čase. Řada 68000 pokračovala modely 68010, 68020, 68030, 68040 – podobně jako u Intelu 286, 386, 486… Posléze byl vývoj této řady ukončen a nastoupila po ní varianta PowerPC (Apple – IBM – Motorola). Řada 68000 prošla drobnou úpravou a stala se základem mikrořadičů ColdFire a DragonBall.

Nakonec Motorola s procesory ztratila dech, její hlavní klient, Apple, přešel na technologii od Intelu, a tak Motorola z výroby polovodičů udělala nejprve samostatnou firmu Freescale (2004), kterou posléze koupila firma NXP, tu pak firma Qualcomm…

2.4 6502

Největší nevýhoda procesorů 8080 a 6800 byla jejich cena. Tedy – pro velké firmy ne, těm to nevadilo, ale vysoká cena bránila rozšíření mezi počítačové nadšence. Chuck Peddle, jeden z návrhářů procesoru 6800, navrhoval svému vedení přejít na levnější výrobní technologii a prodávat procesory v kusovém množství koncovým zákazníkům, ale management o něčem takovém nechtěl slyšet. Ostatně, kdo nikdy neslyšel okřídlenou větu *to je vážná věc, to není na hraní?* Inu, procesory byly vážná věc, průmyslová technologie, a nějací *bastliči* výrobce nezajímali.

A tak Chuck Peddle s pár kolegy od Motoroly odešel a ve firmě MOS Technology navrhnul vlastní procesor MOS 6501/6502. Na jiném místě popisuju, jak to s ním tehdy bylo, tak zde jen ve stručnosti: Procesor 6501 byl vývodově kompatibilní s procesorem 6800, ale vnitřek měl výrazně jiný, zjednodušený. Zmizel jeden akumulátor, indexové registry byly jen osmibitové a tak dále, ovšem namísto 150 dolarů stál pouhých 25. Což se Motorole samozřejmě nelíbilo, tak MOS Technology zažalovala. Peddle obratem stáhnul 6501 z nabídky – ale nevadilo mu to, protože spoléhal hlavně na procesor 6502. Ten měl jiné rozmístění vývodů, takže případná náhrada nebyla tak úplně jednoduchá, a měl jinak zapojený generátor hodin. Především měl ale díky soudnímu sporu postaráno o reklamu.

Procesor 6502 se stal po celém tehdejším západním světě legendou. Používaly ho nejrůznější počítače, domácí i průmyslové, od Apple I po herní konzole… A to i přesto, že měl některá divná omezení a některé divné chyby v instrukcích.

V socialistickém bloku moc populární mezi konstruktéry nebyl, vyrábělo ho snad jen Bulharsko pro svoje počítače Pravec 8D (klon Oric Atmos). Programátoři se s ním ale setkávali v domácích počítačích Atari nebo Commodore.

2.5 6809

Mnohými je tento procesor považován za nejlepší procesor osmibitové éry vůbec. Například Bill Gates tvrdil, že pro lepší osmibitový mikroprocesor nikdy programy nepsal. Vznikal v Motorole ve stejné době, kdy se pracovalo na procesoru 68000, a oba návrhy se částečně ovlivnily. Procesor 6809 rozšiřuje a doplňuje návrh 6800 a dotahuje to, co v 6800 nebylo dotaženo, ať už z důvodu nezkušenosti návrhářů, nebo proto, že to tehdejší technologie neumožňovala.

Na začátku návrhu stála důkladná analýza reálných programů pro 6800. Na základě statické a dynamické analýzy zjistili návrháři, které operace jsou nejčastější a jaké programátorské konstrukce vývojáři nejčastěji používají. Pak z těchto informací vyšli při návrhu architektury a instrukční sady procesoru 6809. A na výsledku to bylo doopravdy vidět.

2.6 Architektury procesorů

Už jsem zmiňoval, že mezi mikroprocesory jsou poměrně zásadní rozdíly – čímž samozřejmě nemyslím to, že je každý jiný, ale mám na mysli rozdíly v přístupu, takříkajíc filosofické.

Za chvíli si ukážeme procesor Intel 8080. Jeho nejznámější rival byl vyvinut firmou Motorola a byl označen 6800. Některé rysy měl procesor 6800 shodné s procesorem 8080 (třeba 16bitovou adresovou sběrnici, osmibitovou datovou sběrnici), ale ve spoustě věcí se lišil.

6800 používá zápis vícebajtových čísel stylem Big Endian (8080 stylem Little Endian). Pokud máte na adresy 0000 a 0001 zapsat hexadecimální číslo 1234h, pro procesor 8080 ho zapíšete tak, že na nižší adresu zapíšete nižší bajt (34h), na vyšší adresu vyšší (12h). U Motoroly 6800 to bude přesně naopak. Ve výpisu paměti na jednom řádku to pak bude vypadat takto:

```
Intel:    0000: 34 12 xx xx
Motorola: 0000: 12 34 xx xx
```

Procesor od Motoroly nerozlišuje paměť a porty a pracuje s obojím stejně. Nemá tedy pro čtení a zápis z/do portů speciální instrukce, ale používá standardní instrukce pro práci s pamětí, které jsou mnohem flexibilnější. Nevýhodou je, že se tak návrháři připravují o souvislý prostor 64 kB pro paměť, protože část musí vyhradit pro porty.

Procesor 8080 má šest osmibitových registrů pro volné použití a jeden akumulátor, s nímž se provádí většina výpočtů. Motorola použila v procesoru 6800 pouhé dva osmibitové akumulátory (A a B) a jeden šestnáctibitový indexový registr X. (Ukazatel zásobníku SP a programový čítač PC jsou i zde, oba šestnáctibitové).

Na první pohled to může vypadat jako nedostatečné – vždyť některé algoritmy využijí klidně šest, sedm registrů. Jak to tedy řešit u 6800?

Procesor 6800 přišel s jednoduchým trikem, který po něm převzali i jeho následníci. Paměť si rozdělil na 256 stránek po 256 bajtech (tj. horní byte adresy je „číslo stránky“ a dolní „adresa ve stránce“). Stránka 0 pak má speciální roli, protože je snadno adresovatelná. Co to znamená?

Procesor 8080 měl u spousty instrukcí pevně dáno, s jakými daty pracuje. Buď s registrem, s přímo zadanou konstantou, nebo s obsahem paměti, adresovaným registry H a L, nebo 16bitovou adresou. 6800 naproti tomu měl instrukční sadu mnohem víc *ortogonální*.

Ortogonální instrukční sada je taková, kdy instrukce může pracovat s různými daty naprosto stejně, bez ohledu na to, jestli jsou v registru, nebo v paměti.

Procesor 6800 proto u instrukcí nabízel několik adresních módů. Operand mohl být určen:

- Implicitně. Například instrukce INCA pracuje s registrem A.
- Bezprostředně (Immediate). LDAA #$02 nahraje do registru A (LoaD Accumulator A) konstantu „bezprostředně zapsanou“ za operačním kódem – tedy 02h.
- Přímou adresou (Direct). LDAA $02 nahraje do registru A hodnotu, která je uložená na adrese, co je zapsaná za operačním kódem. Adresa je pouze osmibitová, vyšších osm bitů se doplní nulami, adresa je tedy 0002h. Instrukce zabírá dva bajty, první je operační kód, druhý je zkrácená adresa – tedy vlastně „adresa ve stránce 0“.

- Rozšířenou adresou (Extended). Obdoba předchozího zápisu, ale adresa je dvojbajtová, lze tedy adresovat paměť v celém rozsahu. LDAA $1234 nahraje do registru A hodnotu z adresy 1234h.
- Indexovanou adresou, tedy adresou, která vznikne součtem hodnoty z indexového registru X a osmibitové konstanty.
- Relativně. V takovém případě se k adrese PC přičetla hodnota konstanty v rozsahu -128 až +127.

Pokud programátor chce k obsahu akumulátoru (tedy registru A nebo B) přičíst nějaké číslo, použije instrukci sčítání a podle adresního módu určí, jestli se bude zadávat přímo nebo z paměti, a pokud z paměti, tak jestli plnou adresou (výhoda: může být kdekoli, nevýhoda: instrukce je o bajt delší a časově náročnější), nebo ze stránky 0, nebo pomocí indexového registru… Hodnota může být přitom klidně načtena z portu, protože, jak jsme si už řekli, procesor 6800 nerozlišuje mezi porty a pamětí a ke všemu přistupuje stejně.

Z popisu adresních módů vyplývá, že stránka 0 je snadno dostupná a je to takový kompromis mezi plnou adresou a registrem v procesoru. Přístup k těmto 256 buňkám paměti je rychlý a instrukce jsou i kratší, lze je tedy považovat za určitý ekvivalent vnitřních registrů procesoru.

Assemblerem procesoru 6800 se v této knize zabývat nebudu, přesto jsem ho tu uvedl. Z architektury 6800 totiž vyšel nejen legendární procesor 6502 (kterému se věnovat budu), ale i další procesory – třeba „poslední osmibitový procesor“ 6809 (samozřejmě v uvozovkách – tento procesor měl na tehdejší dobu ojedinělé možnosti, ale přišel bohužel pozdě) a celá řada procesorů počínaje procesorem MC68000 přes další procesory téže řady až k MC68060 a dál k PowerPC. Tato řada se v běžných domácích a kancelářských počítačích už nevyskytuje (od přechodu Apple na Intel), ale dodneška se udržuje v embedded zařízeních apod.

Dovolte mi nostalgickou odbočku. Po 8080 jsem přešel na Z80 (byl jsem Spectrista) a tenhle procesor mi učaroval. Stovky instrukcí, mnoho registrů, šestnáctibitová aritmetika… Na 6502 jsem se díval s totálním nepochopením: vždyť to má jen tři registry a je to pomalé (=má to malou frekvenci). Ve skutečnosti byla „datová propustnost“ srovnatelná. Ostatně operace s pamětí v zero page zabraly 3 takty procesoru, nejrychlejší operace s pamětí u Z80 trvala 7 taktů. Krásu téhle architektury jsem docenil až později, když jsem se naučil assembler 68000 (Atari ST, Amiga). V čerstvé paměti jsem měl tehdy naučený asembler 8086, se všemi jeho „segmenty“ a „instrukcemi, co používaly vždy přesně určené registry“ – a najednou máte k dispozici ortogonální sadu a osm adresních + osm datových registrů. Třeba instrukce PUSH (která tam vlastně nebyla) fungovala jako obyčejný přesun hodnoty z registru do paměti s adresním módem „přenes do paměti na adresu danou některým adresním registrem, ale předtím hodnotu sniž“ (predekrement).

3 OMEN: Stavíme vlastní počítač

3 OMEN: Stavíme vlastní počítač

Samozřejmě osmibitový a s tak historickým duchem, jak to jen jde.

Když jsem přemýšlel o tom, jak tuto knihu pojmout, bylo mi jasné, že suchý výklad nebude stačit. Tedy, on by stačil, ale já nejsem zrovna příznivcem teoretických učebnic bez přesahu do praxe. Bylo mi tedy jasné, že nějakou konstrukci chci.

Ano, můžu vzít nějaký reálný osmibit a ukazovat na něm, jak se věci mají, možná vás i ponouknout ke stavbě repliky, ale touto cestou se mi moc jít nechtělo. Říkám si, že na vlastní konstrukci, která bude tak jednoduchá, jak jen to lze, půjde ilustrovat požadované věci mnohem líp.

Zvolil jsem tedy cestu konstrukce naprosto jednoduchých počítačů, které mají splňovat několik bodů:

- Budou používat součástky, které se dají běžně koupit.
- Nebudou se ortodoxně držet konstrukcí z 80. let – tedy když chci použít paměť, použiju moderní 32kx8bit statickou RAM, což je jeden čip, a nebudu konstruovat zapojení z osmi čipů dynamické RAM a dalších tří integrovaných obvodů kolem dokola.
- Budou v duchu starých osmibitových časů, tedy jednoduché periferie – klávesnice, displej, reproduktor – a minimální softwarové vybavení.
- Bude jednoduché psát pro ně software.

Klidně vynechám samotný starý procesor a potřebné funkce si naemuluju v moderním jednočipu. Nebo použiju jednočip místo periferií.

S těmito body na paměti jsem sednul a navrhl, upravil či vybral několik konstrukcí, které jsou vhodné pro začátečníky. Když jsem sáhl po cizích zapojeních, tak jsem se snažil vybrat konstrukce rozmanité, u nichž licence umožňuje jejich úpravu.

A tak vznikla „značka OMEN“.

3.1 OMEN?

Nojo. *Osmibitový Mikropočítač pro Elektronické Nadšence*. OMEN. A značení pomocí hláskového kódu: Alpha, Bravo, Charlie, Delta…

V knize bohužel není dostatek místa na popis všech instrukcí a procesorů, navíc se nakladatel se bál, že obsáhlejší kniha bude hůř prodejná. Některé konstrukce a k nim připravený text proto vyjde pouze online, formou dodatků.

OMEN Alpha je jednoduchý počítač ve stylu jednodeskových strojů z konce 70. let s procesorem 8085. Obsahuje jen to nezbytné minimum, ovládáte jej pomocí terminálu, nebo pomocí připojené hexadecimální klávesnice a sedmisegmentového displeje. Přesto lze připojit i složitější zařízení, včetně třeba čtečky SD karet či rozhraní CF/IDE.

OMEN Bravo je verze Alpha, která používá procesor 6502. Periferie jsou víceméně shodné s Alphou, ale software je, logicky, jiný.

OMEN Charlie je jednoduchá handheld herní konzole s černobílým displejem. Pohání ji ATMega328 – tedy přesněji Arduino (ve verzi Arduino Micro). Používá displej ze staré Nokie, dvě tlačítka, joystick a „pípák". Výsledkem je funkční herní konzole se schopnostmi někde na úrovni kapesních her z 80. let. Ale Charlieho můžete naprogramovat v BASICu a běží na něm programy pro procesor 8080. Drobnou úpravou získáte Omen Charlie Color – s větším rozlišením a barevným displejem!

OMEN Delta je počítač s reálným osmibitovým procesorem Zilog Z80. Bohužel bez výstupu na televizi. Místo toho používá sériový port. Stačí pouhé 4 integrované obvody, klidně si to poskládáte na nepájivém kontaktním poli, a na výsledném stroji může běžet CP/M. Ano, CP/M. Klidně si spustíte i Turbo Pascal…

OMEN Echo je počítač s černobílým grafickým displejem. Jako hlavní procesor používá opět jednočip AVR – může to být buď Arduino, nebo „velká" ATMega1284. Kromě procesoru obsahuje už jen jeden čip, sériovou paměť. K tomu pár rezistorů a kondenzátor. K Echu připojíte televizi s video vstupem (nebo LCD monitor se vstupem PAL) a klávesnici PS/2. Pro zájemce o „ještě většího osmibitového ducha" mám i úpravu Echo MKII – s tlačítkovou klávesnicí a la ZX-81. Echo sice používá ATMegu, ale uvnitř může běžet například emulovaný procesor 6502.

Nadšenci můžou Echo použít i v neemulovaném módu a napevno naprogramovat třeba nějaký dialekt BASICu. Výkonem to pak bude podobné třeba zmíněnému ZX-81, Jupiteru ACE či československému počítači Ondra.

OMEN Foxtrot je počítač, který je z velké většiny „syntetizovaný". Ve skutečnosti jde o obří programovatelné logické pole (FPGA), v němž je zkonstruovaný celý počítač: procesor, periferie, adresní logika, … Výstup můžete pustit na VGA, klávesnici použijete PS/2, a uvnitř vám běží osmibitový procesor s výkonem, jakých reálně nedosahují. Použil jsem takzvaný „soft core", tedy „softwarově definované procesorové jádro" s procesorem Motorola 6809. Také tento počítač se objeví až v dodatcích, přímo v této knize není popsaný, především proto, že k jeho pochopení je potřeba vysvětlit mnoho teoretických věcí k programovatelným polím a jazyku VHDL.

OMEN Kilo je reálný počítač s reálným procesorem 6809, svými schopnostmi odpovídající počítačům Alpha a Bravo.

Samozřejmě nejste odkázáni jen na tyto konstrukce – v každé sekci vám poskytnu několik tipů na zajímavé konstrukce, co se nacházejí na internetu.

Nechci vás připravit o tu radost postavit si vlastní počítač a vytvořit si pro něj software takříkajíc „na zelené louce". Kdybyste se ale chtěli podívat, jaké programy jsem si napsal já, můžete zde:

https://github.com/osmibity

3.2 Zapojování v praxi

V této knize si navrhneme počítač takříkajíc „ideově", v podobě schématu, ovšem tuto knihu nečtete přeci jen proto, abyste se kochali schématy, ale proto, abyste si něco vyzkoušeli. Jak si tedy vyzkoušíte stavbu vlastního počítače?

Možností je několik. Začněme tou nejjednodušší: postavit si konstrukci na nepájivém kontaktním poli.

Nepájivé kontaktní pole (breadboard)

To jde? Ano, to opravdu jde. Nepájivá kontaktní pole sice mají svá omezení, ale většinu konstrukcí z této knihy postavíte i na nich.

Nevýhody NKP jsou dané hlavně jejich mechanikou: součástky po čase mohou ztrácet kontakt, stejně tak propojovací kabely. Konstrukce není moc pevná ani vzhledná, navíc někdy neopatrný dotek může způsobit výpadek či podivné chyby. Propojování je pracné, zdlouhavé a náchylné k chybám.

Dobrý tip: Napište si na papír seznam propojek. Můžete si pak odškrtávat, co máte hotové, a zkontrolovat, co vám kde chybí.

Počítejte ale s tím, že oživování nemusí být úplně snadné. Pečlivou prací a kontrolou ovšem riziko chyb snížíte na minimum.

Popisky na obvody

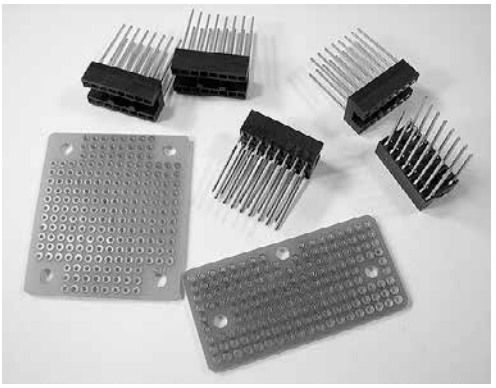
To je trik, co používají někteří konstruktéři: vytisknou si na papír rozmístění vývodů integrovaného obvodu i s popisem signálů ve skutečné velikosti a přilepí si ho na integrovaný obvod. Při zapojování tak nemusíte hledat, kolikátý vývod je jaký, ale přímo vidíte, kde je D0 a kde je /CS…

Ovíjené spoje

Jde o poměrně starou techniku zapojování elektronických obvodů. Součástky umístíte do nějakého univerzálního plošného spoje, nebo jen do tenké nevodivé desky s předvrtanými otvory

(anglicky: perforated board, perfboard), a pomocí izolovaného vodiče propojujete vývody jeden po druhém. Tato technika se v 70. letech používala i pro průmyslovou výrobu velkých počítačů.

Ovíjené spoje jsou o něco robustnější než konstrukce na NKP, jsou pevnější, odolnější, ale docela snadno se rozebírají. Pro ovíjené spoje jsou určené speciální objímky na integrované obvody s dlouhými vývody (wire wrap sockets).



Nevýhodou je, že tyto objímky jsou výrazně dražší než „obyčejné".

Pro ovíjení spojů budete potřebovat izolovaný vodič a speciální nástroj, který slouží na samotné ovíjení (většinou má i část, která slouží k odizolování vodičů). S trochou cviku jde ovíjení docela dobře od ruky a na rozdíl od pájení, u něhož hrozí popálení a vzniká spousta agresivních výparů, je tato technika nenáročná a můžete ji klidně provozovat v obývacím pokoji.

3.3 Pájení na univerzální desce

Pájení je nejpoužívanější způsob propojování součástek v elektronice. Je dostupné i pro domácí použití a s dobrou páječkou můžete zapájet i moderní obvody se stovkami vývodů. Ty ale naštěstí

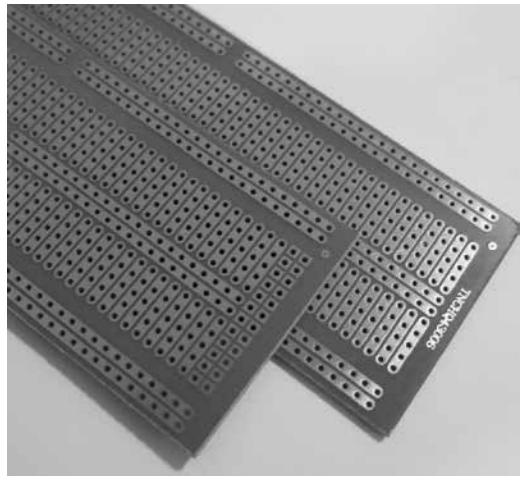
používat nebudeme, zůstaneme u klasických obvodů s vývody s roztečí 2,54 mm. Na pájení takových obvodů stačí dobrá mikropáječka či pájecí pero. Doporučuju si trochu připlatit a koupit něco s regulací teploty.

Nedoporučuju transformátorové („pistolové") páječky. Jednak se kolem jejich hrotu vytváří elektrické pole při zapínání a vypínání, a na to je třeba myslet při pájení integrovaných obvodů, ale hlavně: jsou těžké, a než zapájíte osmdesát vývodů, bude vás bolet ruka.

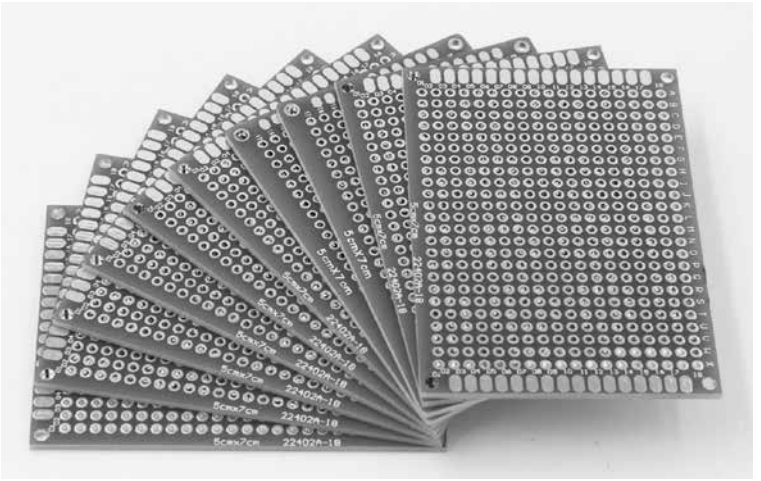
Pokud se rozhodnete konstrukci pájet, vždy doporučím pro integrované obvody použít dobrou patici. Usnadní to případné budoucí opravy. Ne že by nešlo odpájet integrovaný obvod se čtyřiceti vývody, ale vyžaduje to cvik a odsávačku nebo odsávací knot, a i tak hrozí nebezpečí tepelného poškození plošného spoje.

První možnost, jak spájet obvod, je použít takzvanou *univerzální desku plošných spojů*. V angličtině Universal PCB, Veroboard, Prototype board. Jsou to obdélníkové desky s otvory vyvrtanými v rastru 2,54 mm a s pájecími ploškami na druhé straně. Levnější desky plošných spojů mají „co otvor, to plošku", dražší mají třeba trojice otvorů vždy spojené nebo mají rozvedené napájecí napětí po celé desce.

Postup je podobný ovíjeným spojům, jen s tím rozdílem, že zde pájíte, neovíjíte. Připájíte objímky k desce, a pak pomocí krátkých izolovaných vodičů propojujete vše, co má být propojené. Výsledkem je opět více či méně vzhledné „vrabčí hnízdo", s rizikem zapomenutých spojů, kontrola je náročná, změny ještě víc…



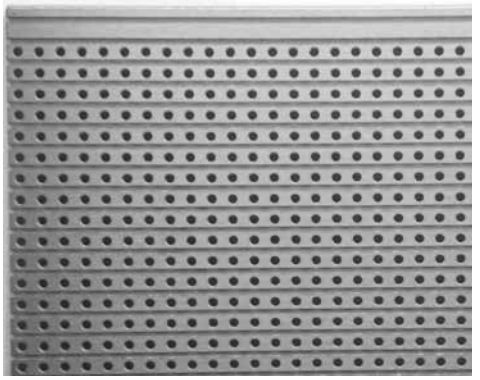
Obrázek 13: Univerzální deska plošných spojů, též „Veroboard"



Obrázek 14: Levnější varianta univerzální desky, provedení „perfboard"

Pro zajímavost: jakmile připojíte k osmibitovému procesoru paměť RAM a ROM, znamená to, že musíte spojit osm datových vodičů a (třeba) 15 adresových (16. bude sloužit k výběru). Musíte tedy připájet dva vodiče – první od procesoru k jedné paměti, druhý od první paměti k druhé paměti – pro každý z těchto signálů. 46 vodičů, tedy 92 pájecích bodů. A to je zatím jen sběrnice a tři obvody…

Jednou ze zajímavých variant univerzálních plošných spojů je takzvaný Strip Board – plošný spoj s propojenými řadami vývodů:



Obrázek 15: Univerzální deska „stripboard"

Pomocí jemné brusky či frézy (na přerušení vedení) a propojovacích vodičů můžete pospojovat i složitější obvody, navíc existují i programy, které dokáží navrhnout a optimalizovat rozložení součástek tak, aby bylo zapotřebí co nejméně propojek a broušení. Hledejte „Veroboard / Stripboard Software".

Plošný spoj na míru

Nejlepší způsob konstruování trvalých zapojení je vytvoření plošného spoje na míru. Minimalizují se tím nutnosti propojování drátovými spojkami a podobné. Takové spoje jsou zároveň velmi spolehlivé, trvanlivé atd. – vždyť jde o průmyslový standard pro stavbu elektroniky.

Mnozí hobby elektronici se návrhu vlastních plošných spojů bojí. Částečně neprávem, protože to zas tak složitá věc není. Pokud nebudete navrhovat vysokofrekvenční obvody a mnohovrstvé plošné spoje, zůstanete u jednovrstvých či dvouvrstvých a u jednoduchých konstrukcí, nemáte se čeho bát, stačí si pamatovat několik základních pouček, jako třeba že krystal má být co nejblíž procesoru, signál veden pokud možno přímo a kolem oscilátoru je dobré udělat velkou plochu se zemním potenciálem.

Jednodušší plošné spoje si můžete vyrobit i doma – stačí obrazec nějak přenést na vrstvu mědi a vyleptat. Já používal fotocestu: obrazec jsem vytiskl na průhlednou fólii na laserové tiskárně, „počmáral" jsem ho černým fixem (zlehka; důvod je ten, že laserová tiskárna i na nejvyšší kontrast měla nerovnoměrné krytí tonerem a černý fix právě takové světlejší ostrůvky překryl) a přitiskl na cuprextit s fotocitlivou vrstvou. Osvítil jsem ho několik minut ultrafialovou rtuťovou výbojkou (horským sluncem), osvícenou desku jsem vyvolal v roztoku hydroxidu sodného (ten odplavil neosvícený fotorezist) a pak jsem leptal v chloridu železitém. Vyleptanou desku jsem vrtal a ošetřoval stříbřením a kalafunou. Pro menší konstrukce je to dostatečné.

Složitější, větší nebo dvouvrstvé desky doporučuju nechat vyrobit u nějaké služby, zaměřené na malosériovou výrobu plošných spojů. České služby jsou fajn a odzkoušené, jen poměrně drahé a uživatelsky nepřívětivé. Doporučím spíš zahraniční služby, kam nahrajete připravené podklady (ve formátu GERBER – vygeneruje každý slušný CAD pro elektroniku) a přímo na webu uvidíte cenu za výrobu (nejčastěji za 10 kusů) a poštovné. Za desku o velikosti shieldu pro Arduino dáte třeba po přepočtu deset korun, jen holt budete čekat, než vám ji pošta doručí.

Takto vytvořený plošný spoj má většinou i potisk a nepájivou masku, takže se velmi snadno osazuje. Na druhou stranu, pokud uděláte v návrhu chybu, napravuje se mnohem hůř – máte třeba deset desek s chybou, kterou musíte řešit nějakou složitější úpravou, broušením, pájením propojek apod. Doporučuju proto nejdřív sestavit prototyp, a až pro ověřené zapojení navrhovat desku plošných spojů.

Já čas od času připravím sadu součástek pro stavbu výše zmíněných počítačů, a to včetně plošných spojů. Sledujte webové stránky této knihy a dozvíte se, kdy budou sady k dispozici.

4 Mikropočítač a jeho součásti

4 Mikropočítač a jeho součásti

4.1 Jak pracuje mikroprocesor?

Zásadní otázka, že? Z těch procesorů, které jsem představil, všechny fungují v hlavních rysech stejně.

Obvod procesoru má vyvedenou adresní sběrnici A0-A15 (tedy s šestnácti vývody). Adresová sběrnice je výstupní; adresu totiž určuje procesor a celý běh systému řídí (až na jednu výjimku, ale o té později).

Procesor má k dispozici také osmibitovou sběrnici D0-D7. Tato sběrnice je obousměrná, data tedy mohou téct jak do procesoru, tak z procesoru. Pomocí této sběrnice procesor přijímá data, která potřebuje, a zapisuje data, která chce uložit.

V této knize budu zapisovat čísla buď v desítkové soustavě., tedy tak, jak jsme zvyklí, nebo v soustavě šestnáctkové. Nebudu zde vysvětlovat, jak šestnáctková soustava funguje, jen upozorním, že pokud není řečeno jinak, tak budu šestnáctková čísla zapisovat s postfixem „h“ – třeba 0A55h a podobně.

Kromě datové a adresní sběrnice nabízí procesor i řídicí signály. Nejčastěji jde o signál „čtení z paměti“ a „zápis do paměti“. Když procesor potřebuje číst, nastaví datovou sběrnici na vstup, vystaví na adresovou sběrnici adresu paměti, ze které chce číst, a aktivuje signál „Čti paměť“ (Memory Read, /MEMR). Po uplynutí doby, dané konstrukcí procesoru a jeho časovacími možnostmi, očekává procesor, že na datové sběrnici jsou data z paměti na dané adrese.

V případě zápisu do paměti opět procesor nastaví adresní sběrnici na potřebnou hodnotu. Na datovou sběrnici, přepnutou do režimu „výstup“, nastaví požadovanou hodnotu pro zápis a aktivuje signál „Zápis do paměti“ (Memory Write, /MEMW).

Procesory odvozené od Motoroly 6800 si vystačí pouze s adresováním paměti. V systémech s těmito procesory musí návrhář počítat s tím, že část rozsahu musí vyhradit pro vstupně – výstupní zařízení. Procesory odvozené od Intelu 8080 mají alternativní dvojice /IOR, /IOW (Input-Output Read / Write). Procesor tak má pro paměť celý prostor a periferie mají vlastní adresy (u 8080/8085 to je 8 bitů, u Z80 16 bitů).

Procesor Z80 nepoužívá /MEMR, /MEMW, /IOR a /IOW. Namísto toho má dvě dvojice signálů: první určuje, jestli se pracuje s pamětí (/MREQ) nebo s periferií (/IORQ). Druhá dvojice pak říká, zda se čte (/RD), nebo zapisuje (/WR). Výsledné signály si pak musíte poskládat pomocí hradel.

CLOCK

Každý procesor má vstup, určený pro systémové hodiny. Některé procesory mají hodiny jednofázové, některé dvoufázové (8080). Hodinový signál udává takzvaný takt procesoru a označuje se jako T. Takt je nejmenší jednotka času pro procesor a synchronizují se jím veškeré operace.

U starých osmibitových mikroprocesorů trvaly instrukce několik taktů (T) – vždy alespoň jeden. Postupně se na sběrnici vystavila adresa pro přečtení instrukce, pak se přečetla hodnota, ta se dekódovala a podle typu instrukce se vykonávaly další akce. U některých procesorů se díky technice pipeline dělo všechno toto najednou – zatímco se jedna instrukce provádí, druhá už se dekóduje a na pozadí se třetí čte. Postupně se výkon zvyšoval a moderní procesory dokáží provádět nejen několik instrukcí najednou, ale zvládají i spekulativní provádění dvou větví programu – procesor například vykonává podmíněnou instrukci skoku, pokud je výsledek operace 0, ale v pipeline už jsou připravené obě varianty: pro nulu i pro nenulu. Podle výsledku se pak jedna větev „potvrdí" a druhá zahodí. Ale u starých osmibitů platí varianta známého úsloví *co na srdci, to na jazyku*: co na sběrnici, to v procesoru!

DMA

Většina procesorů umí odpojit své sběrnice od systému (BUS REQUEST). V takovém případě dokončí aktuální operaci a odpojí datovou i adresní sběrnici. Ke zbytku systému teď může přistupovat jiný obvod. Nejčastěji obvod, který provozuje takzvaný DMA – Direct Memory Access. Pomocí DMA může systém přenášet bloky dat mezi pamětí nebo mezi pamětí a periferií rychleji, než to zvládne procesor sám. Po přenesení požadovaného počtu dat obvod DMA požadavek na sběrnici odvolá, procesor se opět rozeběhne a dál pracuje, jak má.

WAIT

Většina procesorů také umožňuje pozastavení cyklu mezi posláním adresy a zápisem/čtením dat. Pokud je periferie pomalá, po příchodu požadavku na čtení nebo zápis dat informuje procesor, že je potřeba počkat. Procesor začne vkládat čekací cykly, dokud periferie neoznámí, že je připravená poslat nebo přijmout data. Procesor pak pokračuje v cyklu čtení nebo zápisu…

Práce procesoru

Procesory mají vstup RESET, kterým se nastaví do výchozí konfigurace. Po RESETu se nastaví interní registr PC (Program Counter) na výchozí hodnotu (ta se typ od typu liší) a procesor začíná pracovat.

Výše zmíněným postupem přečte hodnotu z adresy, dané hodnotou v registru PC. Tato hodnota je instrukční kód – tedy kód instrukce, kterou má procesor vychovat. Většina představených procesorů používá jednobajtové kódy instrukcí, tedy 00h až FFh, a každá instrukce má právě jeden instrukční kód. Procesory bohatší, Z80 nebo 6809, používají i vícebajtové instrukce, takže se za určitých okolností čtou hodnoty dvě, z adresy PC a PC+1.

Tento první krok, načtení instrukce, se nazývá též M1 – Machine Cycle One, strojový cyklus 1. První strojový cyklus u procesorů 8080 nebo Z80 trvá několik taktů hodin (T). Procesory mívají

vyvedený i signál M1, kterým okolí informuje, že právě probíhá cyklus čtení instrukce. Další přístupy do paměti už čtou a zapisují data.

Po tomto cyklu mohou následovat další strojové cykly (neplést s takty hodin!), pokud je potřeba – například pro načtení nebo zápis dat.

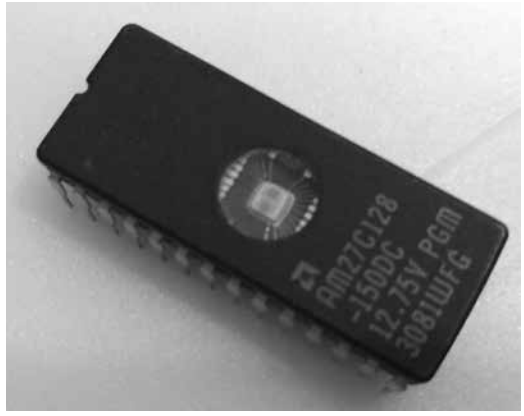
4.2 Paměti

Asi nemá smysl připomínat, že se polovodičové paměti v té době dělily na paměti RAM a ROM, a od pamětí ROM se používaly nejčastěji varianty PROM a EPROM. RAM se zase podle technologie ukládání dělily na SRAM a DRAM.

No dobře, tak rychlá připomínka:

ROM

Paměť ROM se vyznačuje tím, že je *pouze ke čtení*. Tedy že jednou zapsaný obsah se už nemění. Což platilo pro paměti ROM a PROM. Novější technologie (EPROM, EEPROM, FLASH) umožňují změnit zapsaný obsah s větším či menším úsilím. Velká výhoda je, že paměť ROM udrží svůj obsah i po odpojení napájecího napětí. Paměti ROM se proto používaly především pro uložení základních ovládacích programů. Po zapnutí počítače začal procesor provádět instrukce, uložené právě v této paměti.



Obrázek 16: EPROM (foto WestonChants, CC-BY-SA)

ROM (Read Only Memory) je paměť, z níž lze pouze číst data, která do ní uložil výrobce, nejčastěji podle zadání objednatele, tedy v desetitisícových sériích.

PROM (Programmable ROM) je paměť ROM, kterou výrobce dodal nenaprogramovanou a uživatel si ji naprogramoval, jak potřeboval.

EPROM (Erasable PROM) je mazatelná paměť ROM. Dodává se prázdná, uživatel si ji naprogramuje sám, a pokud je potřeba, lze ji opět smazat (většinou ultrafialovým zářením). Poznáte ji podle skleněného okénka uprostřed čipu.

EEPROM přišly později. Jsou to paměti EPROM, které lze mazat i bez UV záření, pouze elektricky (Electrically Erasable PROM). Jejich dalším vylepšením přišly paměti FLASH.

RAM

Paměti RAM (Random Access Memory) jsou paměti, do nichž se dá zapisovat a zase číst stále a stále dokola. Název by měl být přesněji RWM – Read Write Memory, pojem RAM pocházel z doby, kdy nejčastější přepisovatelnou pamětí byla pásková jednotka, u níž se k datům přistupovalo sekvenčně. Polovodičová paměť naproti tomu umožňovala přístup k datům v náhodném pořadí, a tak se toto označení ujalo. Paměť RAM obsahuje jednak data, s nimiž program pracuje, jednak (u von Neumannovy architektury, která se u domácích počítačů používá) i samotné uživatelské programy.

Polovodičové paměti RAM se vyrábějí dvojího typu. V jedněch je informace uložena v klopných obvodech, což je náročnější na prostor v čipu a dražší, ale jednodušší na práci. V druhých se používá parazitní kapacita přechodů, jejich paměťové buňky jsou velmi malé, takže se jich vejde na jeden čip násobně víc, ale zase jsou pomalejší.

SRAM, neboli *Statická RAM,* je RAM vytvořená tak, že paměťovou buňku tvoří klopný obvod. Jakmile je nastaven do určitého stavu, zůstává v něm. Nevýhodou je, že paměti SRAM v době osmibitů mívaly kapacitu tak 1 až 2 kB. Dnes není problém sehnat čipy SRAM s kapacitou 512 kB.

DRAM je *Dynamická RAM*. V dynamické paměti je informace uložena v kondenzátoru (přesněji ve zmíněné parazitní kapacitě přechodu). Nevýhoda je, že při čtení hodnoty paměťové buňky se náboj z toho kondenzátoru odčerpá, takže po každém čtení je potřeba zapsat přečtené informace zpět. Druhá nevýhoda je, že tyto kondenzátory mají velmi malou kapacitu a vybíjejí se samovolně. Vybíjení je poměrně rychlé, a proto musí být paměť *občerstvována*. Občerstvování spočívá v periodickém čtení řádků stále dokola. O toto občerstvování se musely postarat vnější obvody. Výhodou DRAM je na druhou stranu velká kapacita – v době osmibitů to bylo 16 či 64 kilobitů.

Paměti „tenkrát"

V dobách prvních osmibitů nebylo výjimkou, že domácí počítač (natož jednodeskový) měl celý jeden kilobajt paměti RAM (tedy té, do které jste si psali vlastní programy), a k tomu jeden, možná dva kilobajty paměti ROM (nejčastěji technologie EPROM – poznáte ji podle skleněného okénka nad čipem). Například počítač Sinclair ZX-81 měl 8 kB ROM (Sinclair byl dost velký zákazník na to, aby si mohl dovolit vyrobit vlastní sérii ROM), v níž byl uložen BASIC, a 1 kB

statické RAM, v níž byl uložen váš program, a zároveň obsah obrazovky (32 x 24 znaků) – tedy pro samotný program už moc prostoru nezbývalo. Bylo jen logické, že nejčastější používanou periferií byl „memory pack", v němž bylo 16 kB dynamické RAM.

V tomto modulu bylo osm čipů typu 4116 – každý z nich měl kapacitu 16k x 1 bit, a osm dohromady jich dalo 16 kilobajtů. Kromě těchto čipů tu byla i nezbytná logika, která se starala o správné přepínání adres a o občerstvování.

Obrázek 17: Ram pack se 16 kB pro ZX-81

Totiž: statické paměti RAM v té době mívaly kompletní adresovou sběrnici. Třeba obvod 2114 měl 10 adresových vstupů (A0-A9) a 4 datové (D0-D3; měl tedy organizaci 1k x 4). Dva takové obvody „vedle sebe" daly dohromady 1 kB statické paměti RAM.

Dynamické paměti bývaly nejčastěji udělané tak, že měly šířku 1 bit – takže bylo potřeba zapojit 8 obvodů vedle sebe. Navíc neměly všechny adresovací vstupy vyvedené, ale používaly multiplexovanou adresu. Už zmíněná DRAM 4116 měla sedm vstupů adres (A0-A6) a pro vybrání sloužily dva vstupy RAS a CAS (Row Address Select, resp. Column Address Select). Nejprve se na adresní vstupy přivedla vyšší část adresy a aktivoval se vstup RAS. Poté se přivedla nižší část a aktivoval se vstup CAS. Tím byla zadaná kompletní adresa, a paměť vrátila nebo zapsala data.

Bylo to komplikované, to ano, ale tohle je jen část komplikací. Další nastaly s obnovováním – refreshem. Pravidelně bylo totiž potřeba číst jeden řádek paměti po druhém, aby se z nich neztratily data. Pokud byla použita paměť DRAM například pro zobrazování obsahu displeje, nebyl to

problém, protože tam k pravidelnému čtení docházelo tak nějak „z definice“. Ovšem pro zbytek paměti bylo potřeba tohle obnovování vyřešit externími obvody – čítači a multiplexery.

Procesor Z80 měl zabudovanou podporu refreshe – interní sedmibitový čítač R a signál RFSH. Čítač se staral o adresování řádků 0-127 a procesor ve chvíli, kdy měl přečtenou instrukci a dekódoval ji, pustil na sběrnici právě obsah registru R a aktivoval signál RFSH. Logika okolo dynamických pamětí mohla na tento signál reagovat a správně aktivovat potřebné vstupy dynamické paměti.

Jaké paměti použijeme dnes?

Dnes není problém sehnat statické paměti RAM o kapacitách 128 kB – 1 MB za ceny v řádech desetikorun či jednotek stokorun. Vzhledem k tomu, že osmibitové procesory adresují většinou maximálně 64 kB paměti, máme vystaráno a není potřeba vůbec řešit dynamické RAM, obnovování apod.

Jako základní typ použijeme paměť SRAM s organizací 32k x 8, která se označuje 62256. Je to paměť, vyráběná mnoha výrobci pod různými značkami – například já jsem sehnal tyto paměti od výrobce Hitachi s označením HM62256BLP-7. Výrobce Alliance Memory jej značí AS6C62256. Tyto paměti mají přístupovou dobu 70 nanosekund, což je dostatečné pro konstrukce s frekvencemi v řádu jednotek MHz. Pokud chcete použít frekvenci vyšší, sáhněte po rychlejších typech, např. AS7C256B-15 s rychlostí 15 ns.

Pokud ve svém zapojení použijete nějaký způsob stránkování paměti, může se vám hodit paměť s vyšší kapacitou. Pak doporučím typy AS6C1008 (128k x 8) nebo AS6C4008 (512k x 8)

Na pozici ROM můžeme použít buď EEPROM, nebo FLASH. FLASH jsou rychlejší a dnes i dostupnější než EEPROM, ale zase nabízejí až příliš velkou kapacitu, kterou nevyužijeme.

V konstrukcích pracuji s typem 28C256, což je EEPROM s organizací 32 kB x 8. 32 kilobytů je poměrně hodně a stačil by i typ 28C64 s kapacitou 8 kB x 8. Můžete jej použít přímo místo 28C256.

FLASH paměti jsou vhodné třeba ty ze série SST39SF: SST39SF010 (128 kB x 8), SST39SF010 (256 kB x 8) a SST39SF040 (512 kB x 8). A opět platí, ž jejich kapacita bude využita jen zčásti, pokud neimplementujete nějaký obvod pro stránkování paměti.

Velmi zajímavá, levná a malá alternativa jsou sériové paměti RAM, EEPROM a FLASH. Díky tomu, že jsou v malých pouzdrech s osmi vývody, jsou výrazně levnější a zabírají málo místa. Bohužel je nelze připojit k osmibitu přímo jako pracovní paměť – osmibit vyžaduje paměť s paralelním přístupem k datům, tedy kompletní adresa a kompletní data v jednom cyklu.

Konkrétní popis a typy si představíme u jednotlivých konstrukcí.

4.3 Periferie

K čemu by byl počítač, který by si jen tiše pro sebe něco počítal a nikomu nic neříkal? Ačkoli slovo „periferie“ zní hanlivě, jako něco na okraji, tak pro počítač to je zásadní součást: místo, kde se setkává s okolním světem. Bez periferií zkrátka není počítač počítačem. Pojďme se podívat, jaké periferie připadají v úvahu, speciálně pak s přihlédnutím k domácím počítačům z 80. let.

Klávesnice

Klávesnice byla u domácích počítačů naprosto zásadní periferie. Pomocí klávesnice počítač programujete a ovládáte. Velké sálové počítače v předcházející éře klávesnice nezbytně nutně neměly; pracovaly s děrnými páskami a děrnými štítky jako s dávkovými úlohami. Ovšem později, s nástupem terminálů, se vlastně i u nich objevily. Domácí počítače a jednodeskové počítače měly klávesnice od začátku. Některé přímo zabudované, jiné připojovací.

Klávesnice u jednodeskových počítačů byly často řešené podobně jako u kalkulaček – 16 až 25 tlačítek, uspořádaných do matice. 16 tlačítek zadávalo hexadecimální číslice (0-9, A-F), zbývající vyvolávaly některé základní funkce, jako přístup do paměti, k registrům, reset, přerušení nebo práci s externím úložištěm.

U domácích počítačů byl princip stejný: tlačítka, uspořádaná interně do matice, navenek pak jako QWERTY klávesnice s písmeny, číslicemi a dalšími ovládacími funkcemi. U počítačů ZX-80, ZX-81 a ZX-Spectrum byly kupříkladu čtyři řady po deseti klávesách, ale uvnitř zapojené jako matice 5x8 tlačítek. Jiné domácí počítače, například Atari či Commodore, měly kláves podstatně víc, ale vnitřně opět v matici, např. 8x8.

Pokud dneska budeme stavět klávesnici v duchu tehdejších strojů, máme na výběr z několika postupů.

Můžeme použít existující klávesnici ze starého stroje. Občas se v bazarech objevují klávesnice z Commodorů či Atari. Stačí si najít zapojení matice a připojit vhodným způsobem k vlastnímu počítači. Problém je, že takové řešení není moc dobře replikovatelné a pokud konstrukci zveřejníte, budou ostatní limitováni tím, zda seženou stejnou klávesnici.

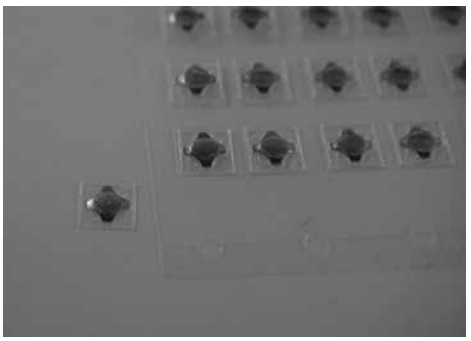
Lze použít i standardní PC klávesnice, ideálně s rozhraním PS/2. Práce s tímto rozhraním je poměrně snadná a zvládne ji i nenáročný jednočip. Klávesnice s USB vyžaduje sofistikovanější technologii…

Třetí možnost je postavit si vlastní klávesnici z tlačítek, například na univerzálním plošném spoji. Stačí pořídit dostatečné množství tlačítek s hmatníky, připájet k desce a propojit.

Poslední dostupná možnost je konstrukce vlastní membránové klávesnice, podobné, jako mělo ZX-81 nebo SAPI-1. Technologicky jde jen o velký plošný spoj, na který je nalepena distanční

mezivrstva a na ní hmatník, který má zespodu vodivé propojky. Třeba nalepené čtverečky alobalu. Vylepšená konstrukce používá malé kovové „kloboučky“ (klíčové slovo: key dome).

Výroba takové klávesnice není technologicky moc náročná, jen je potřeba trocha pečlivosti a spousta trpělivosti. Možný problém pak představuje poměrně velký plošný spoj, takže je to vhodnější postup pro ty konstruktéry, kteří si jej dokáží sami vyrobit, protože malosériová výroba je placena právě od velikosti desky plošných spojů.



Obrázek 18: pružné kovové kloboučky pro klávesnice (Key dome)

Displej

Každý domácí počítač potřebuje pořádný displej. Jenže tady jsme paradoxně s pokrokem v lehce horší situaci.

U jednodeskových počítačů, které používají displej ze sedmisegmentovek, není problém. Sedmisegmentovky stále fungují, stále se vyrábějí a stále není problém je sehnat, dokonce i v podobě modulů s nějakým použitelným rozhraním.

Pokročilejší konstrukce mohou použít monochromatické LCD displeje 16x2, 20x4 nebo grafické 128x64. Připojení těchto displejů jako periferií je velmi jednoduché, protože používají většinou osmibitovou sběrnici a řídicí signály, které jsou se světem osmibitových procesorů kompatibilní. Sice nejsou takové počítače moc „retro“, v 80. letech takové displeje nebývaly moc běžné, ale dnes jsou dostupné a jednoduše použitelné.

V osmdesátých letech byl snad nejčastější displej televizní – totiž prostá černobílá nebo barevná televize, připojená k počítači nejčastěji přes anténní vstup. Kvalita obrazu byla leckdy nevalná: počítač vygeneroval TV signál, většinou v normě PAL nebo NTSC, a tento videosignál se v modulátoru namoduloval na televizní nosnou frekvenci. Ta se po anténním kabelu přenesla do televize, kde ji hned první obvod za vstupem demoduloval zpátky na videosignál. Některé televize měly přímo videovstup, čímž se mezikrok s modulací mohl obejít, pokud tedy počítač měl videovýstup.

Existovaly návody, jak vyvést přímo videosignál u počítačů, které takový vývod neměly, existovaly i návody, jak přibastlit videovstup do televizí, které jej neměly, a šťastní uživatelé si užívali mnohem kvalitnějšího obrazu.

Bohužel dnes je docela problém takto televizi používat. Moderní televize mají sice videovstupů spoustu, včetně takzvaného „kompozitního videa", což je to, co potřebujeme, ale dost často se starými počítači prostě nefungují, protože jsou citlivé na kvalitu signálu a jeho časování, a některé stroje to s tou přesností neměly tak úplně skvělé. Dobré je, pokud máte starou přenosnou barevnou televizi s videovstupem – ale je těžká, křehká, zabírá spoustu místa… Naštěstí se v posledních letech objevily levné displeje, převážně z Číny, s rozumnou úhlopříčkou a se schopností zpracovávat signál v normě PAL. Původně jsou určené do aut pro přenos signálu z parkovacích kamer, ale u starých počítačů fungují úplně krásně.

Asi nejlepší by bylo použít nějaký VGA monitor nebo monitor s HDMI. Bohužel, tyto normy vyžadují mnohem vyšší frekvence signálů, než v amatérské konstrukci s osmibitem dokážeme vytvořit. Ano, existují zapojení, kde miniaturní jednočip AVR generuje videosignál pro VGA, ale rozlišení je poměrně hrubé.

Kazetový magnetofon

Dostáváme se k zásadní periferii, totiž k takové, která dokáže uložit a opět nahrát nějaká data či program (z čistě technického hlediska platí, že program jsou vlastně jen data). Nejlevnější způsob, používaný u domácích počítačů, pracoval s kazetovým magnetofonem coby úložištěm. Některé osmibity měly speciální kazetové magnetofony s možností strojového ovládání (Commodore, Atari), takže si počítač mohl přehrávání zastavit či spustit. Jiné (Sinclair s výjimkou modelu Spectrum +2) používaly obyčejný kazetový magnetofon, který obsluhoval uživatel. Zadal příkaz, spustil kazetu a čekal, dokud se data nenahrála do paměti.

Pokud chcete ukládat data na audiokazetu, musíte je proměnit na zvuk, a důležitá věc je, že musíte být schopni zvuk opět převést zpátky na data. Vznikly různé způsoby záznamu – můžete například logickou 1 reprezentovat jedním pulsem určité frekvence a logickou 0 pulsem jiné frekvence, a při čtení pak měřit čas, který uplyne mezi hranami signálu. Takový záznam používalo například Spectrum. Jiný způsob používal počítač PMD-85, kde se pomocí signálu z UART modulovala nosná frekvence a při čtení po její demodulaci zůstal sériový signál.

Pro osmibity je ukládání na kazety dostatečné a zároveň jednoduché, ale na rovinu přiznám, že bych ho nepoužil. Dnes je docela problém sehnat kazetový magnetofon, natož kazety do něho. Možná by šlo použít třeba diktafon. Například diktafon, v němž se nahrává na SD kartu. Ale pak ruku na srdce: není jednodušší rovnou na tu kartu zaznamenat data?

Disketa

Disketa byla velká výhra, a kdo ji u svého osmibitu měl, ten byl king! Disketám v 80. letech patřila budoucnost – tedy až do 90. let, kdy přišly levnější, spolehlivější a větší média. U domácích počí-

tačů s kapacitou paměti v řádech desítek kilobytů ale taková 5,25" nebo 3,5" disketa se závratnou kapacitou 720 kB (u HD až 1,44MB, ale HD se u osmibitů moc nepoužívaly) představovala efektivní a rychlé úložiště.

Některé počítače – Commodore či Atari – měly od výrobce disketovou mechaniku jako rozšiřující periferii. U jiných se o připojení diskety postarali externí dodavatelé a firmy (například známé mechaniky Beta a Opus u ZX Spectra). Počítače z konce 80. let už měly disketové mechaniky zabudované (Spectrum +3 například).

V Československu disketové mechaniky u domácích počítačů dlouho nebyly rozšířené. Nejen kvůli své ceně (která při individuálním dovozu ze Západu představovala třeba měsíční plat), ale i kvůli tomu, že jste nemohli dost dobře dojít do prvního obchodu s počítačovými potřebami a koupit si krabičku disket…

Disketová mechanika jako periferie není úplně jednoduché zařízení k připojení. Navenek komunikuje hodně „surovým" způsobem, a proto je potřeba použít takzvaný *řadič*, což je obvod vysoké integrace, který dokáže disketovou mechaniku ovládat, tedy správně nastavit hlavičky, zjistit, jak je disketa právě natočená, správně zakódovat data do toku bitů pro disketu (FM či MFM kódování)…

Poznámka: Z hlediska snadnosti připojení na tom paradoxně mnohem lépe jsou pevné disky s rozhraním IDE (ATA) – toto rozhraní šlo připojit skoro „přímo na procesor".

Disketové mechaniky jsou dnes už většinou nostalgickou vzpomínkou, diskety se také těžko shánějí, a pro novou konstrukci bych diskety rozhodně nedoporučil…

SD karta

Dobrá, tak tedy co dnes použít pro ukládání programů? Kazety jsem znectil, diskety taky, tak co zbývá? Asi nejlepší je použít paměťové karty ve standardu SD – to jsou ty malé, co jsou v mobilech, fotoaparátech a dalších zařízeních. Jejich kapacity se dnes pohybují v řádech jednotek až stovek gigabytů, což je pro osmibit až až. Hlavní výhoda, kterou SD karty mají, je ta, že mají velmi jednoduché rozhraní – nepotřebujete přesné časování jako u USB, stačí vám pouze nějak naemulovat rozhraní SPI, a to zvládne i pomalý osmibit. S trochou dobré vůle si naimplementujete i souborový systém FAT, takže přenos dat mezi vaším osmibitem a pracovním počítačem bude snadný.

Tiskárna

Jedna z nejžádanějších periferií té doby byla bezpochyby tiskárna. Ve světě tiskáren se stalo pár dobrých věcí, konkrétně standardizované paralelní rozhraní Centronics (k jehož napojení stačil obyčejný paralelní port) a standardizovaná sada ovládacích příkazů ESC. Nebyl tedy problém připojit tiskárnu, kterou jste někde nějak získali, ke svému *samo domo* vytvořenému řadiči a dopsat si ovladače. Největší problém bylo sehnat tu tiskárnu.

Sériová linka

Sériové rozhraní, pro které se vžilo spousta různých označení (RS-232, i když to ve skutečnosti popisuje něco jiného, nebo třeba UART) bylo u domácích počítačů často k dispozici také. Někteří výrobci dodrželi standard, tedy včetně konektorů, a k takovým počítačům se daly připojit další zařízení se stejným rozhraním, jiní výrobci (Sinclair) když už sériové rozhraní nabídli, tak bylo hodně *proprietární*.

Implementace sériového rozhraní pomocí jednoho obvodu UART není dnes velký problém, a pokud si k němu připojíte převodník UART/USB, tak můžete stolní počítač použít jako sui generis terminál k osmibitu. Takové rozhraní se vždycky hodí., proto ho použijeme i my.

Joystick

Co by byl pořádný domácí osmibit bez joysticku? Česky se tomu říkalo „křížový ovladač" a vlastně šlo jen o držák, připojený ke čtyřem tlačítkům (nahoru, dolů, doleva, doprava) a s jedním, dvěma nebo více tlačítky pro akci („střílení"). Téměř standardem byly joysticky s devítipinovým konektorem CANON, kde byl jeden vodič společný a pět či šest vodičů pro tlačítka (pro každé jedno). Takové joysticky se připojovaly přímo k Atari či Commodoru, pro Spectrum jste potřebovali tzv. „interface", nejčastěji Kempston nebo ZX Interface 2. Uvnitř těchto krabiček byl jednoduchý adresovací obvod a budič sběrnice, nic složitějšího.

Dnes můžeme použít buď analogové joysticky, co se jako modul prodávají za dolar z Číny a technicky jde o dva potenciometry, zapojené do kříže, nebo klasické se spínači, co seženete z výprodeje. A pro různé retroherní automaty či konzole můžete použít buď moderní repliky tehdejších mechanismů, nebo prostá čtyři tlačítka…

4.4 Periferní obvody

PPI 8255

Obvod Intel 8255, nazývaný PPI (Programmable Peripheral Interface), někdy označovaný i jako PIO (Parallel Input/Output) je čip, který umožňuje vytvoření až tří paralelních vstupně-výstupních portů (bran), jejichž funkce lze programem pozměnit.

Obvod 8255 obsahuje tři obousměrné paralelní osmibitové porty PA, PB a PC, proto se označuje jako PIO – Parallel Input and Output. K procesoru se připojuje přes datovou sběrnici a dva adresové vstupy. Je určen pro systémy s procesorem 8080, takže nepřekvapí, že používá řídicí signály /RD a /WR – ty je možné propojit napřímo s výstupy /IOR a /IOW. Dále obsahuje vstup RESET a vstup /CS – známý Chip Select, který připojuje tento obvod ke sběrnici.

Ta nejzajímavější vlastnost obvodu 8255 je, že je konfigurovatelný. Podle kombinace na vstupech A0 a A1 procesor komunikuje buď s registry jednotlivých portů PA, PB, PC, nebo zapisuje a čte do/z osmibitového řídicího registru CWR. Pomocí zápisu do tohoto registru lze

jednak ovládat přímo jednotlivé bity portu PC, ale také nastavit mód práce. Obvod totiž může fungovat jako tři nezávislé datové porty, ale také může rozdělit porty do dvou skupin (PA + polovina PC, PB + druhá polovina PC) a rozdělené signály portu PC využívat jako řídicí signály pro přenos dat, např. pro oznámení, že data byla přijata nebo naopak že je požadováno jejich převzetí apod.

Podrobněji se práci s tímto obvodem budeme věnovat v kapitole o mikropočítači OMEN Alpha.

Z80 PIO

Většina výrobců procesorů kromě klíčového obvodu, totiž procesoru, navrhla a nabídla i větší či menší sadu vlastních periferií. Zilog pro svůj Z80 navrhl čtveřici periferií Z80PIO (paralelní rozhraní), Z80SIO (sériové rozhraní), Z80CTC (čítače/časovače) a Z80DMA (řadič přímého přístupu do paměti). Tyto obvody měly například kompatibilní řídicí vstupy a podporovaly speciální přerušovací systém Z80.

Samozřejmě že můžete v systému s procesorem Z80 použít i periferní obvod 8255, ale ztratíte tím třeba právě možnost využít všech možností přerušení a musíte si řídicí signály pro Z80 překódovat pro 8255.

Obvod Z80PIO nabízí, na rozdíl od 8255, pouze dvě vstupně-výstupní brány. Na druhou stranu má pro každou bránu dedikované řídicí vstupy a dokáže každý jednotlivý bit každé brány nastavit nezávisle na vstup či výstup.

VIA 6522

VIA je akronym pro Versatile Interface Adapter, a jak už číselné označení napovídá, jedná se o periferii, určenou primárně pro procesory z řady 6502. Nabízí opět dva dvousměrné paralelní osmibitové porty, ale dokáže s nimi pracovat i jako s šestnácti nezávislými vstupně-výstupními linkami. Kromě paralelních portů obsahuje obvod i dva čítače / časovače a posuvný registr pro sériové rozhraní.

Existuje i jednodušší předchůdce PIA 6521, bez čítačů a sériového rozhraní. Oba obvody mají analogické čipy v řadě 68xx: 6821 a 6822.

SIO 8251 (USART)

Obvod Intel 8251 se používal v osmibitových počítačích spolu s obvodem 8255. Zatímco 8255 nabízí paralelní rozhraní, 8251 nabízí sériové rozhraní. Sériové rozhraní má několik výhod proti paralelnímu, především tu, že se mnohem líp vede na větší vzdálenosti. Zatímco paralelní rozhraní má limit někde v řádu desítek centimetrů a na metrové vedení už potřebujete velmi dobrý kabel a výkonný budič, pro sériové rozhraní není problém několik metrů. Navíc si vystačí s menším počtem vodičů.

V osmdesátých letech bylo sériové rozhraní velmi oblíbené, protože umožňovalo připojit tehdy populární terminály.

S procesorem komunikuje obvod Intel 8251 opět přes datovou sběrnici a pomocí řídicích signálů /RD, /WR. Obvod se připojuje na sběrnici ve chvíli, kdy je aktivní vstup /CE. Vstup C/D určuje, jestli se zapisují nebo čtou data (=0), nebo řídicí / stavové informace (=1). V praxi se připojuje k některému z adresních signálů, např. k A0. Pokud byl obvod adresován například tak, že byl aktivní na adrese 20h, a vstup C/D byl zapojen na A0, tak se adresou 20h přistupovalo do datového registru, adresou 21h do řídicího.

ACIA 6850

I Motorola pro své procesory 680x připravila obvod pro sériové rozhraní. Jeho velká výhoda proti obvodu 8251 je rychlost a jednoduchost. V počítači OMEN Alpha, kde běží procesor s taktem 1,8432 MHz, dokáže 6850 posílat a přijímat data rychlostí 115200 Bd.

Podrobněji se práci s tímto obvodem budeme věnovat v kapitole o mikropočítači OMEN Alpha.

ACIA 6551

Jde o obvod podobný předchozímu, ale lehce přizpůsobený řídicím signálům procesoru 6502. Proto nepřekvapí, když si jej podrobněji popíšeme v kapitole o mikropočítači OMEN Bravo.

Čítače, časovače a další

Paralelní a sériové porty byly asi nejčastější periferie, s nimiž jste se u osmibitových počítačů setkali. Nejčastěji to byly právě obvody 8255 a 8251, a to i v systémech s jinými procesory, třeba i u Z80. Důvodem byla jejich dobrá dostupnost a snadnost připojení

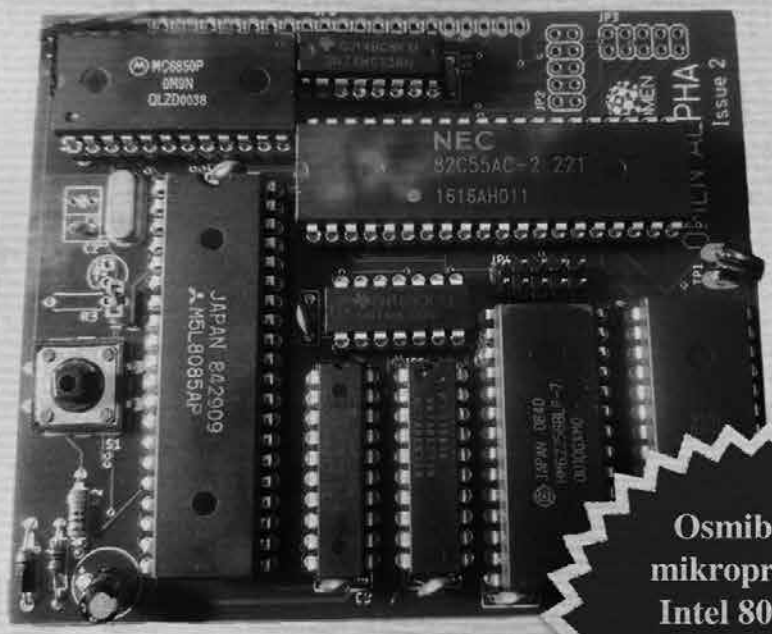
Dalšími používanými periferiemi byly například čítače a časovače 8253 (nebo Z80-CTC). Tyto obvody obsahují programovatelné čítače, které dokázaly počítat nějaké vnější události, např. hodinové impulsy, a při splnění určité podmínky buď změnily signál na výstupu, nebo např. vyvolaly přerušení. Podobné periferie jsou dnes standardní výbavou jednočipových počítačů (mikrokontrolérů).

Mezi periferie se počítaly i zobrazovací jednotky. Některé displeje fungovaly naprosto transparentně, tj. zobrazovaly data, která byla někde v paměti, bez účasti procesoru, jiné bylo potřeba předem přepnout, nastavit do nějakého vhodného pracovního módu – tak fungovaly například videořadiče od Motoroly MC6845 (používané i v grafických kartách MDA a Hercules u IBM PC), nebo např. TMS9918 (výrobce Texas Instruments).

Další periferie, například vnější paměti apod., většinou nemívaly vlastní obvod a připojovaly se přes standardní rozhraní. Výjimkou byly floppy disky (diskety), které měly vlastní řadiče – známé

typy Intel 8272, WD2797 nebo WD1793, které se používaly např. v disketových jednotkách, vyráběných pro počítač ZX Spectrum…

5 OMEN Alpha



Osmibitový mikroprocesor Intel 8085, 32 kilobyte RAM a 32 kilobyte ROM

OMEN ALPHA

Jednoduchý, výkonný, rozšiřitelný

Ideální osmibitový počítač pro každého správného nadšence! Jednoduchý, výkonný, rozšiřitelný. Můžete použít připravená rozšíření, nebo si postavit vlastní periferie, fantazii se meze nekladou. Vhodný pro výuku číslicové a mikroprocesorové techniky i pro výuku programování v assembleru nebo v dalších jazycích (BASIC, C, ...) Díky zabudovaným obvodům pro řízení periferií jej můžete použít i pro řízení dalších komponent.

5 OMEN Alpha

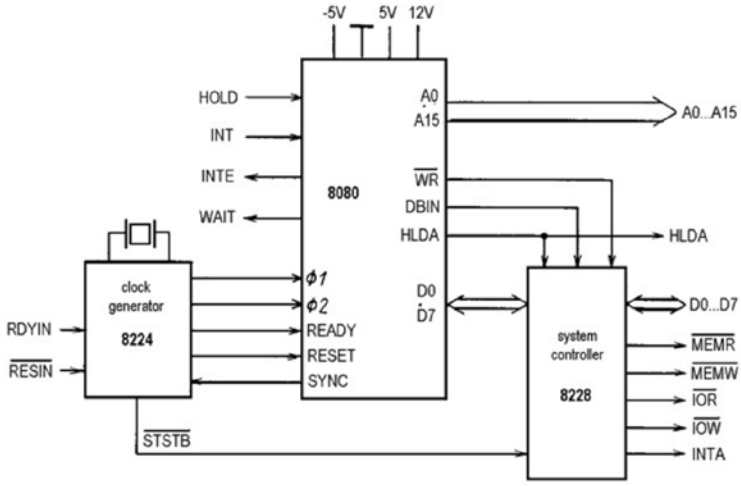
Možná se opakuju, ale napíšu to: k tomu, abyste si postavili počítač, není potřeba nic moc extra. Stačí vám jen nepájivé kontaktní pole, propojovací vodiče, napájecí zařízení a pár součástek: procesor, paměť a nějaké periferie.

Ovšem budu upřímný: v případě procesoru 8080 to tak moc neplatí. Tedy ne že by to nešlo, samozřejmě že to jde, ale problém je v tom, že *procesor 8080* vyžaduje jednak tři napájecí napětí a také nějaké podpůrné obvody.

Z různých technologických důvodů nemá procesor 8080 na čipu integrované některé důležité součásti, především řízení sběrnice a generování hodinových pulsů. Slouží k tomu dvojice externích obvodů 8224 a 8228, a ačkoli je možné se bez nich obejít a postavit si vlastní alternativy, tak takový postup nedoporučuju.

Obvod 8224 slouží ke generování hodinového kmitočtu, ve správné fázi a se správnými poměry period, a k tomu synchronizuje RESET a některé další signály. Obvod bere připojený kmitočet a dělí ho devíti. To je pak pracovní kmitočet procesoru. Pokud má připojený generátor frekvenci 18 MHz, bude 8080 pracovat na frekvenci 2 MHz.

Obvod 8228 generuje z takzvaného *stavového slova*, které procesor posílá po datové sběrnici, a několika *řídicích signálů* plné řídicí signály /MEMR, /MEMW, /IOR a /IOW. Navíc dokáže fungovat jako jednoduchý řadič přerušení – má pouze jednu úroveň a při vzniku požadavku pošle na sběrnici instrukci RST 7 (tedy skok na adresu 0038h, jak si povíme později).



Obrázek 19: „Svatá trojice 8080“

Teprve až tato *svatá trojice*, jak se těmto třem obvodům někdy přezdívá, tvoří samotný „mikroprocesor 8080“.

V tom nejjednodušším systému, kde není zapotřebí řešit přerušení ani pozastavování činnosti či přebírání sběrnice, stačí nepoužité vstupy HOLD a INT připojit na log. 0, vstup RDYIN na log. 1 a výstupy INTE, WAIT, HLDA a INTA ignorovat. Celý procesor pak komunikuje se zbytkem systému pomocí čtyř řídicích signálů /MEMR, /MEMW, /IOR a /IOW, které říkají, že procesor chce číst z paměti (MEMory Read), zapisovat do paměti (MEMory Write), popřípadě číst nebo zapisovat z/do vstupně-výstupních obvodů (Input/Output Read, Input/Output Write).

Přebírání sběrnice, DMA i přerušení budeme probírat později, ale jen pro představu: Přerušení slouží k tomu, aby systém mohl některé vnější události, například příchod znaku na sériový port, stisknutí klávesy nebo uplynutí časového intervalu, rychle obsloužit. Bez tohoto mechanismu je potřeba se stále periodicky dotazovat periferií: „Není zapotřebí obsluha? Nestalo se něco?“

DMA (Direct Memory Access, neboli Přímý přístup do paměti) se používal u osmibitových systémů pro spolupráci s velmi rychlými – na tehdejší poměry – periferiemi, jako byly třeba disketové mechaniky. Ty dodávaly data rychleji, než je procesor stíhal číst a ukládat do paměti, proto se používal tento postup, při kterém externí obvod, zvaný řadič DMA, požádá procesor o odpojení od sběrnic. Procesor se odpojí od datové, adresní i řídicí sběrnice, a řízení přebere řadič DMA, který přímo čte data z periferie a ukládá je do paměti do předem nastavené oblasti.

Jednoduché systémy se bez těchto technik obejdou.

5.1 Architektura 8080

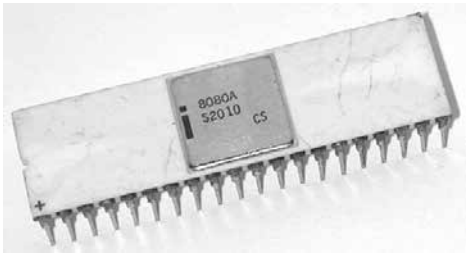
Procesor 8080 byl uveden na trh v roce 1974. Bohužel trpěl několika nedostatky, z nichž snad největší bylo to, že výstupy procesoru snesly jen malé zatížení. Brzy přišla varianta 8080A, která byla naprosto totožná co do vývodů a instrukcí, ale měla posílené vývody a byla rychlejší. Když se dneska hovoří o mikroprocesoru 8080, má mluvčí na mysli nejčastěji právě „áčkovou“ variantu.

Mikroprocesor 8080 byl v jednom pouzdře DIL se 40 vývody.

Z těchto 40 vývodů bylo 16 pro adresovou sběrnici (A0 – A15), 8 pro datovou (D0 – D7), 4 piny sloužily k napájení (zem, +5 V, +12 V, -5 V) a zbytek byly řídicí signály (RESET, INT, WR apod.)

Mikroprocesor pro svou činnost vyžadoval dva podpůrné obvody – 8224, který se staral o správné generování a fázování hodin, a 8228, který posiloval datovou sběrnici a zároveň se staral o gene-

rování řídicích signálů MEMR, MEMW, IOR, IOW a dalších. (MEMx obsluhovaly paměť, IOx vstupně-výstupní zařízení, R znamenalo čtení, W zápis).



Obrázek 20: Intel 8080

Díky šestnácti adresním linkám může mikroprocesor adresovat paměť o kapacitě 64 kB (2^{16}). Z paměti může přenést osmibitové slovo (tedy hodnotu 0 – 255).

V procesoru se nachází několik základních bloků:

- Časovací a řídicí obvody se starají o správnou synchronizaci ostatních částí, o správné vyhodnocení řídicích signálů, co přicházejí zvenčí, a o posílání stavových informací ven.
- Registr instrukcí drží instrukční kód, načtený z paměti, dokud jej nezpracuje dekodér.
- Dekodér instrukcí rozloží instrukční kód na posloupnost základních operací, o jejichž provedení se postará řadič.
- ALU, neboli aritmeticko-logická jednotka, se stará o základní matematické operace (sčítání, odčítání, AND, OR, XOR) s osmibitovými čísly.
- Akumulátor (též registr A) je osmibitový pracovní registr procesoru. S ním se provádí naprostá většina operací, zejména aritmetické, logické či porovnávací.
- Pracovní registry B, C, D, E, H a L jsou registry, které může programátor využít pro dočasné uložení hodnot. Jsou opět osmibitové. Některé instrukce ale dokážou pracovat s tzv. registrovým párem – v takovém případě se berou registry B a C, resp. D a E / H a L jako jeden šestnáctibitový registr. Pár registrů H a L je často používán pro uložení adresy při nepřímém adresování – např. „do akumulátoru se přesouvá hodnota, která je uložena v paměti na adrese, uložené v registrech HL“.
- Ukazatel zásobníku zvaný též registr SP, je šestnáctibitový registr, v němž je uložena adresa tzv. zásobníku. Více si o jeho funkci řekneme v kapitole o podprogramech.
- Programový čítač (registr PC) je šestnáctibitový registr, v němž je uložena adresa, z níž procesor čte instrukci. Po přečtení instrukce se zvýší o 1, 2 nebo 3, takže ukazuje na další instrukci. (Záleží na délce instrukce)
- Příznakový registr F není pro většinu instrukcí samostatně přístupný. Obsahuje několik tzv. příznakových bitů, které udržují informaci o tom, jestli poslední výsledek byl 0, jestli byl zá-

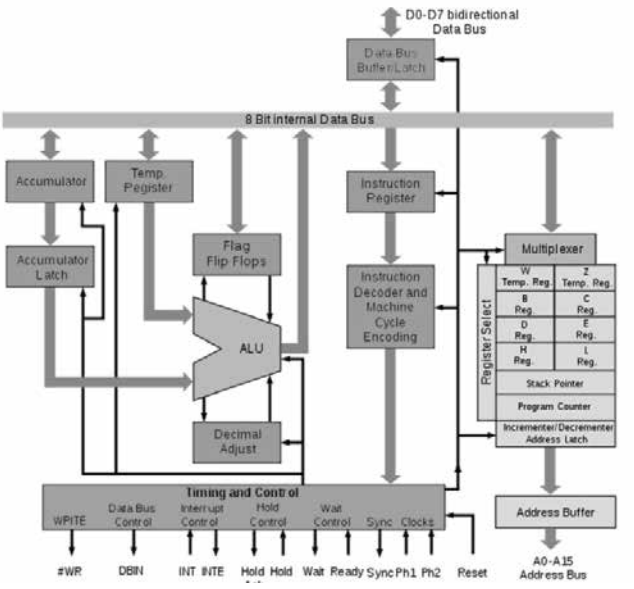
porný, jestli při výpočtu přetekla hodnota přes hranici 255 (nebo pod 0) atd. Podle těchto bitů pak může procesor vykonat tzv. podmíněné instrukce.

Krom těchto částí obsahuje procesor i další skryté registry (W, Z, ACT), ke kterým ale nemá programátor přístup. Mikroprocesor je používá pro ukládání mezivýsledků dle své potřeby.

Registry

15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00	(bitová pozice)
Hlavní registry																**Páry**
A								Flags								Program Status Word
B								C								B
D								E								D
H								L								M (Nepřímá adresa)
Ukazatel zásobníku – SP																Stack Pointer
Programový čítač – PC																Program Counter
Stavový registr – F								S	Z	-	AC	-	P	-	C	Flags

Fyzická architektura



Obrázek 21: Architektura 8080

Uprostřed schématu je aritmeticko-logická jednotka (ALU). Nalevo od ní je pracovní registr – akumulátor a registr pro ukládání mezivýsledků. Nad ALU je blok příznakových bitů (Flag Flip Flops), pod ní kombinační logika pro převod mezi binárními a decimálními hodnotami. Napravo je registr pro instrukci a instrukční dekodér. Pravá část schématu obsahuje blok pracovních registrů, čítač instrukcí PC a další registry. Kromě těchto částí obsahoval procesor ještě obvody pro řízení datové sběrnice, pro řízení adresové sběrnice a obvody časování a řízení běhu procesoru.

Procesor byl osmibitový, to znamená, že pracoval s údaji o šířce 8 bitů. Navenek komunikoval obousměrnou datovou sběrnicí D0-D7. Směr komunikace určoval procesor podle toho, jestli potřeboval číst data, nebo je zapisovat. Procesor dokázal adresovat 64 kB paměti (k tomu sloužila adresová sběrnice s šestnácti signály A0-A15).

Z hlediska programátora jsou zajímavé právě pracovní registry B, C, D, E, H a L, akumulátor, ve kterém se provádějí všechny aritmetické i logické instrukce, ukazatel zásobníku a registr příznaků, který říká, jak dopadla poslední operace, a který slouží k podmíněným skokům (například „skoč, pokud výsledkem předchozí operace je 0“).

Obvod 8080A se nejčastěji používal s dvojicí podpůrných obvodů 8224 a 8228. První jmenovaný se staral o správné vytváření hodinových pulsů (8080A vyžaduje dva nepřekrývající se hodinové pulsy) a synchronizaci signálů, druhý jmenovaný se staral o řízení sběrnice a vytváření základních řídicích signálů /MEMR, /MEMW, /IOR a /IOW.

Signály /MEMR a /MEMW říkají, že procesor chce číst z paměti (MEMory Read) nebo do ní zapisovat (MEMory Write). Signály IOR a IOW oznamují, že procesor chce číst nebo zapisovat do obvodů pro vstup a výstup. Takovéto rozdělení umožňovalo využít celý prostor 64 kB pro paměť, a další prostor 256 adres pro různé periferie (k nim se ještě dostaneme).

Po spuštění procesoru se většinou vnější obvody postaraly, aby ze všeho nejdřív přišel signál RESET. Tento signál uvedl procesor do výchozího stavu, což technicky znamenalo především nastavit čítač instrukcí (PC) do stavu 0000h. Pak začal procesor provádět instrukce.

První hodinový puls se nazýval FETCH a jeho úkolem bylo vyzvednout kód instrukce z paměti. Řídicí logika v tomto cyklu přepne datovou sběrnici na vstup, na adresovou sběrnici pustí hodnotu čítače instrukcí PC a aktivuje signál /MEMR – čtení z paměti. Kdesi venku, mimo procesor, musí být připojená paměť, většinou EPROM nebo RAM, a v ní uložený program. Paměť je připojena na adresovou i datovou sběrnici, a povolovací vstup /CE je připojen právě na /MEMR. Když procesor tento signál aktivuje, paměť na výstup vystaví hodnotu, kterou má uloženou na dané adrese – v tomto případě na adrese 0000. Technicky je to prostě jen osmibitová hodnota, například D3h – binárně 11010011.

Procesor tuto hodnotu vidí na datových vstupech a uloží si ho do registru instrukcí. To je zase náš známý osmibitový klopný obvod typu D. Na jeho výstupu je připojen velmi složitý kombinační

obvod, který z těchto osmi bitů pozná, co má udělat, a podle toho v dalších cyklech aktivuje jednotlivé vnitřní komponenty. Například na vstupy ALU připojí akumulátor a některý z pracovních registrů, na řídicí vstup ALU nastaví operaci „sečti“, a v dalším kroku uloží výsledek opět do akumulátoru. U některých instrukcí si může vyžádat další čtení z paměti nebo zápis do ní. Když vykoná procesor vše, co daná instrukce předepisuje, zvýší hodnotu PC o 1 a celý cyklus se opakuje. U některých instrukcí (skoky) se PC nezvyšuje, místo toho je do tohoto registru zapsána nová hodnota.

Ready / Wait

Některé periferie, popřípadě pomalé paměti, vyžadují, aby procesor chvilku počkal, protože nemají požadovaná data hned. V takovém případě uvedou vstup READY – tedy „připraveno“ – do 0. Procesor z toho pozná, že data nejsou připravena, a čeká. Na adresní sběrnici je stále vystavená adresa. Procesor čeká do té doby, než bude signál READY zase 1. Během čekání uvede svůj výstup WAIT do 1, a tím oznamuje, že čeká na data. Jakmile má periferie data připravená, pošle je na datovou sběrnici a uvolní signál READY zpět do log. 1. Procesor tím vystoupí z čekací smyčky, přečte data a pokračuje dál v provádění instrukce.

Hold (DMA)

Procesor umožňuje některým periferním obvodům, aby převzaly kontrolu nad sběrnicí. Slouží k tomu signál HOLD. Jakmile je aktivovaný, dokončí procesor aktuálně prováděný cyklus a vstoupí do stavu HOLD. V tomto stavu se odpojí od adresové i datové sběrnice a aktivuje výstup HLDA (HOLD Acknowledge). Jakmile je HLDA aktivní, může periferní obvod přebrat řízení sběrnice. Nejčastější scénář použití stavu HOLD je při takzvaném přímém přístupu do paměti (Direct Memory Access, DMA). Používá se ve chvíli, kdy některá rychlá periferie (nejčastěji to bývaly disky, zvukové obvody nebo videoobvody) potřebuje zapisovat nebo číst data. Speciální obvod (DMA Controller) obdržel např. informaci o tom, že periferie, například disk, má připravená data k přenosu. Požádal procesor o přístup na sběrnici (HOLD), a jakmile procesor potvrdil, že je odpojen, začal tento obvod číst data z periferie a zapisovat je do paměti mnohem větší rychlostí, než by to dokázal samotný procesor.

Práce s takovým obvodem pak pro programátora vypadala tak, že do DMA Controlleru nastavil adresu, kam se budou data zapisovat nebo odkud se budou číst, řekl, kolik dat očekává, a oznámil periferii, že požaduje např. data z nějakého diskového sektoru. Pak pokračoval v práci. Ve chvíli, kdy periferie měla data připravená, přenesl je výše popsaným způsobem DMA Controller do paměti, a pak dal procesoru vědět, že je hotovo. Jakým způsobem? Třeba přerušením!

Přerušení

Přerušení je velmi podstatný koncept v procesorovém světě. Jde o způsob, jakým dává okolí vědět procesoru, že se stalo něco, na co je třeba bezprostředně zareagovat. Může to být například informace o tom, že na klávesnici někdo stiskl klávesu, může to být signál, že přišla data po sériové lince, nebo může jít o pravidelný hodinový signál. Může jít o naprogramovaný signál typu „za 10 milisekund“, nebo může jít třeba o výše zmíněný signál od obvodu DMA.

U procesoru 8080 slouží k vyvolání přerušení vstup INT. Na konci každé instrukce procesor testuje stav tohoto vstupu. Pokud je aktivní, pozná, že jde o přerušení. Pokud není přerušení zakázáno (což může programátor zařídit speciální instrukcí), udělá to, že přečte jednu instrukci z datové sběrnice, zakáže další přerušení a instrukci vykoná.

Tou jednou instrukcí bývala instrukce RST 0 až RST 7 – speciální obvod se staral o to, aby různé požadavky na přerušení měly různé instrukce. Instrukce RST jsou vlastně zkrácené instrukce volání podprogramu na adresách 0, 8, 10h, 18h, … 38h.

Pro jednoduché systémy šlo zařídit, aby vždy přerušení vyvolalo instrukci RST 7, tedy volání podprogramu na adrese 0038h. Po příchodu požadavku na přerušení tedy procesor skočil do podprogramu na adrese 0038h, tam musel programátor požadavek vhodně obsloužit, pak opět přerušení povolit a vrátit se zpět do hlavního programu.

O toto jednoduché přerušení se staral obvod 8228, který při začátku přerušovacího cyklu poslal na datovou sběrnici hodnotu FFh, což je operační kód instrukce RST 7.

Operace, cykly T a M

Po zapnutí napájení nebo po signálu RESET je procesor uveden do základního stavu. Je zakázáno přerušení a do registru PC (šestnáctibitový ukazatel na instrukci) je uložena nula. Ostatní registry mají obsah takový, jaký měly před RESETem (po zapnutí napájení tedy náhodný).

Hodiny běží… Co se v procesoru děje?

Probíhá první strojový cyklus (zvaný M1). V tomto cyklu procesor pošle na adresovou sběrnici obsah registru PC (teď to je 0), zvýší obsah tohoto registru o 1 a pošle požadavek „čtení z paměti“ (MEMR). Systém okolo by se měl postarat, aby na tento požadavek odpověděla paměť a poslala po datové sběrnici obsah na dané adrese. Mikroprocesor si tento obsah přečte, uloží do registru instrukcí, dekóduje a provede požadovanou činnost. Někdy není potřeba žádný dodatečný čas, jindy se například zapisuje do paměti nebo z ní čte, takže procesor zařadí některé z cyklů M2, M3, M4, M5. Po dokončení instrukce se celý proces opakuje.

V případě, že instrukce vyžaduje nějaký parametr, následují cykly čtení z paměti. Instrukce může jako parametr požadovat 1 nebo 2 bajty. Ty jsou načteny z adresy PC+1 a PC+2. PC je zvýšen o 2 nebo 3. Instrukce je pak provedena (což si opět může vyžádat nějaké strojové cykly) a po jejím dokončení se celý proces zase opakuje.

Strojový cyklus M1 se skládá ze čtyř nebo pěti taktů systémových hodin. Protože každá instrukce obsahuje vždy cyklus M1, v němž je načtena a dekódována, tak je jasné, že nejrychlejší instrukce zabere čtyři hodinové cykly (zapisuje se jako 4 T). Pokud běží hodiny na frekvenci 2MHz, tak bu-

dou ty nejrychlejší instrukce prováděny s frekvencí 500kHz (tj. každá bude trvat 2 mikrosekundy). Nejpomalejší instrukce zaberou 18 taktů (18 T). Informace o tom, jak je která instrukce „dlouhá" (tj. kolik T trvá procesoru, než ji zpracuje) je důležitá ve chvíli, kdy je třeba optimalizovat kód a zrychlit jej.

5.2 Jak komunikovat s okolím?

Intel samo sebou, stejně jako další výrobci mikroprocesorů v té době, vyvinul i podpůrné obvody, které více nebo méně sofistikovaně poskytovaly funkce, potřebné pro komunikaci procesorového systému s okolním světem.

Čistý počítač sám o sobě totiž moc parády nenadělá. Mikroprocesor provádí program, uložený v paměti, manipuluje s daty, uloženými tamtéž, a navenek je pozorovatelné pouze to, že systém pálí elektřinu. To, co dělá z počítače užitečný stroj, je schopnost komunikovat s okolním světem.

Periferií může být velké množství, jak jsme si řekli už v kapitole o periferiích. Teď sestoupíme o kousek níž a podíváme se, jak jsou připojeny.

Periferie se nejčastěji připojují na systémovou sběrnici, a to přes oddělovací obvody. Tyto obvody se starají o to, aby ve správný čas byla na sběrnici připojena správná periferie správným směrem. Nejjednodušší způsob, jak takové oddělení zařídit, bylo připojit k datové sběrnici osmibitový jedno- či obousměrný budič s třístavovými výstupy. Na povolovací vstup pak stačilo připojit dekódovanou adresu…

Dekodér adres

Velmi podstatná součást osmibitového systému je dekodér adres. Systém totiž většinou obsahuje více druhů paměti (RAM, ROM) a různé periferie. Dekodér adres pak říká, na jaké adrese v adresním prostoru je připojená jaká periferie či paměť.

Nejjednodušší případ nastává, když máme třeba v systému s procesorem 8080, který má 16bitovou adresní sběrnici, pouhé dva čipy paměti – jeden RAM, jeden ROM, oba 32 kB. Můžeme je připojit vedle sebe *jedna ku jedné* a nejvyšší bit adresy (A15) použít jako dekódovací: pokud bude 0, bude připojena ROM a odpojena RAM, pokud bude 1, bude připojena naopak RAM a odpojena ROM.

Tím se ROM objeví pro procesor na adresách 0000 – 7FFFh, RAM na adresách 8000h – FFFFh.

Ovšem ne vždy je zapojení takto jednoduché. Často se používalo třeba dělení prostoru na čtvrtiny, kdy v nejnižších 16 kB sídlila ROM, v následujících pak RAM, a většinou v blocích (fyzických) po 16 kB. Bylo proto potřeba udělat dekodér 1-ze-4, na jehož vstupy se přiváděly nejvyšší bity adresy A14, A15, a výstupy pak sloužily jako *chip select* pro jednotlivé fyzické paměti.

Nejčastěji (alespoň v československých konstrukcích) se používal obvod 3205. Ten byl původně navržený Intelem pro procesorovou řadu 3000. V Československu ho vyráběla Tesla, a proto se používal. Alternativou byl obvod 74138 ze standardní řady TTL 74xx, popřípadě obvod 8205, navržený právě pro řadu procesorů 8080. Ale výsledek byl shodný – tedy až na to, že 3205 měl mnohem vyšší spotřebu.

Obvod 3205 má tři adresové vstupy A, B, C, tři povolovací vstupy /E1, /E2 a E3 a osm výstupů /Y0 až /Y7 (někdy značených i /Q0 až /Q7). Výstupy jsou negované, aktivní v logické 0, což je výhoda, protože zároveň většina periferních obvodů má povolovací vstupy aktivní v 0.

3205 se používal nejčastěji připojený na horní 2 nebo 3 bity adresní sběrnice procesoru, čímž umožňoval rozdělit prostor na 4 či 8 částí. Povolovací vstupy se pak připojovaly k signálům /MEMR, /MEMW apod., popřípadě se připojily k dalším bitům adresy, aby se ještě víc omezil prostor…

Používal se rovněž pro periferie. Ty mají u procesoru 8080 adresu pouze osmibitovou. Pro jednodušší sestavu s malým počtem periferií stačilo třeba použít systém „co adresní vodič, to jedna periferie“ a použít takzvané neúplné dekódování adresy.

Neúplný dekodér fungoval např. tak, že se řeklo, že adresní vodič A0 vybírá periferii X a vodič A1 vybírá periferii Y, obojí v log. 0. Na adrese FEh (1111 1110b) se pak, logicky, objevila periferie X, periferie Y pak byla na adrese FDh (1111 1101b). Ovšem nejen tam. Periferie X se objevovala i na všech možných adresách, které splňovaly pravidlo XXXX XX10b. Tedy na adrese 02h, 06h, 0Ah, 0Eh, 12h, …

Co když byly oba bity, tedy A0 i A1, nulové? Co se stalo, když jste třeba četli z adresy 00h? Pak došlo ke kolizi na sběrnici, obě periferie přistupovaly najednou a výsledek byl nepredikovatelný. Proto bylo lepší používat úplnou adresaci, ale někdy, většinou z důvodů ekonomických, se na dekodéru šetřilo, a pak bylo na programátorovi, aby použil takovou adresu, která s ničím nekolidovala…

Paralelní port

Máme tedy procesor, paměti a dekodér adres. Vytvoření jednoduchého paralelního portu pak není problém. Stačí vzít obvod typu „osmibitový klopný obvod D“, ideálně „s posíleným výstupem“ a připojit jej na sběrnici. Hodinový vstup připojíte na dekódovanou adresu, a je to!

V praxi se používaly obvody 3212/8212, které jsou osmibitové, ale jednosměrné. Druhá alternativa byla použít dva obvody 3216, které jsou sice jen čtyřbitové, zato obousměrné.

Paralelní port stačil pro většinu tehdejších periferií – pro klávesnici, pro zobrazovací jednotky apod. Nevýhoda takto jednoduchého portu ale byla ta, že nenabízel žádný způsob řízení komunikace, neumožňoval třeba vyvolat přerušení na vnější požadavek atd. Proto se používaly složitější programovatelné obvody, jako například PPI 8255.

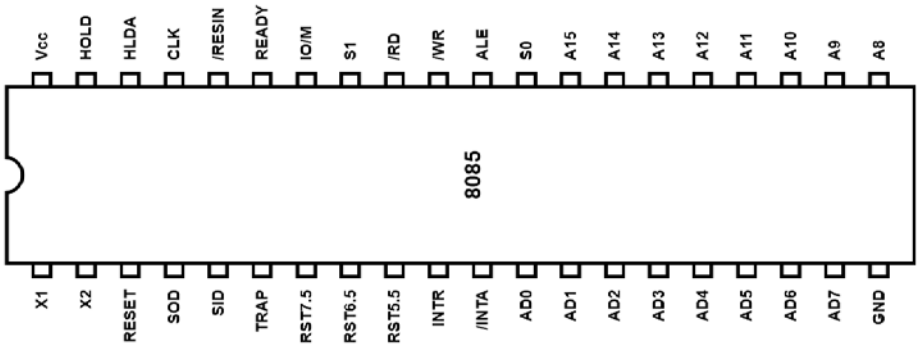
5.3 Vlastní počítač? Jak by mohl vypadat?

Ačkoli byl 8080 široce používaná „klasika“, pro novou konstrukci bych ho nedoporučil – minimálně kvůli třem napájecím napětím. Sáhl bych po lehce rychlejším a vylepšeném procesoru 8085, respektive jeho CMOS verzi 80C85 (vyrábělo hned několik výrobců). Procesor 8085 si vystačí s pouhým jedním napájecím napětím (standardních +5 voltů) a nepotřebuje speciální obvody pro generování hodin (stačí mu obyčejný hodinový puls, nebo dokonce jen připojit krystal s kondenzátory). Navíc má dva vývody pro sériový vstup a výstup a celkem pět přerušovacích vstupů – kromě INTR ještě vstup TRAP (nemaskovatelné přerušení) a tři vstupy RST (RST5.5, RST6.5 a RST7.5).

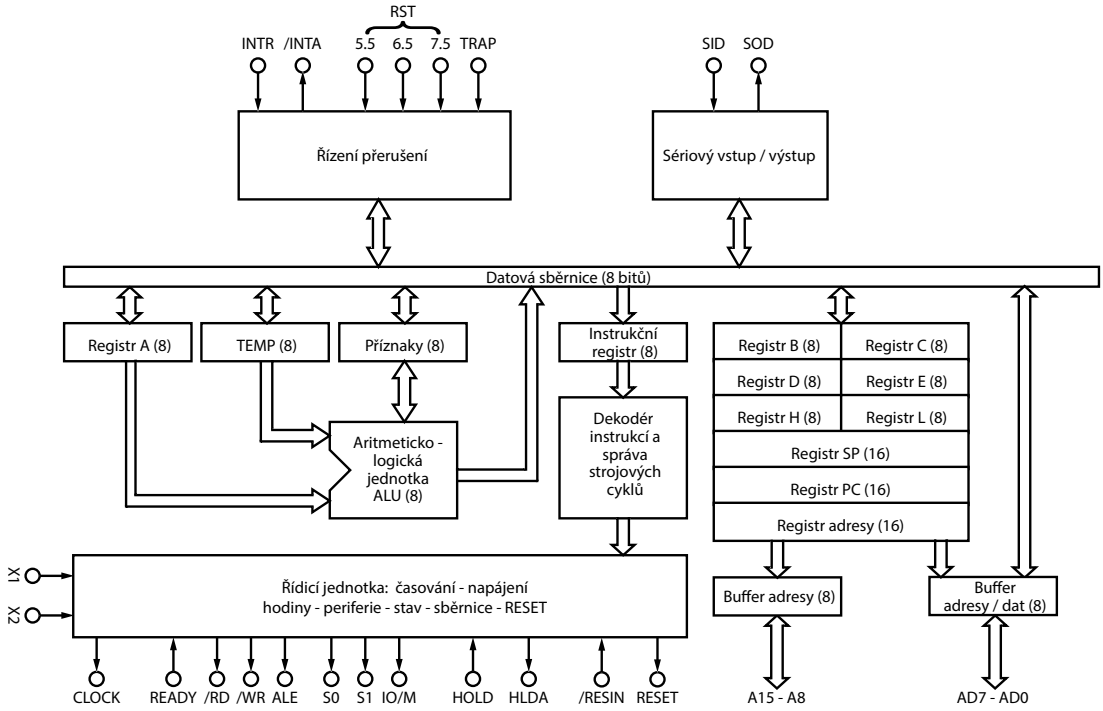
Ani 8085 nedokáže pracovat úplně bez vnějších obvodů: návrháři totiž kvůli nutnosti vejít se do pouzdra se 40 vývody sáhli k takzvanému multiplexovanému adresování. Procesor má vyvedených osm vyšších bitů adresy (A8-A15) a nižších 8 bitů se vede po datové sběrnici (vývody AD0-AD7). Adresu posílá procesor nadvakrát. K řízení slouží signál ALE (Address Latch Enable). Pokud je v log. 1, znamená to, že procesor posílá po AD0-AD7 dolních 8 bitů adresy. Pokud je v log. 0, fungují tyto vývody jako datová sběrnice.

Návrhář systému proto musí zajistit vnější obvod (osmibitový latch), který se postará o to, aby zachytil spodních 8 bitů adresy. Typicky lze použít třeba obvod 74573 (starší konstrukce používaly např. dva obvody 7475).

5.4 Vývody 8085



5.5 Vnitřní struktura 8085



Obrázek 22: Architektura 8085

Na první pohled se od 8080A moc neliší – i 8085 má sadu registrů B, C, D, E, H, L, akumulátor A, Stack pointer SP, programový čítač PC, ALU, řídicí obvody, řadič… Navíc proti 8080 je rozšířena část řízení přerušení (o vstupy RST a TRAP), část sériového výstupu (SID, SOD) a zabudovaný generátor hodin.

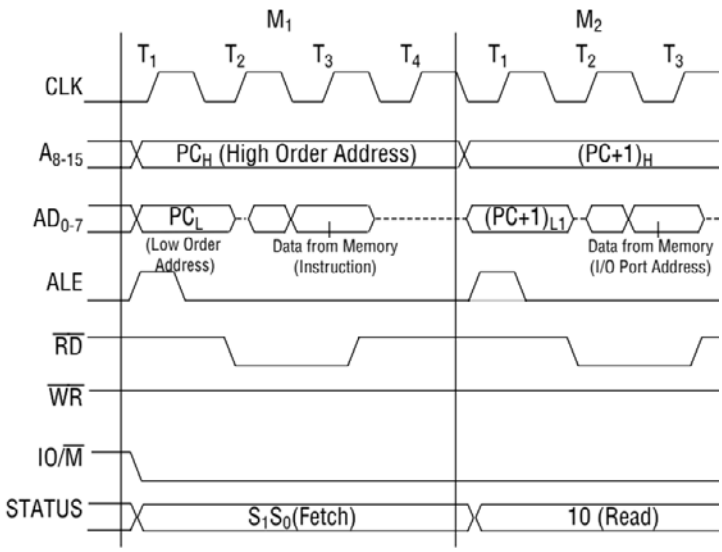
Dole je vidět zmíněné rozdělení adresní sběrnice na dvě části – samostatných 8 horních bitů A15-A8, a multiplex pro spodních 8 bitů adresy / 8 bitů dat (AD7-AD0).

Řídicí signály mají následující funkce:

Signál	Funkce
A8-A15 (výstup, třístavový)	Horní část adresy. Během stavů HOLD, HALT a při RESETu je ve stavu vysoké impedance
AD0-AD7 (vstup/ výstup, třístavový)	Nižších osm bitů adresy, popřípadě datová sběrnice
ALE (výstup)	Address Latch Enable. Oznamuje, že na vývodech AD0-AD7 je nižší část adresy (1). Pokud je 0, znamená to, že AD0-AD7 funguje jako datová sběrnice
S0, S1 (výstupy)	Stavová informace. Spolu se signálem IO/M říká, jestli procesor čte instrukci, jestli čte / zapisuje data do paměti, jestli komunikuje s periferií, jestli obsluhuje přerušení nebo že je zastaven. V jednoduchých konstrukcích se nepoužívají.
IO/M (výstup)	Oznamuje, jestli procesor hodlá komunikovat s pamětí (0), nebo s periferiemi (1)
/RD (výstup, třístavový)	Pokud je 0, procesor čte data z vnějšího zařízení (paměti, portu). Vnější zařízení musí požadovaná data poslat na datovou sběrnici.
/WR (výstup, třístavový)	Pokud je 0, procesor chce zapsat data, co poslal na datovou sběrnici, do portu nebo do paměti.
READY (vstup)	V dobách, kdy paměti a periferie byly pomalejší než procesor, mohly poslat během cyklu čtení nebo zápisu signál „počkej, než budou data připravena“. Slouží k tomu právě vstup READY. Pokud je 1, procesor jede na plný výkon. Pokud je 0, procesor po nastavení signálu /RD či /WR počká, dokud nebude READY zase 1, pak teprve pokračuje dál. My ho klidně připojíme na 1 a budeme věřit, že moderní paměti jsou dost rychlé…
HOLD (vstup)	Normálně 0. Pokud je nastaven na 1, procesor dokončí nejnutnější operaci (cyklus čtení nebo zápisu) a odpojí svoji datovou sběrnici, adresní sběrnici a signály /RD, /WR a IO/M. Odpojením mám na mysli přepnutí do stavu Z (vysoké impedance). Jakmile je procesor odpojen, potvrdí, že je vše hotovo, signálem HLDA. Jednoduché systémy tento signál nepoužívají, můžete jej připojit na log. 0
HLDA (výstup)	Za normálního provozu 0. Pokud byl dán požadavek HOLD, tak se po dokončení nezbytných operací a odpojení sběrnic tento signál nastaví do log. 1.
INTR (vstup)	Interrupt request, tedy Požadavek na přerušení. Procesor testuje stav tohoto signálu na konci každé instrukce a také ve stavu HALT (po instrukci HLT) a HOLD. Pokud je 0, nic se neděje, pokud je 1, začne proces obsluhy přerušení. Procesor přečte z datové sběrnice hodnotu a považuje ji za instrukci, kterou vykoná. Nejčastěji se posílají jednobytové instrukce RST, ale je možné poslat i tříbytovou instrukci CALL. O poslání instrukce se musí postarat vnější obvody, takzvané řadiče přerušení. Programem je možné zakázat vyvolání přerušení tímto signálem.

Signál	Funkce
/INTA (výstup)	Interrupt Acknowledge. Po příchodu požadavku přerušení INTR použije procesor výstup /INTA k načtení výše zmíněné instrukce (namísto signálu /RD)
RST5.5, RST 6.5, RST 7.5 (vstupy)	Přímé vstupy přerušení. Fungují podobně jako signál INTR, ale nevyžadují celý ten opruz s načítáním instrukce pomocí /INTA. Místo toho si samy interně zařídí skok na patřičná místa. Stejně jako INTR dokážou „probudit" procesor ze stavu HALT a HOLD. Programátor může tyto signály ignorovat pomocí speciální instrukce („maskování přerušení")
TRAP (vstup)	TRAP je podobný signálům RST x.5 s tím rozdílem, že toto přerušení nemůže programátor zamaskovat.
/RESET IN (vstup, Schmitt KO)	Vstup nulování. Po startu systému nebo při havárii by měly vnější obvody tento signál, který je normálně v log. 1, přepnout do log. 0. Tím započne interní proces inicializace: procesor odpojí sběrnice, nastaví programový čítač na adresu 0000h, ukončí všechny případné čekací cykly a začne pracovat „od začátku". Tento vstup je vybavený Schmittovým klopným obvodem, to znamená, že je možné k němu připojit přímo oblíbený obvod s rezistorem a kondenzátorem, který se postará o RESET při zapnutí napájecího napětí.
RESET OUT (výstup)	Indikuje, že je procesor ve stavu RESET. Tento signál je synchronizovaný s hodinami a můžete ho použít jako systémový RESET pro zbytek systému.
X1, X2 (vstupy)	Slouží k připojení hodinového krystalu. Jeho frekvence je interně vydělena dvěma a výsledek dává systémový takt. Vstup X1 může být použit i jako vstup hodinového signálu z externího obvodu.
CLK (výstup)	Systémové hodiny. Jejich frekvence je poloviční proti frekvenci na vstupu X1 (tedy třeba proti frekvenci připojeného krystalu)
SID (vstup)	Sériový vstup. Hodnota na tomto vývodu je načtena do nejvyššího bitu akumulátoru pomocí instrukce RIM.
SOD (výstup)	Sériový výstup. Hodnota na tomto vývodu je nastavena nebo nulována pomocí instrukce SIM.
Vcc	Napájecí napětí. Nominálně 5 voltů, u CMOS verze může být někde mezi 3 V a 6 V.
GND	Zem

V datasheetu (je jich spousta různých od různých výrobců, já bych doporučil datasheet od OKI, jejich čip nesl název MSM80C85AH) najdete i velmi důležitý graf, který ukazuje průběhy signálů na sběrnici. Podívejme se na něj spolu:



Obrázek 23: časování 8085

První řádek ukazuje hodinový signál. M1 je první strojový cyklus, T1 až T4 jsou jeho hodinové takty. Při vyzvedávání instrukce tedy procesor řídí výstupy takto:

T1: Nastaví horní část adresy instrukce na výstupy A8-A15, dolní část na výstupy AD0-AD7, a zároveň aktivuje signál ALE (Address Latch Enable). Tento signál řídí už zmíněný buffer mimo procesor, jehož úkolem je zapamatovat si právě spodní část adresy. Zároveň nastavuje signál IO/M do hodnoty 0, tedy „komunikace s pamětí". Signály /RD a /WR jsou neaktivní, stavové signály nás nemusí zajímat. Se sestupnou hranou hodin v čase T1 se deaktivuje signál ALE a dolní část adresy zůstává zachycena v bufferu.

T2: Procesor přepíná signály AD0-AD7 na vstup dat a aktivuje signál /RD, což informuje okolní systém, že procesor hodlá číst (/RD) z paměti (IO/M je v log. 0). Paměť je připojena ke sběrnici a vybrána adresa.

T3: S náběžnou hranou hodin přečte procesor stav na datové sběrnici a začne jej vyhodnocovat coby instrukci. Poté deaktivuje signál /RD, takže se paměť může opět odpojit.

T4: Procesor dekóduje instrukci.

Dejme tomu, že instrukce potřebuje jeden osmibitový parametr. Ten je uložen, jak bývá zvykem, na následující adrese. Následuje tedy strojový cyklus M2, během něhož jsou čtena data z paměti.

Když se podíváte zase na schéma, vidíte, že je téměř totožný s cyklem M1, až na to, že trvá pouhé tři takty – odpadá „dekódovací“ takt.

5.6 Přerušení

U 8085 k práci s přerušením slouží hned tři mechanismy.

První je identický jako u procesoru 8080 – vnější zařízení nastaví signál INTR, procesor si přečte z řadiče přerušení jednu instrukci a vykoná ji. Nepřekvapivě jde o instrukce skoku do podprogramu (pro programátory, zvyklé na vyšší jazyky: něco jako vyvolání handleru, volání funkce atd.) O správné poslání instrukce se musí postarat vnější obvody, většinou specializované obvody, nazývané řadiče přerušení.

U 8080 šlo tuto složitost zjednodušit pomocí obvodu 8228 – připojením jednoho z vývodů (INTA) přes rezistor 1 K k napětí 12 voltů se z tohoto obvodu stal jednoduchý „řadič přerušení“, který po příchodu požadavku na přerušení poslal na sběrnici hodnotu FFh, což je hodnota, kterou si procesor dekóduje jako instrukci RST 7. (RST 7 uloží aktuální adresu na zásobník a udělá odskok na adresu 0038h, ale o tom víc později). 8085 tuto vymoženost nemá, takže buď použijete řadič přerušení, nebo si vystačíte s interními vstupy RSTx a TRAP.

Druhý způsob je použití vstupů RSTx. Jejich funkce je podobná té, co jsem popisoval u obvodu 8228 o odstavec výš – vygenerují si skokovou instrukci samy. Ono to není tak úplně doslova, protože mechanismus je lehce jiný, ale princip je stejný.

Vstupy RST 5.5 a RST 6.5 (asi klidně můžeme říkat pět a půl a šest a půl) jsou, podobně jako vstup INTR, pravidelně kontrolovány, a jsou-li ve stavu 1, procesor vyvolá přerušení.

Vstup RST 7.5 má, na rozdíl od předchozích, zabudovaný vnitřní klopný obvod, který reaguje na vzestupnou hranu. Výhoda je, že stačí jen krátký signál a procesor si jej interně „podrží“ až do doby, kdy testuje přerušovací vstupy.

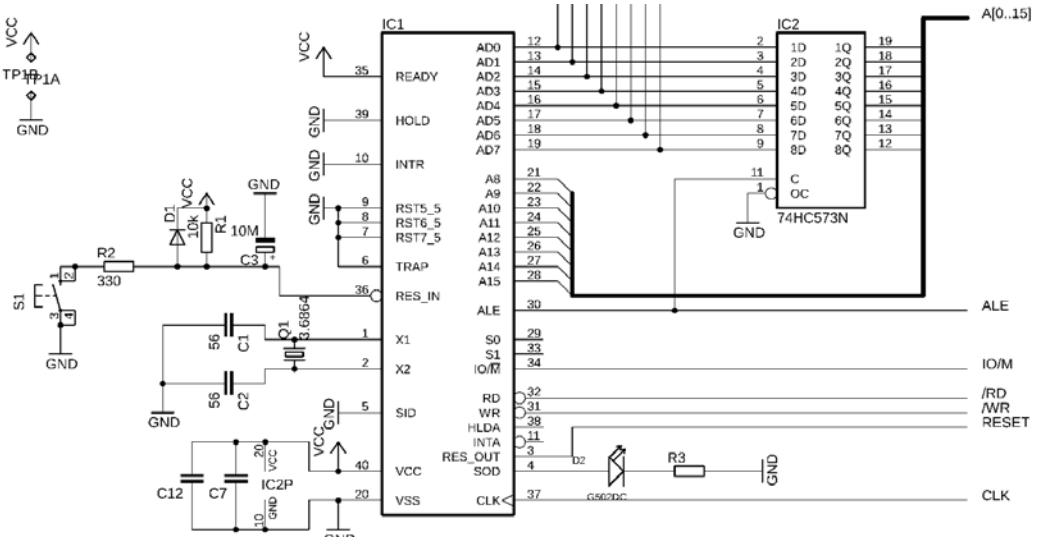
Třetí způsob je signál TRAP. Všechny předchozí způsoby přerušení může programátor zablokovat, „zamaskovat“ pomocí instrukce DI (Disable Interrupt) a povolit pomocí EI (Enable Interrupt). Pokud jsou přerušení zakázána (DI), procesor signály INTR a RST ignoruje. TRAP je výjimka, tento signál vyvolá přerušení i tehdy, když jsou zakázána. Proto se mu také říká *nemaskovatelné přerušení*.

Co se stane, když přijde naráz víc požadavků na přerušení? Procesor má zabudovaný jednoduchý mechanismus priorit, viz následující tabulka, a vykoná obslužnou rutinu pro požadavek s nejvyšší prioritou.

Signál	Priorita	Adresa, kde je uložena obslužná rutina	Vyvoláno pomocí…
TRAP	Nejvyšší (1)	0024h	Vzestupná hrana nebo log. 1 během testování
RST 7.5	2	003Ch	Vzestupná hrana (pozdržená)
RST 6.5	3	0034h	Log. 1 během testování
RST 5.5	4	002Ch	Log. 1 během testování
INTR	Nejnižší (5)	Podle poslané instrukce	Log. 1 během testování

https://8bt.cz/mag85

5.7 Zapojení centrální procesorové jednotky



Obrázek 24: Alpha CPU

Takto bude vypadat srdce celého počítače. Dovolte jen pár poznámek:

V roli adresového bufferu je obvod IC2 typu 74573 (ať už ve verzi HC, HCT nebo ALS). Funkčně je shodný s obvodem 74373, až na jeden rozdíl: 373 má vývody přeházené, 573 má hezky všechny vstupy na jedné straně pouzdra a výstupy na druhé. Hodinový vstup je připojen na vývod ALE, povolovací vstup OE jsem připojil natvrdo k zemi. Mohl bych ho připojit na vývod HLDA a zajistit tak odpojení adresní sběrnice ve stavu HOLD, ale tento stav nebude náš počítač používat.

Nebudeme používat ani HOLD, ani READY, ani přerušení, proto jsou tyto vstupy připojeny na neutrální úroveň (READY je 1 – stále připraveno, HOLD a přerušení na 0). Z výstupů jsem nepoužil stavové S0, S1, výstup RESET OUT, a logicky ani HLDA a INTA. Stejně tak nezapojené zůstalo i sériové rozhraní – vstup SID je připojen k zemi, výstup SOD je volný. Můžete si k němu připojit LEDku přes rezistor a zkoušet si třeba bliknout…

Vlevo od procesoru jsou dva obvody. Níž je generátor hodinových pulsů s krystalem a dvěma kondenzátory. Krystal jsem zvolil o frekvenci 3,6864 MHz. Proč? Proč ne 4 nebo, cojávím, 27?

Zaprvé – frekvence nesmí být nižší než 1 MHz. *Píšou to v datasheetu.* Zadruhé – neměla by být vyšší než 10 MHz. *To tam taky píšou.* Ideální je někde okolo 4 MHz, protože procesor pak běží na příjemně pomalé frekvenci okolo 2 MHz, tedy zhruba tak, jak běžela originální 8080A.

Proč ale 3,6864 MHz? No, protože pak bude procesor běžet na frekvenci 1,8432 MHz, tedy na poloviční.

Dobře, ale proč ne třeba 4 MHz?

No protože 1843200 Hz / 16 = 115200 a 1843200 Hz / 64 = 28800. Už svítá? Správně, 115200 a 28800 jsou standardní frekvence pro sériovou komunikaci. Z frekvence 1,8432 MHz je odvodíme bez problémů pomocí dělení celým číslem. Obvod ACIA 6850, který použijeme, umí dělit buď 16, nebo 64 (nebo také jedničkou, ale to nechme stranou). Kdybychom chtěli tyto frekvence získat třeba z 4 MHz, museli bychom hodiny dělit koeficientem např. 34,72 nebo 138,88, což moc dobře nejde, takže bychom museli volit nějaký blízký dělitel (asi 32 a 128) a v přenosu by byly chyby. Takže tady raději oželím trochu výkonu, když vím, že mi to usnadní další zapojování.

A poslední věc: kondenzátory u krystalu! Jsou důležité k tomu, aby krystal kmital na správné frekvenci. Jejich kapacitu můžete buď spočítat, nebo vyčíst z datasheetu. Já z datasheetu vyčetl 56 pF, což se mi tedy zdálo jako obrovská kapacita. Nasadil jsem 33 pF, a i to bylo moc. Nakonec jsem nechal jen jeden kondenzátor 33 pF u vstupu X2. Pak se ukázalo, že 56 pF bylo z datasheetu CMOS verze a já měl použitou NMOS, kde se doporučují při těchto frekvencích kapacity okolo 7 pF.

Nad generátorem hodin je obvod pro /RESET. Pomocí kondenzátoru k zemi a rezistoru k napájecímu napětí je po zapnutí napájení na vstupu /RESET IN logická 0, dokud se kondenzátor pomalu nenabije. Vhodný poměr odporu a kapacity si můžete buď spočítat, nebo experimentálně vyzkoušet. Myslím, že rezistorem 10 K a kondenzátorem 10 M nic nezkazíte. Dejte systému trochu času na start, protože po zapnutí napájení ještě chvíli napětí kolísá. Díky tomuto triku je v tu dobu procesor stále ve stavu RESET, a když nastartuje, je napětí už ustálené.

Ještě víc vlevo je tlačítko, které připojuje vstup na zem. Je to magické tlačítko RESET, které náš počítač přivede k rozumu, pokud se náhodou někdy zapomene a skončí třeba v nekonečné smyčce. Díky kondenzátoru jsou odfiltrované i zákmity…

Procesor tedy máme *pořešený*. S okolním světem komunikuje pomocí adresní sběrnice, datové sběrnice, signálů IO/M, /RD, /WR a hodinového signálu CLK. To je opravdu všechno, víc není pro jednoduchý počítač potřeba.

5.8 Zapojení pamětí RAM a ROM

Samotný procesorový obvod, jak jsme si ho navrhli v minulé kapitole, k ničemu není. Respektive – můžete ho zapojit, připojit datovou sběrnici přes rezistory (třeba 10k) na zem a sledovat, jak se na nejvyšším bitu adresy (A15) mění 1 a 0, protože procesor stále dokola čte celou paměť od 0000 do FFFFh, všude najde operační kód 00 (to jsou ty rezistory připojené k zemi), ten znamená „nedělej nic a načti další instrukci“, takže se obsah registru PC (Program Counter) zvýší o 1 a celý cyklus se opakuje. Když si k A15 připojíte LEDku přes rezistor, měla by bliknout cca 7x za sekundu. To znamená, že celý paměťový rozsah zvládne na dané frekvenci projít procesor za sekundu sedmkrát.

Právě jste stvořili úžasný 16bitový čítač. Ale to asi nebude to, co chceme. Musíme zajistit, aby procesor měl co vykonávat. Musíme mu dát paměť, kde bude program a data.

Procesor po spuštění nemá žádný mechanismus, jak třeba vzít program odněkud odjinud a nahrát ho do paměti. O to se musíte postarat sami. A nejčastější způsob je ten, že použijete paměť ROM (spíš EPROM či EEPROM / FLASH), v níž je uložený nějaký základní ovládací program.

Tím programem byl nejdřív takzvaný bootstrap, tedy krátký program (512 byte třeba), který dokázal načíst data třeba z pásky, ze čtečky karet, z pevného disku, zkrátka odněkud, nahrát je do paměti a spustit. S podobným postupem se můžeme setkat třeba u Arduina: zde je v procesoru uložený bootstrap loader, který čeká, jestli náhodou počítač neposílá nějaká (přesně daná) data. Pokud ano, přepne se do programovacího módu a nahraje přicházející data do paměti.

Později se z bootstrapu stal Monitor. Monitor byl už o něco větší program, který už komunikoval s obsluhou a uměl provést jednoduché příkazy, např. pro prohlížení paměti, její změnu, načtení programu z externích pamětí a jeho spuštění. Pokud pamatujete na počítač PMD-85 v první verzi, tak se po zapnutí ohlásil právě monitor. Vy jste pomocí MGLD mohli nahrát do paměti něco z magnetofonu, spustit to pomocí JUMP, nebo z externího rom packu stáhnout a spustit BASIC G

U některých počítačů, většinou domácích, už se monitor ani nepoužíval, už startoval rovnou vyšší jazyk (BASIC).

Profesionální počítače, určené třeba pro práci s CP/M, se zase pomalu vrátily k malým obslužným programům. Neříkalo se jim bootstrap, na to byly příliš velké, a ani Monitor, protože nekomunikovaly s obsluhou. Ujala se zkratka BIOS podle jedné ze základních částí CP/M (Basic Input Output System). BIOS je spíš kolekce rutin a programů, která ovládá hardware konkrétního počítače. Systém nemusí řešit, kde přesně a jak je připojen disk, to ví právě rutina v BIOSu. Po startu se spustí CP/M loader právě z BIOSu.

U jednodeskových počítačů se používal model „Monitor“. Takový monitor se vešel i do 1 kB ROM. K tomu nabídl jednodeskáč třeba 1 kB RAM, a bylo!

Chtěl jsem dodržet ducha doby, ale na rovinu přiznávám, že tehdejší paměti, třeba 2 kB EPROM a 1 kB SRAM, jsou už dnes dílem nesehnatelné, dílem sice sehnatelné, ale zato se s nimi mizerně pracuje… Opravdu vás nechci nutit shánět ultrafialovou mazačku EPROM!

Proto nepoužijeme EPROM, použijeme EEPROM, tu můžeme vymazat jednodušeji. I naprogramovat…

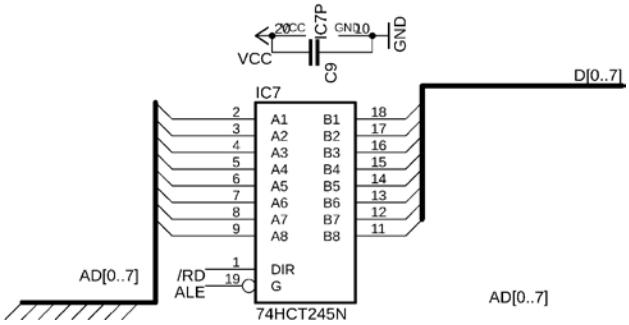
5.9 Budič sběrnice

Ale ještě předtím by bylo fajn udělat jednu úpravu. Totiž posílit a oddělit datovou sběrnici. Procesory obecně nemají příliš silné výstupní obvody, a když na datovou sběrnici připojíte víc obvodů, přetížíte ji. Se dvěma paměťovými obvody, obvodem ACIA a bufferem na adresu budeme už na hraně. Proto před tím, než budeme cokoli přidávat, připojíme posilovač.

Mimochodem: Pravidlo pro nízkou spotřebu a malé zatížení sběrnice zní: Používat CMOS kde to jde. Vyhněte se obvodům TTL (74, 74LS, 74ALS apod.) a hledejte CMOS verze (74HC, 74HCT, 74ACT). Totéž platí i pro periferní obvody – například dále probíraný obvod PIO 8255 lze sehnat i v CMOS verzi 82C55, která má nižší spotřebu a téměř nezatěžuje sběrnici. CMOS verze bývají většinou plně kompatibilní se svojí předlohou, občas bývají mírně vylepšené…

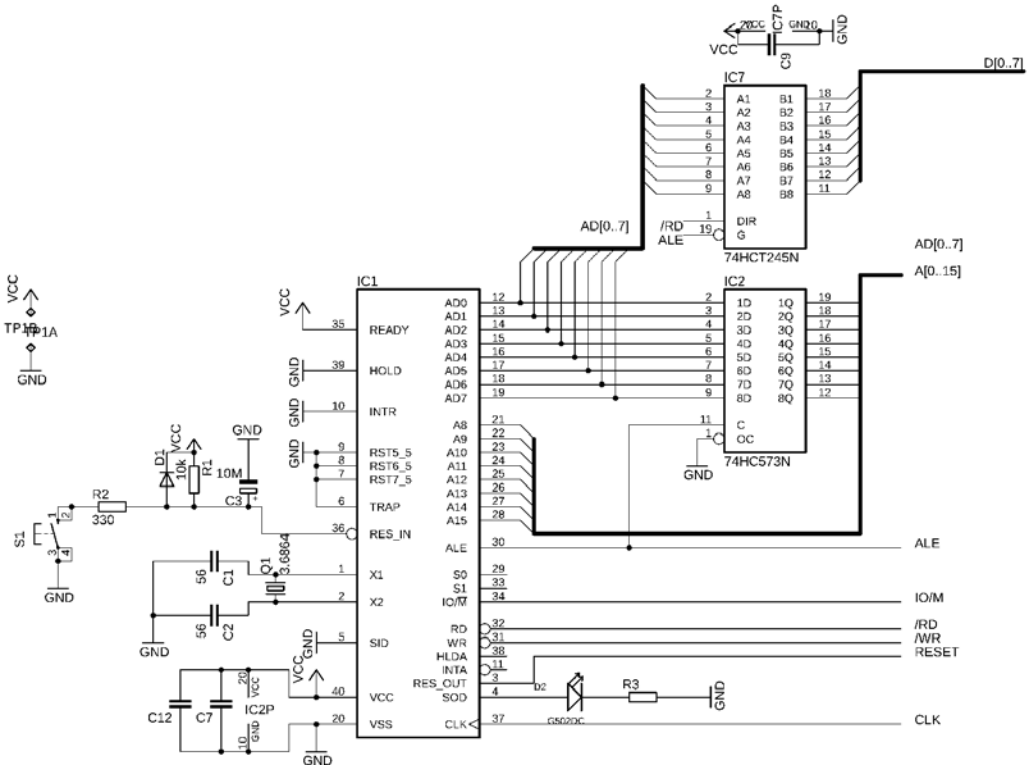
Použijeme obvod 74HCT245, což je osmice obousměrných bufferů a budičů. Vstupem DIR se určuje směr toku dat (0 = z B do A, 1= z A do B), vstupem G se přepíná obvod do stavu vysoké impedance (pokud je vstup G=1, je obvod odpojen a výstupy A i B ve stavu vysoké impedance).

Spojil jsem vstup G se signálem ALE, díky tomu je ve chvíli, kdy procesor na sběrnici posílá spodní část adresy, obvod odpojen a neruší. Takže kdyby nějaký periferní obvod chtěl posílat data ve chvíli, kdy procesor řeší latchování nižší části adresy (ALE=1), je datová sběrnice oddělená.



Obrázek 25: Alpha, budič sběrnice

Díky tomuto zapojení jsou k datové sběrnici připojeny maximálně dva obvody, a díky signálu ALE jsou zapojeny „proti sobě“, tedy vždy je aktivní jen jeden z nich. Celá „procesorová“ část vypadá tedy takto:



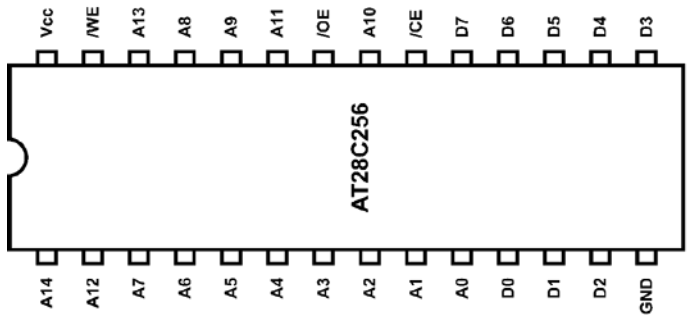
Obrázek 26: Alpha, procesor s budičem

5.10 (EEP)ROM

Já vím, není to nic úplně jednoduchého, co by se vám válelo doma, ale doporučuji buď koupit programátor pamětí EEPROM, nebo si ho postavit z Arduino MEGA. Není potřeba vymýšlet kolo, návody už existují a většinou stačí jen propojit vývody z MEGY s vývody paměti. Já to dělám zrovna tak, použiju nepájivé kontaktní pole, to propojím s Arduinem Mega, do pole pak zasouvám paměti a programuju jako po másle.

https://8bt.cz/eeprog

Použijeme EEPROM o velikosti 32 kB. Tyto paměti se vyrábějí pod označením 28C256 (podle výrobce např. AT28C256). Jsou dobře dostupné a mají i vhodné pouzdro DIP.



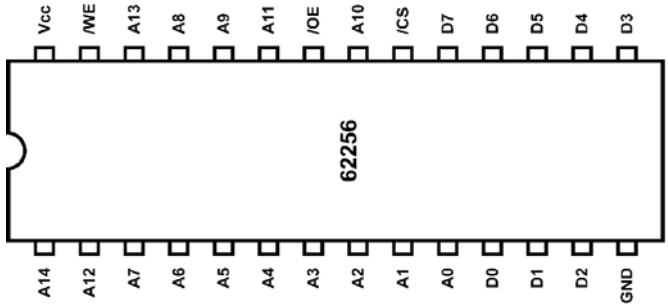
Obrázek 27: Zapojení 28C256

Datové vývody D0-D7 není třeba představovat. A0 až A14 je naše známá sběrnice s adresou (15 bitů = 32 kB) a tři signály slouží k ovládání paměti. /WE povoluje zápis (my ho v systému připojíme natvrdo k log. 1 a tím zápis zakážeme – paměť naprogramujeme v programátoru.) /CE (Chip Enable) říká paměti, že má reagovat na pokyny – pokud je 0, paměť reaguje, pokud 1, paměť se odpojí. A konečně /OE (Output Enable) povoluje čtení z paměti, tzn. že na výstupech IO0-IO7 je vystavena hodnota, uložená v paměti na adrese dané adresovými vstupy.

No jo, ale co když budeme potřebovat třeba jen 8 kB ROM? V takovém případě zkrátka připojíte vstupy A13 a A14 na log. 0 (zem), a v paměti budou tři čtvrtiny prostoru nevyužité. Takový způsob zapojení se někdy používá i záměrně – v jedné paměti máte např. čtyři alternativní firmwary, a buď pomocí mechanických přepínačů, nebo pomocí nějaké vnitřní logiky, si můžete volit, který firmware chcete použít. (I některé počítače to tak měly – vzpomínáte na ZX Spectrum 128 a jeho dva BASICy?)

5.11 RAM

Paměť RAM použijeme typu 62256. Její vývody jsou rozložené takto:



Letmým pohledem zjistíme, že jsou obě paměti zapojeny naprosto stejně! Až na určité rozdíly ve značení. Místo /CE (Chip Enable) je signál /CS (Chip Select), ale význam je stejný.

5.12 Paměťový subsystém a adresace

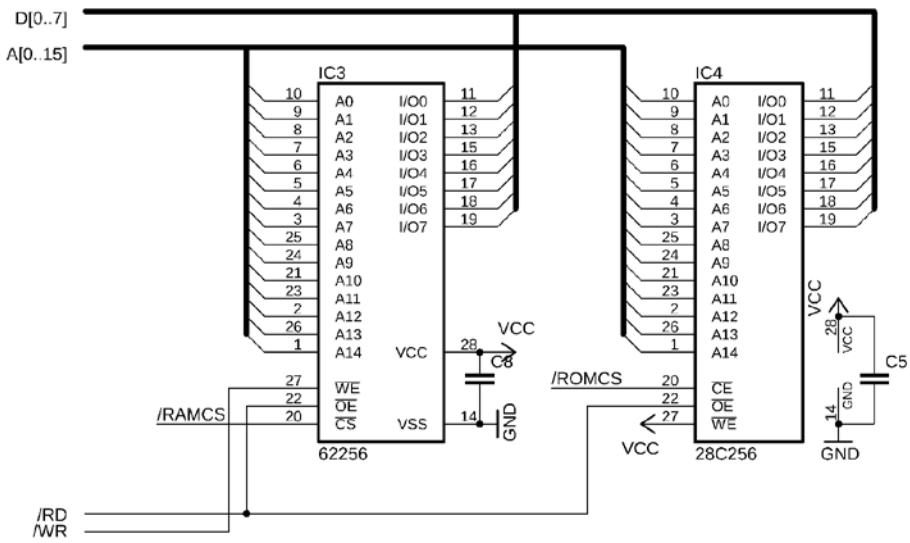
Máme tedy dva kusy paměti, každý s velikostí 32 kB, a jeden procesor s adresním prostorem 64 kB. Úplně se nabízí rozdělit dostupný prostor na dvě poloviny, jedna bude od 0 do 32767, druhá od 32768 do 65535, v jedné bude ROM, ve druhé RAM. Otázka pro bystré hlavy: Kde bude která paměť?

Odpověď se skrývá v předchozích odstavcích: procesor po RESETU začne provádět program od adresy 0000h, takže na téhle adrese by měla být ROM. Ergo ROM bude na adresách 0000h – 7FFFh, RAM na adresách 8000h – FFFFh.

A teď: jak budeme připojovat? Oba obvody připojíme naprosto stejně: datovou sběrnici na datovou sběrnici procesoru, adresovací signály A0 až A14 na odpovídající vývody procesoru. Vývod A15 z procesoru necháme zatím stranou, ten použijeme později.

Signál /OE (Output Enable) připojíme na signál /RD. Připomeňme si: Signálem /RD říká procesor, že bude číst. Vstup /OE u paměti slouží k tomu, že povolí výstup uložených informací na datovou sběrnici. To je vlastně přesně to, co potřebujeme: když chce procesor číst, pošle signál /RD, a ten „otevře“ výstup naší paměti.

Analogicky připojíme vstup /WE (Write Enable) u paměti RAM k signálu /WR u procesoru. Jen u RAM, u EEPROM ne. Asi nechceme, aby náhodná chyba v programu přepisovala obsah paměti EEPROM…



Obrázek 28: Alpha, zapojení pamětí

Poslední vstup, který zbývá, je Chip Select (/CS, u RAM značen jako /CE, ale funkce je stejná). To je „univerzální povolovací vstup". Pokud je neaktivní (log. 1), tak paměť ignoruje vše, co se na ostatních vývodech děje. Pokud je aktivní (log. 0), vykonává operace čtení a zápisu podle výše popsaných signálů.

Protože jsme spojili paralelně datové výstupy obou pamětí, bylo by fajn nějak zajistit, aby v jeden okamžik posílala svoje informace jen jedna paměť. Která? No to záleží na stavu vodiče A15.

Vodič A15 je v log. 0 tehdy, když procesor přistupuje na adresy 0000h – 7FFFH, v log. 1 pokud přistupuje na horní polovinu prostoru. Pokud je tedy A15 = 0, měla by být vybrána paměť EEPROM, pokud je A15 = 1, měla by být vybrána paměť RAM.

Ovšem ne vždy! Jen tehdy, když procesor přistupuje k paměti! Vzpomeňte si – k tomu slouží signál IO/M. Pokud procesor chce přistupovat k paměti, je tento signál v log. 0.

Tedy shrnuto:

Stav /CE	**…pokud je**
/CE u paměti EEPROM (/ROMCS) aktivní (0)	A15 = 0 a zároveň IO/M = 0
/CE u paměti RAM (/RAMCS) aktivní (0)	A15 = 1 a zároveň IO/M = 0

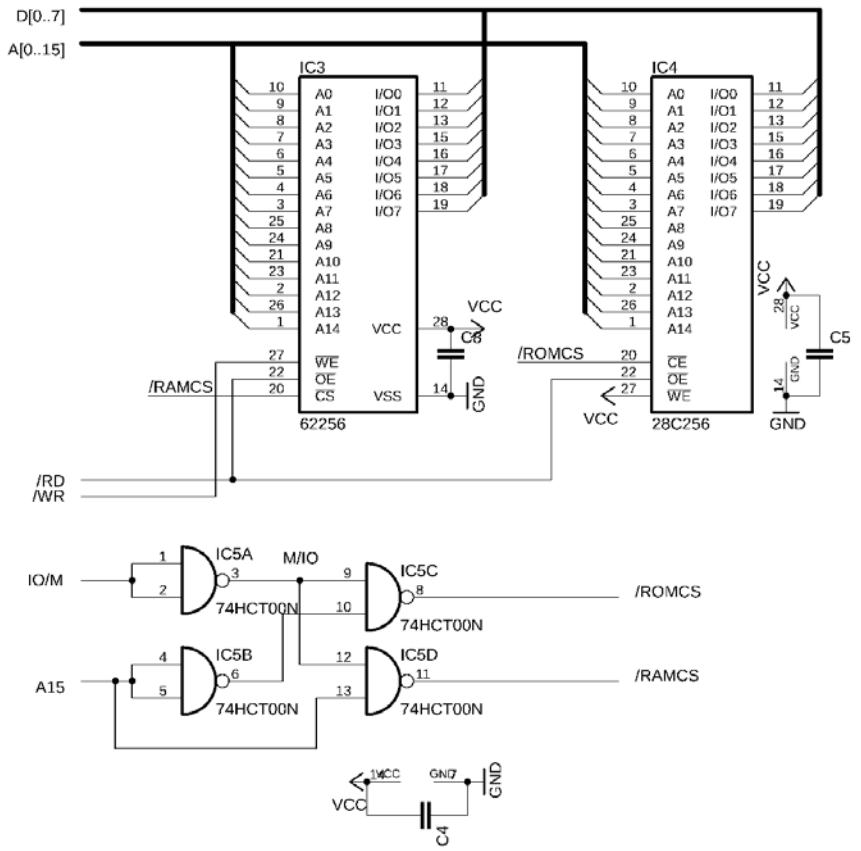
Pro paměť EEPROM jde tedy o logickou funkci A15 OR IO/M, pro paměť RAM to je (NOT A15) OR IO/M. Abychom zbytečně nepoužívali hradla OR a NOT, použijeme jedno pouzdro 74HCT00 – tedy 4 x NAND, a to takto:

/ROMCS = (NOT A15) NAND (NOT IO/M)

/RAMCS = A15 NAND (NOT IO/M)

Jedno hradlo bude invertovat IO/M, druhé bude invertovat A15, třetí vytvoří signál /ROMCS, čtvrté /RAMCS.

A máme to! Kompletní schéma je zde:



Obrázek 29: Alpha, paměti a dekodér

Můžete si zkusit vytvořit první program, nahrát si ho do EEPROM a nechat blikat LEDku na výstupu SOD.

Důležité upozornění: Pokud nevíte, jak na to, a slova „assembler“ a „stroják“ vám nic neříkají, nalistujte si prosím v druhé části knihy kapitolu „Programování v assembleru 8080“

5.13 Oživení zapojeného počítače

Můj kolega Marcel, programátor, občas říkal: „Mám to hotové, teď už to jen naprogramuju…”

Při stavbě Alphy jsme se dostali do stejného bodu: Máme ji hotovou, teď už ji jen postavit!

Dovolte mi pár poznámek a rad ke stavbě, které se budou hodit, ať už zvolíte cestu nepájivého pole, ovíjení, nebo plošného spoje (na webu naleznete odkaz, kde je možné stáhnout podklady pro jejich výrobu, a to jak ve formátu pro CAD Eagle, tak i ve formátu Gerber, s nímž umí pracovat všichni výrobci plošných spojů).

Sám jsem postavil několik prototypů tohoto počítače, udělal jsem spoustu chyb a dopustil se několika omylů. Teď se s vámi o ně podělím, abyste nemuseli prošlapávat už prošlapanou cestu.

Já jsem, poměrně sebevědomě, nechal vyrobit první plošný spoj. Bohužel stalo se přesně to, co zná každý autor: po nějakém čase vlastní chyby nevidíme. A tak jsem přehlédl, že u paměti jsou dva adresní vývody spojené (A6 jsem pojmenoval jako A5 a Eagle jej spojil s reálným A5) a že místo /RD zapojuju /WR. To se holt stane. Nebo že prohodím RAM a EEPROM. Ony jsou oba ty obvody stejné, jen u jednoho je zápis připojen na /WR a u druhého neaktivní. A ten první je vybrán přes /RAMCS, druhý přes /ROMCS. A pak je holt prohodíte. Nevidíte to v tom schématu a nevidíte, dokud si to celé neproměříte a neřeknete si: „*****!“

Když jsem všechny tyhle chyby napravil, nevhodné spoje proškrábl a chybějící připájel, tak jsem zjistil, že to už tak trochu funguje, ale vůbec ne tak, jak by mělo. Jako by to provedlo pár instrukcí, a pak se to zbláznilo… Pomocí analyzátoru jsem sledoval, co se děje na sběrnici, stále dokola, přemýšlel, uvažoval, hledal… Vypadalo to na chybu v EEPROM – ale ta byla naprogramovaná správně. Proměřoval jsem patici, jestli jsou všechny signály v pořádku – a byly! V zoufalství jsem analyzoval znovu a znovu, až jsem při jednom pokusu zapojil sondy na nožičky EEPROM místo na vývody adresového latche. Najednou se ukázalo, že A5 „visí ve vzduchu“. A při bližším zkoumání se podezření potvrdilo: na patici signál byl, na nožičce ne, protože příslušný pin v patici byl nějaký „vymletý“ a neměl kontakt s vývodem. Po lehkém přiohnutí začalo vše fungovat, jak mělo.

Ostatně i u druhého prototypu, u opraveného plošného spoje, jsem čelil podobné chybě. Ale nakonec se ukázalo, že při pájení patice do desky jsem vynechal jeden pin. Chybka, co mě stála dvacet minut hledání.

Ono věřit, že sestavíte vše podle schématu, zapnete a bude to fungovat, je docela odvážné. Většinou to tak není. Většinou uděláte i při pečlivé práci chybu, buď při návrhu, nebo při sestavování. Proto je dobré postupovat po částech a hned si ověřovat, jestli vše funguje.

Já začínám připájením patic pro integrované obvody, pak připájím napájecí konektor, piny a další „mechanické" součástky. Pak proměřím napájecí vedení – jestli je zem všude tam, kde podle schématu má být, jestli je napájecí napětí také všude tam, kde má být, a hlavně: jestli mezi zemí a napájecím napětím není zkrat. Na to mi stačí obyčejný multimetr se zvukovou signalizací.

V druhé fázi proměřuju sběrnice. Jednu sondu zabodnu do patice datového budiče v místě D0 a měřím: D0 u RAM, D0 u ROM, D0 u sériového obvodu, D0 u PPI. Posunu o jeden pin a pokračuju s D1 u všech obvodů. Pak si ještě překontroluju, jestli AD0 až AD7 vede od procesoru k latchi a k budiči. Pak proměřím A0 až A7 mezi latchem, paměťmi a dekodérem. Pak A8 až A15. Pak jen rychle proměřím ALE, /RD a /WR. A když toto všechno funguje, pokračuju dál.

Zapájím krystal a kondenzátor. Osadím procesor do patice a poprvé zapnu. Je mi jedno, co bude dělat, důležité je, jestli generuje správně hodinové signály. To si proměřím na výstupu CLOCK, nejlépe osciloskopem nebo logickým analyzátorem. Voltmetr ukáže na tomto výstupu něco kolem 2,5 V – to zhruba odpovídá střídavému signálu se střídou 1:1. Jakmile se ukáže, že vše kmitá, můžeme pokračovat dál. Ještě si pro jistotu zkontrolujte, jestli obvod pro RESET funguje správně (a zase pomůže logický analyzátor).

Můžete osadit adresní latch 574, budič 245, obvod NAND 7400, který generuje systémové signály, a můžete osadit i EEPROM. Když si do ní nahrajete jednoduchý „SOD BLINK" program bez použití podprogramů, tak i bez zapojené RAM dokážete otestovat, že vše funguje, že procesor správně čte instrukce a provádí je. (Testovací program si připravíme v další kapitole.)

V dalším kroku připojte RAM a sériový komunikační obvod (vysvětlíme si později). Díky tomu můžete už komunikovat s Alphou přes sériové rozhraní z PC. Poslední krok pak bude paralelní rozhraní a další periferie. Ale k nim se ještě dostaneme.

5.14 První program (pro pokročilé)

Když už víte, jak se takový procesor 8085 programuje, můžeme zkusit naši konstrukci oživit.

Programy budeme psát v assembleru 8085 – pokud nevíte, jak tento jazyk funguje, přeskočte na další část knihy a pak se sem zase můžete vrátit…

Nejjednodušší program bude obligátní „blikání LEDkou“. Po pravdě řečeno – zatím nemáme moc jiných způsobů, jak se přesvědčit, že systém funguje, kromě zmíněného připojení rezistorů k datové sběrnici, simulace instrukce NOP a sledování změn na adresové sběrnici. Ale máme LED na výstupu SOD!

Zapojte tedy obě paměti k procesoru a zkusíme si naprogramovat první program. Bude to nějak takto:

```
inicializace();
while(true) {
  SOD = 1;
  delay();
  SOD = 0;
  delay();
}
```

A máme to, rachota skončila, můžeme pokračovat…

Vlastně ne, teď to teprve začne. Musíme tohle všechno naprogramovat v assembleru. Začneme inicializací.

```
 .org 0  ; inicializace()
RESET: DI
   LXI SP, 0000h
```

Nejprve řekneme assembleru, že program bude začínat na adrese 0. U procesoru 8080 / 8085 to tak je, že nula je výchozí bod procesoru po spuštění. Bývá dobrým zvykem si tento bod nějak pojmenovat – třeba návěštím RESET.

První instrukce je DI. Ta zakáže přerušení. Náš systém sice přerušení nepoužívá, ale je rozumné na začátku práce, než jsou nastavené všechny potřebné věci, zakázat přerušení. Kdyby nějaké přišlo dřív, než je nastavený například ukazatel zásobníku, systém by zhavaroval.

Další instrukce nastaví ukazatel zásobníku na konec paměti RAM. Ta u Alphy zabírá paměťový prostor 8000h – FFFFh. Takže nastavíme SP na hodnotu „o jedna vyšší než poslední volná adresa“ – tedy FFFFh+1 = 0000h. Instrukce LXI SP poslouží výborně.

Po této inicializaci máme procesor připravený na práci. Nemusíme inicializovat nic víc – ostatně v našem systému nic víc není. Můžeme tedy napsat tu smyčku:

```
LOOP:
  MVI A, 11000000b
  SIM
  CALL DELAY
  MVI A, 01000000b
  SIM
  CALL DELAY
  JMP LOOP
```

Nejprve nastavujeme SOD na 1. K tomu slouží instrukce SIM. Připomeňme si: tato instrukce vezme hodnotu v registru A a podle ní nastaví SOD a masku přerušení (ta nás teď nezajímá). Pro nastavení musí být druhý nejvyšší bit (D6) roven 1. Pokud tomu tak je, procesor pošle hodnotu nejvyššího bitu (D7) na výstup SOD.

Následuje volání čekací smyčky. Tu zatím neřešíme, ta bude „někde, nějak, později". Prostě jen skočíme na adresu, kde bude čekací rutina, a ta se pomocí instrukce RET zase vrátí zpátky.

Následuje analogicky nastavení SOD na hodnotu 0, opět volání čekací rutiny, a po ní skok zase na začátek celého procesu.

Tím je jádro programu hotové. Jak na tu čekací smyčku?

K čekání se mohou použít různé způsoby. Například instrukce NOP – zabere 4 takty, což u našeho systému představuje zhruba 2,17 mikrosekund. Pokud jich použijeme tisíc, bude to 2,17 milisekund, pokud sto tisíc, bude to 217 milisekund, tedy skoro čtvrt sekundy. To by šlo použít.

Teda, šlo, kdybychom mohli do paměti uložit sto tisíc instrukcí. Nemůžeme. 32768 je maximum. Co s tím?

Opět použijeme smyčku. Prázdnou, uvnitř se nebude dít nic. Jen budeme odečítat od zadané hodnoty k nule, a až dojdeme k té nule, smyčka skončí. Nějak takto:

```
delay() {
   unsigned int d = KONSTANTA
   do {
     d--;
   } while (d!=0)
}
```

Použijeme dvojici registrů D, E. Do ní si uložíme časovací konstantu, a pak budeme jen ve smyčce snižovat hodnotu DE o 1 a kontrolovat, jestli je výsledek 0. Pokud ano, tak skončíme, pokud ne, točíme se znovu.

```
DELAY:
   LXI D, KONSTANTA
DLOOP:
   DCX D
   MOV A,D
   ORA E
   JNZ DLOOP
   RET
```

Smyčka DLOOP začíná snížením hodnoty ve dvojici registrů D a E – DCX D. Teď musíme zkontrolovat, jestli je výsledek 0. Bohužel (šestnáctibitová) instrukce DCX nenastaví příznak zero (to dělají jen instrukce osmibitové). Použije se opět trik, totiž logický součet (OR) hodnot v registrech D a E. Výsledek bude 0, pokud D i E budou nulové. A protože nemáme takto obecnou instrukci, musíme si nejprve hodnotu z jednoho registru zkopírovat do akumulátoru a pak udělat OR s druhým registrem. Instrukce ORA podle výsledku nastaví příznak Z a my tedy můžeme podle toho skákat – instrukcí JNZ, což je vlastně JUMP if NOT ZERO.

A teď otázka za sto bodů: Jak dlouho ta smyčka bude probíhat? Jasně, záleží to na té konstantě a bude to přímo úměrné. Ale kolik to bude? Dalo by se to spočítat?

Inu, dalo. Vezměte si k ruce tabulku s instrukcemi a jejich trváním (dám ji do přílohy, nebojte) a napište si, kolik která instrukce zabere taktů:

```
DELAY:
   LXI D, KONSTANTA ; 10 T
DLOOP:
   DCX D      ;  6 T
   MOV A,D    ;  4 T
   ORA E      ;  4 T
   JNZ DLOOP  ; 10 T / 7 T – 10 pokud se skáče, 7 pokud se neskáče
   RET        ; 10 T
```

Pokud bude konstanta = 1, proběhne tento cyklus přesně jednou. Zabere to tedy 10 + {6 + 4 + 4 + 7} + 10 = 41 T

Pokud bude konstanta = 2, proběhne tento cyklus takto:

10 + {6 + 4 + 4 + 10} + {6 + 4 + 4 + 7} + 10 = 65 T

Pokud bude konstanta = 3, poběží následujícím způsobem:

10 + {6 + 4 + 4 + 10} + {6 + 4 + 4 + 10} + {6 + 4 + 4 + 7} + 10 = 89 T

Je tedy vidět, že základní čekací doba je 41 T, a pokud zvýšíme konstantu o 1, zvýší to čekání o 24 T. Vzorec tedy je:

T = 41 + (N – 1) x 24

a z toho: N = (T – 41) / 24 + 1

Pokud bude konstanta rovna 100, bude čekání 41 + 99 x 24 = 2417 T.

Je to logické. První a poslední instrukce (LXI, RET) proběhnou vždy jen jednou (20T). Vnitřní smyčka jsou čtyři instrukce, a doba jejich provádění se liší podle toho, jestli už se dosáhlo nuly. Pokud ne, tak instrukce JNZ skáče a trvá 10T. Pokud ano, instrukce JNZ neskáče a trvá jen 7T. Průchod smyčkou tedy trvá 24T (6 + 4 + 4 + 10), pokud jde o poslední průchod, tak pouze 21T.

Vzorec můžeme samozřejmě zapsat jako T = 24 x N + 17

Co se stane, když bude konstanta rovna 0? No, hned první DCX D odečte jedničku od 0000h, operace přeteče a výsledek bude FFFFh. Smyčka se tedy vykoná nikoli nula krát, ale 65536krát!

Kolik vlastně taktů potřebujeme takto „propálit“, aby bylo zpoždění třeba půl sekundy? Víme, že procesor pracuje s taktem 1,8432 MHz, to znamená, že za sekundu vykoná 1 843 200 taktů. Na půlsekundové zpoždění tedy potřebujeme polovinu, ergo 921600 taktů. Zanedbáme takty, které zabere volání podprogramu instrukcí CALL, časování není zde úplně kritické.

Magická konstanta N bude tedy:

N = (921600 – 41) / 24 + 1 = 38399 (zaokrouhleně)

A co kdybychom chtěli blikat ještě pomaleji, třeba se sekundovými čekacími smyčkami? No, je to prosté, použijeme dvojnásobnou konstan… aha! Dvojnásobná hodnota je 76798, a to se nám do registrů nevejde. Tak zavoláme půlsekundové čekání dvakrát!

Anebo – co když do smyčky mezi DCX D a MOV naházíme několik NOPů? Třeba pět:

```
DELAY:      LXI     D, KONSTANTA ; 10 T
DLOOP:      DCX     D            ;  6 T
            NOP                  ;  4 T
            NOP                  ;  4 T
            NOP                  ;  4 T
            NOP                  ;  4 T
```

```
        NOP                 ;  4 T
        MOV     A,D         ;  4 T
        ORA     E           ;  4 T
        JNZ     DLOOP       ; 10 T / 7 T
        RET                 ; 10 T
```

Pět NOPů, každý po 4T, to máme 20 T. O tuto hodnotu se zvýší čas nutný k průchodu smyčkou. Vzorec se tedy změní na:

T = 61 + (N - 1) x 44

Půlsekundové zpoždění s touto rutinou bude vyžadovat konstantu 20945, sekundové pak 41890. Platíme za to tím, že program zabere v paměti víc místa.

Všimli jste si drobné nevýhody celé rutiny? Pracuje s registry D, E a A. Co když ale v A jsou nějaké informace, které chceme zachovat? Pak nezbývá než použít PUSH PSW a POP PSW, tedy uložit registr A a opět ho obnovit.

5.15 Překlad a spuštění

https://www.asm80.com/

Program máme napsaný, teď nastal čas ho přeložit. Spusťte si ASM80 a kliknutím na New file založte nový soubor. Do něj napište výše uvedené příkazy. Uložte jej pod názvem test.a80 – přípona .a80 je důležitá, podle ní překladač pozná, že jde o program pro procesor 8080/8085). Klikněte na Compile.

V seznamu souborů vlevo se objeví dva nové soubory: test.a80.lst a test.a80.hex. První je výpis programu včetně operačních kódů a umístění v paměti, takzvaný listing:

```
0000                        .ORG   0
0000                        ; inicializace()
0000   F3           RESET:  DI
0001   31 FF FF             LXI   SP,0000h
0004                LOOP:
0004   3E C0                MVI   A,11000000b
0006   30                   SIM
0007   CD 13 00              CALL   DELAY
000A   3E 40                MVI   A,01000000b
000C   30                   SIM
000D   CD 13 00              CALL   DELAY
0010   C3 04 00              JMP   LOOP
```

```
0013                WAIT:     EQU   38399
0013                DELAY:
0013   11 FF 95              LXI   D,WAIT
0016   1B           DLOOP:    DCX   D
0017   7A                     MOV   A,D
0018   B3                     ORA   E
0019   C2 16 00               JNZ   DLOOP
001C   C9                     RET
```

Druhý (HEX) je kód, určený k naprogramování paměti. Stáhněte si jej (pravý klik na název, Download) a použijte k nahrání do paměti EEPROM. Konkrétní postup se liší podle zvoleného programátoru.

Paměť zapojte zpátky do systému a zapněte napájení. Pokud jste neudělali při zapojení nikde chybu, měla by dioda po chvilce blikat s frekvencí 1 Hz. Úvodní prodleva je dána velikostí RC článku na vstupu RESET.

Funguje? Gratuluji, právě jste si postavili vlastníma rukama osmibitový počítač a naprogramovali ho!

Námět na cvičení: Tato úloha by šla přepsat tak, aby nevyžadovala paměť RAM. Konstrukci tak můžete oživit jen s třemi IO: procesor, adresový latch a paměť ROM. Stačí místo volání podprogramu (CALL) zkopírovat čekací rutinu na dané místo. Dvakrát.

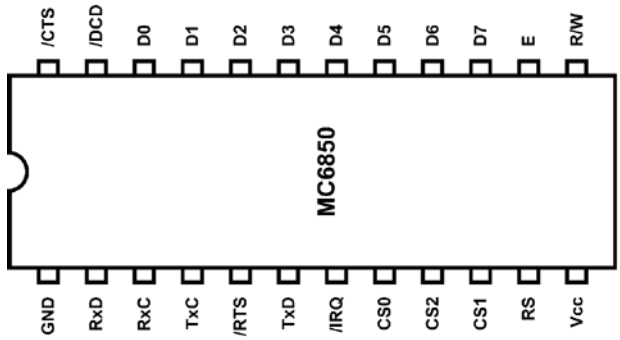
5.16 Sériová komunikace

Gratuluji ke zprovoznění počítače. Jen je trošku hloupé, že počítač vlastně umí dělat jen to, co mu vypálíte do EEPROM, a nemůžete s ním nijak komunikovat. To teď napravíme. Přidáme první vstupně – výstupní obvod, totiž sériové rozhraní.

Použijeme obvod 6850, přesněji verzi 68B50 (verze B umí pracovat i na frekvenci 2 MHz). Jedná se o obvod, označovaný ACIA – Asynchronous Communications Interface Adapter. Pod slovy „asynchronní komunikační rozhraní“ je trochu cudně skryto, že jde o standardní sériové rozhraní, jaké známe z „COM portů“ a dalších sériových portů na počítači. My k jeho výstupům připojíme nějaký převodník USB-to-Serial, a tím získáme možnost komunikovat s Alphou přes terminálový program z počítače.

Při testech se ukázalo, že i verze MC6850, tedy bez „B“, zvládne provoz touto rychlostí, ale může se to lišit kus od kusu…

ACIA 6850



Obvod 6850 je sériový komunikační obvod. Jeho hlavním úkolem je převést zaslaný bajt na sériový signál (tj. správně odvysílat start bit, datové bity, případně paritní bit, a nakonec stop bit) a opačně, tj. načíst správně časovaný sériový signál a připravit ho k předání procesoru.

Asynchronní v popisu znamená, že spolu s daty není přenášen hodinový signál ani není činnost přesně časována – když přijde bajt, je vyslán, když přijdou vstupní sériová data, jsou načtena. Synchronizace, tj. to, že bude načteno opravdu to, co načteno být má, zajišťují právě start a stop bity – podle jejich správného průběhu pozná obvod, že jsou data v pořádku.

Vysílání dat po výstupu TXDATA (Tx = transmit) a příjem na vstupu RXDATA (Rx = receive) je časován pomocí systémových hodin. U našeho počítače je systémový kmitočet roven 1,8432 MHz. Tento kmitočet je v ACIA vnitřně dělen (dělitel je programově nastavitelný na hodnoty 1, 16 a 64). Použijeme dělení 16, což znamená, že komunikační rychlost bude 1843200 / 16 = 115200 bitů za sekundu (baud).

Obvod 6850 s procesorem komunikuje pomocí osmibitové datové sběrnice (D0-D7), několika CS vstupů (CS = Chip Select), které určují, kdy se s obvodem komunikuje (CS0 = 1, CS1 = 1, /CS2 = 0 – ve všech ostatních případech je datová sběrnice odpojena), vstupu E (Enable, obvod komunikuje jen pokud je E=1), vstupu R/W, který udává, zda se z obvodu čte (1) nebo se do něj zapisuje (0) a vstupu RS, který udává, jestli se čtou/zapisují data (1), nebo řídicí hodnoty (0).

Kromě těchto signálů nabízí i signál /IRQ (Interrupt Request – požadavek na přerušení). Pomocí přerušení může ACIA dát vědět procesoru, že například přišla nějaká data po sériovém portu, nebo že data k odeslání byla odeslána apod. My přerušení nepoužijeme.

Z hlediska programátora se tedy obvod 6850 jeví jako dva registry (vybrané pomocí vstupu RS), z nichž jeden je datový, druhý „systémový“. Funkci osvětlí tabulka:

		VSTUPY	REGISTRY
RS	R/W	TYP REGISTRU	FUNKCE
0	0	Zápis	Řídicí registr (Control Register, CR)
0	1	Čtení	Stavový registr (Status Register, SR)
1	0	Zápis	Data k vyslání (Transmit Data Register, TDR)
1	1	Čtení	Přijatá data (Receive Data Register, RDR)

Všimněte si, že do řídicího registru je možné pouze zapisovat, nelze z něj číst, naopak stavový registr lze pouze číst, nelze do něj zapisovat. Není to totiž potřeba. Totéž s datovými registry – při čtení se čte to, co obvod přijal (a nezajímá nás to, co jsme odeslali). Při zápisu je jasné, že chceme data vyslat, nedávalo by smysl zapisovat do přijatých dat.

Řídicí registr CR

Osmibitový registr CR řídí čtyři funkce obvodu:

Bity 0 a 1 nastavují dělicí poměr hodin, viz následující tabulka.

CR1	CR0	Dělitel
0	**0**	1
0	**1**	16
1	**0**	64
1	**1**	RESET

Poslední kombinace nenastavuje dělitele, ale celý obvod resetuje do výchozího nastavení, tj. vyprázdní registry a nuluje příznaky. My použijeme kombinaci 01, tj. dělení 16. Kdyby bylo potřeba rychlost snížit, použijeme kombinaci 10, tj. dělení 64, a obvod bude komunikovat rychlostí 28800 Bd.

Bity 2, 3 a 4 nastavují délku vysílaných a přijímaných dat (sedmibitové nebo osmibitové), zda se pracuje s paritou a kolik je stop bitů. Tyto informace najdete např. i v nastavení Hyperterminálu, když půjdete hledat detaily připojení. Pro naše účely použijeme kombinaci 101, tj. osmibitový přenos, bez parity, 1 stop bit.

Bity 5 a 6 určují, jestli vysílač po odvysílaném bajtu bude žádat o přerušení a v jakém stavu bude výstup RTS

Bit 7 určuje, jestli přijímač bude vyvolávat přerušení v případě chyby.

Pro bližší popis odkazuju zájemce opět k datasheetu, my použijeme hodnotu 15h (0 00 101 01 binárně), tj. osmibitový přenos, bez parity, přerušení zakázané a přenosová rychlost 115200 Baud (Baud je jednotka „bitů za sekundu" – včetně start a stop bitů).

Stavový registr SR

Svět není dokonalý, život není fér a asynchronní sériové přenosy nejsou nijak sladěné s biorytmy našeho procesoru. Pokud pošlu bajt do 6850, začne ho obvod vysílat. Ovšem předtím je dobré podívat se, jestli už dokončil vysílání toho předchozího. Při příjmu je zase dobré se podívat, jestli nějaký bajt už načetl, a pokud ho načetl, tak ho zpracovat, aby se uvolnilo místo pro další bajt. Popřípadě získat informaci o tom, jestli nedošlo k chybě. Tyhle informace jako když najdete ve stavovém registru SR.

Pokud načtete bajt ze SR, tak vás jednotlivé bity informují o následujícím:

Bit 0 – Receiver Data Register Full (RDRF). Pokud je nastaven na 1, znamená to, že obvod načetl bajt po sériové lince do přijímače a bylo by záhodno ho zpracovat. Jakmile procesor přečte stav datového registru, je bit RDRF nastaven na 0. Nula znamená, že žádný nový bajt nepřišel.

Bit 1 – Transmitter Data Register Empty (TDRE). Jakmile zapíšete do datového registru bajt, nastaví se TDRE na 0 a obvod začne bajt vysílat po sériové lince. Jakmile ho vyšle, nastaví tento bit na 1. Programátor by si měl před tím, než nějaký bajt pošle, zkontrolovat, že může – tedy že bit TDRE = 1.

Bity 2, 3 pracují s řídicími signály CTS, DCD, a já je tady s klidným svědomím opomenu.

Bit 4 – Framing Error (FE) znamená, že přijatá data byla špatně časována, např. že nepřišel požadovaný STOP bit.

Bit 5 – Receiver Overrun (OVRN). Overrun, neboli hezky česky *přeběh*, je stav, kdy přijímač přijal bajt, ale procesor ještě nezpracoval předchozí přijatý. Ten nově přijatý je tedy zahozený (protože jej není kam dát, žádný vnitřní buffer není) a nastaví se OVRN na 1, aby bylo jasné, že došlo k chybě.

Bit 6 – Parity Error (PE). Pokud využíváme přenos s paritou, zkontroluje 6850 paritní bit. Pokud byl chybný, nastaví příznak PE.

Bit 7 – Interrupt Request (IRQ) říká, že obvod požádal o přerušení z nějakého závažného důvodu. Buď byl přijat bajt a je povolené přerušení při přijetí dat, nebo byl odeslán bajt a je povoleno přerušení při odeslání, nebo vypadla nosná (DCD).

6850 a OMEN ALPHA

Jak připojit tento obvod k naší konstrukci? Tak, je jasné, že datová sběrnice přijde na datovou sběrnici procesoru. Co se vstupy CS, R/W a RS?

Vstup E povoluje obvodu komunikovat s procesorem (v logické 1). My budeme chtít tento obvod připojit jako periferii, tedy vstupně-výstupní obvod. Vývod IO/M shodou okolností říká právě to, jestli procesor chce pracovat s pamětí (0), nebo s periferií (1). Můžeme ho tedy přímo připojit na vstup E, tím se zajistí, že obvod ACIA bude reagovat jen na požadavky pro zápis a čtení do / z periferií a bude ignorovat operace s pamětí.

R/W určuje, jestli se bude zapisovat (0) nebo číst (1). V našem systému má podobnou funkci signál /WR – je 0, pokud procesor hodlá zapisovat, jinak je 1, takže může sloužit jako signál čtení (i když není /RD aktivní). Je to v pořádku, stačí to?

Podívejme se na to: Pokud bude procesor chtít zapisovat, pošle správný signál, ve všech ostatních případech bude číst. Nevadí to něčemu? Nevadí, protože bude odpojený od sběrnice díky vstupům CS a E.

Jenže moment: Podívejte se pořádně na průběhy signálů – signál IO/M je nastaven dřív, než proběhne cyklus ALE, a /WR je přitom neaktivní (=1). Takže ACIA začne číst a posílat hodnoty po datové sběrnici, takže by teoreticky mohl ovlivnit latchování adresy. Teď oceníme oddělovač sběrnice ze začátku této kapitoly.

Odpověď tedy zní: Nestačí to! Měli bychom signál E držet neaktivní, pokud ALE=1, protože obvod nemá oddělené povolovací vstupy pro zápis a čtení (jako mají třeba paměti). Popřípadě použít /CS2 pro ALE. Ale díky oddělovači to není třeba řešit.

Pracovat se s 6850 bude tedy pomocí instrukcí IN a OUT. Ty používají osmibitovou adresu, takže se teď ještě musíme postarat nějak o to, aby obvod správně z adresy poznal, že procesor komunikuje s ním. Navrhuju, pokud proti tomu nic nemáte, použít vstupy CS0, CS1 a /CS2 a připojit je na adresní vodiče A7, A6 a A5 takto:

A7	A6	A5	A4	A3	A2	A1	A0
CS0	CS1	/CS2	-	-	-	-	RS

Obvod bude vybraný pouze tehdy, pokud budou nejvyšší tři bity adresy rovny 110. Na stavu ostatních bitů nebude záležet. Jakákoli adresa, která splní masku „110x xxxx“ vyhoví. Vyhovují tedy adresy C0h – DFh.

Na nejnižší bit A0 jsem připojil vstup RS. Ten vybírá mezi datovým (1) a řídicím / stavovým registrem (0). Adresy, které budou mít v nejnižším bitu 1, budou přistupovat k datovému registru (C1h, C3h, C5h, … DDh, DFh). Ostatní adresy budou přistupovat k řídicímu či stavovému registru (C0h, C2h, C4h, … DCh, DEh). Programátor si může vybrat kteroukoli z nich, jsou ekvivalentní.

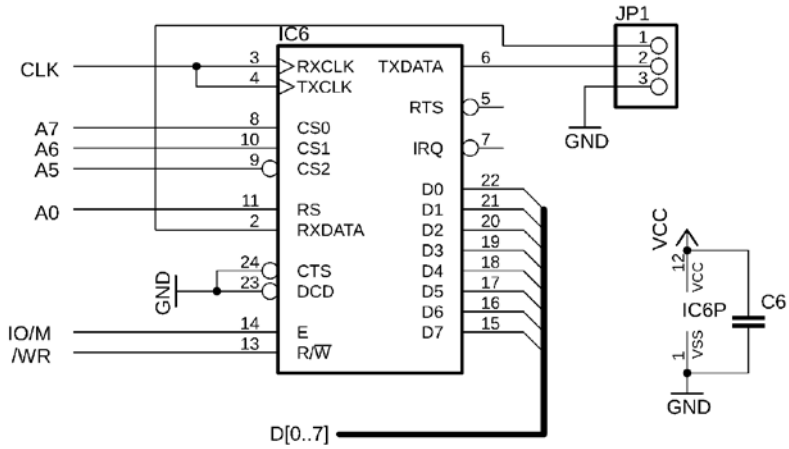
Přesto doporučuju použít ty adresy, které mají v bitech A1-A4 logické 1. Do budoucna to ušetří případné mrzení s připojováním dalších periferií.

Proto si zapište – ACIA v počítači OMEN Alpha je zapojena takto:

Adresa	Zápis	Čtení
0DEh	Řídicí registr	Stavový registr
0DFh	Data k odeslání	Přijatá data

Musíme také přivést hodinový signál na vstupy RXCLK a TXCLK. Oba spojíme, protože chceme vysílat stejnou frekvencí jako přijímat. Připojíme je na vývod CLK z procesoru. Signál /RTS budeme ignorovat, vstupy /CTS a /DCD, sloužící k řízení přenosu, připojíme na zem.

Vývody RXDATA (vstup) a TXDATA (výstup) jsou už to sériové rozhraní… připojte je k převodníku USB-to-UART, samo sebou kříženě (TxD na vstup RXDATA, RxD na výstup TXDATA). Počítač je připraven komunikovat, stačí ho jen naprogramovat!



Obrázek 30: Alpha, sériové rozhraní

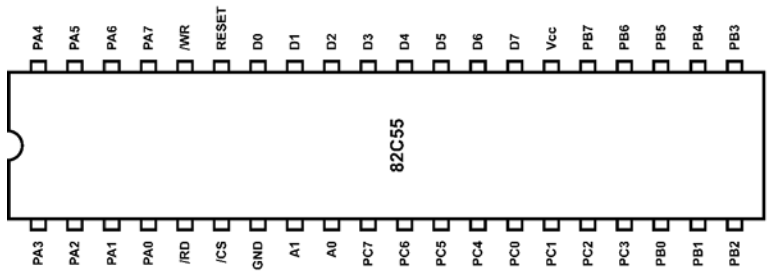
5.17 Další rozšíření

Omen Alpha je hezký jednodeskový počítač, použitelný se sériovým rozhraním. Ovšem něco tomu chybí…

Už vím! Chybí tomu takový ten *feeling* jednodeskového stroje. Chybí tomu tlačítková klávesnice a displej ze sedmisegmentovek. Můžeme je připojit?

Pro připojení dalších periferií můžeme použít třeba podobný způsob jako u adresového latche – tedy osminásobné buffery, třístavové budiče apod. Já zvolil o něco mocnější způsob, totiž komplexní obvod pro řízení paralelních portů, známý jako PIO (Parallel Input/Output) 8255 (někdy se značí i jako PPI – Programmable Parallel Interface). Jde o obvod z rodiny podpůrných obvodů Intel 82xx, což jsou podpůrné obvody pro procesory z řady 80xx (vzpomeňte na zmiňované obvody 8228, 8224 nebo 8251).

5.18 PPI 8255

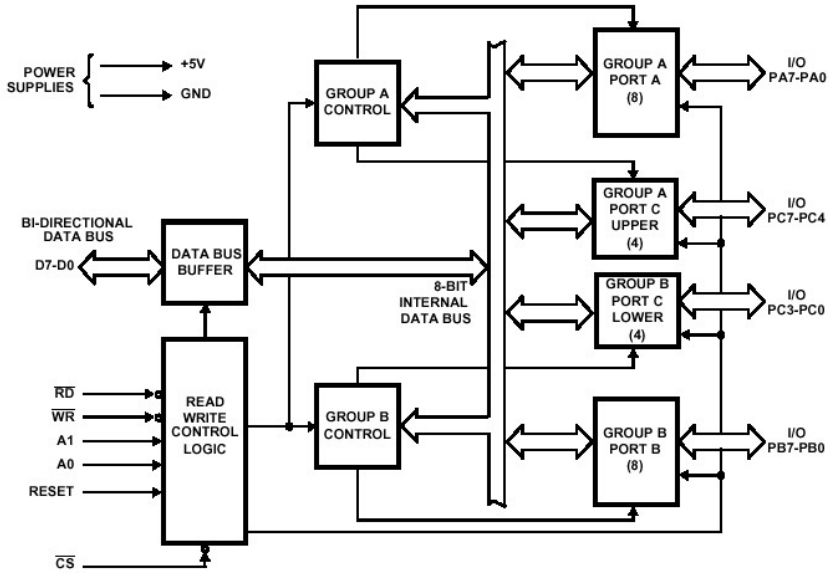


Obvod 8255 je notoricky známý, protože se používal v nejrůznějších počítačích i domácích konstrukcích paralelních portů. Počítač PMD-85 používal několik takových obvodů (pro řízení portů, pro přístup k periferiím i pro přístup k paměťovému modulu), v počítači PMI-80 řídil tento obvod displej a klávesnici, v počítači Didaktik Gama se přes tento obvod řešilo stránkování paměti i připojování joysticku, v nejrůznějších konstrukcích modulů a portů pro domácí počítače byl tento obvod… Jeho výhodou tehdy bylo, že byl snadno dostupný i v ČSSR.

Naštěstí je dobře dostupný dodnes, a co je ještě lepší: je dostupný ve verzi CMOS (82C55). Původní verze totiž měla neskutečně velkou spotřebu, až 120 mA. CMOS verze má spotřebu desetinovou a menší.

Pro srovnání: procesor 8080A v původní (NMOS) verzi odebíral v průměru 60 mA napětí +5 V, 40 mA z napětí +12 V, procesor 8085A požadoval v maximech 135 mA, CMOS verze 80C85

odebírá 10-20 mA. Není divu, že československé počítače s procesorem 8080A a dvěma obvody 8255 potřebovaly zdroje dimenzované na jednotky ampérů, když jen procesor a periferie spotřebovaly (a protopily) dobrého půl ampéru.



Obrázek 31: Struktura PPI 8255

Obvod 8255 nabízí tři osmibitové vstupně-výstupní paralelní porty, značené jako A, B a C (či častěji jako PA, PB, PC). K procesoru se připojuje pomocí datové sběrnice, dvou vstupů /RD a /WR, které říkají, jestli se z obvodu čte nebo do něj zapisuje, a pomocí dvou adresních vstupů A0 a A1. Ty říkají, jestli se přistupuje k některému z portů nebo k řídicímu registru. Nechybí samozřejmě vstup /CS.

A1	A0	/RD	/WR	/CS	Funkce
0	0	0	1	0	Čtení PA
0	1	0	1	0	Čtení PB
1	0	0	1	0	Čtení PC
1	1	0	1	0	Čtení řídicího registru
0	0	1	0	0	Zápis do PA

A1	A0	/RD	/WR	/CS	Funkce
0	1	1	0	0	Zápis do PB
1	0	1	0	0	Zápis do PC
1	1	1	0	0	Zápis do řídicího registru
X	X	X	X	1	Datová sběrnice odpojena
X	X	1	1	0	Datová sběrnice odpojena

Poslední řídicí vstup je RESET. Logická 1 na tomto vstupu vynuluje řídicí registr a nastaví všechny porty jako vstupní.

Připojení k našemu systému je tedy jasné – datová sběrnice na datovou sběrnici, RESET na RESET, /WR a /RD jsou taky jasné, A0 a A1 připojíme k adresním vodičům A0, A1. Porty připojíme k nějakým vhodným konektorům. Zbývá vyřešit jen jeden zásadní problém, totiž co se vstupem /CS.

Na rozdíl od obvodu sériového rozhraní 6850 má obvod 8255 jen jeden vstup Chip Select, a my přitom potřebujeme, aby obvod reagoval na kombinaci hned několika signálů:

Signál IO/M je ve stavu 1

Bity A7-A5 v adrese jsou jiné než 110 (tuto kombinaci už má sériové rozhraní).

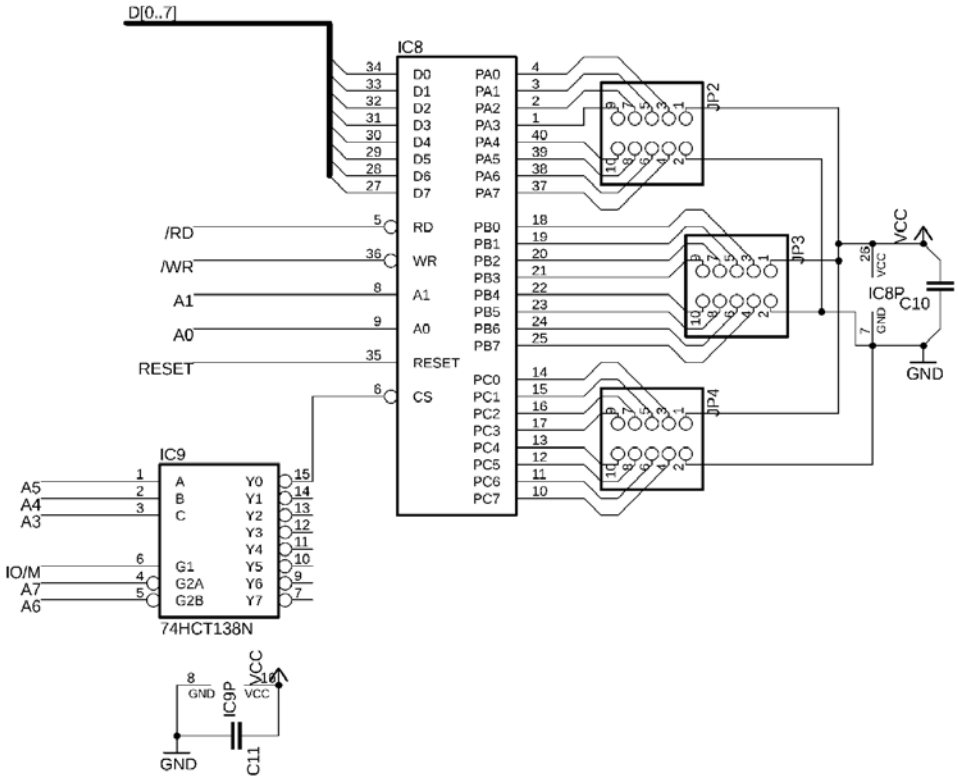
Můžeme říct, že CS bude vybrán tehdy, když IO/M=1 a zároveň A5=1. Tím se obvod objeví na adresách, které mají tvar XX1X XXXX. Naštěstí nám k tomu stačí jedno jediné hradlo NAND. Naneštěstí žádné volné nemáme. Nakonec jsem si řekl, že použiju obvod 74138, tedy dekodér 1-z-8. Tento obvod má tři vstupy pro data a tři povolovací vstupy (jeden neinvertující, dva invertující). Připojil jsem jej tak, že povolovací vstupy jsou připojené na signály IO/M, A7 a A6 – IO/M na neinvertující vstup, adresní signály na invertující. Obvod je tedy vybraný tehdy, když procesor komunikuje s periferií na adresách 00XX XXXX. Hexadecimálně 00h – 3Fh.

Datové vstupy A, B, C jsem připojil na adresní signály A5, A4 a A3. Jednotlivé výstupy /Y0 – /Y7 jsou tedy adresované takto:

A5 (A)	A4 (B)	A3 (C)	Aktivní vývod	Adresu
0	0	0	/Y0	00h – 07h
0	0	1	/Y4	08h – 0Fh
0	1	0	/Y2	10h – 17h

A5 (A)	A4 (B)	A3 (C)	Aktivní vývod	Adresu
0	1	1	/Y6	18h – 1Fh
1	0	0	/Y1	20h – 27h
1	0	1	/Y5	28h – 2Fh
1	1	0	/Y3	30h – 37h
1	1	1	/Y7	38h – 3Fh

Pro první obvod 8255 jsem použil výstup /Y0. Obvod se tak objevuje na adresách 0000 0XXX (binárně). Případný další periferní obvod lze adresovat dalšími výstupy. Teoreticky můžeme k některému vývodu Y připojit i sériový obvod z předchozích kapitol. Ovšem můžeme to samozřejmě nechat tak, jak to je.



Obrázek 32: Alpha, paralelní porty

Obvod 8255 se tedy objevuje na osmi adresách:

Registr 8255	Adresy
Port A	00h, 04h
Port B	01h, 05h
Port C	02h, 06h
Řídicí registr	03h, 07h

Opět použijeme ty vyšší, kde jsou bity X rovny 1, tedy 04 – 07h.

Programování 8255

Před prvním použitím obvodu 8255 musí procesor říct, jak se mají které porty chovat – které mají být vstupní, které výstupní, popřípadě jestli mají mít ještě nějakou funkci navíc. Slouží k tomu řídicí slovo, které je potřeba nejprve nahrát do řídicího registru. Má následující formát:

Bity	Funkce
7	Vždy 1
6,5	Režim brány A (00 = základní, 01 = jednosměrný strobovaný, 1x = obousměrný strobovaný)
4	Směr brány A (0 = výstup, 1 = vstup)
3	Směr horní poloviny brány C – bity C7 až C4 (0 = výstup, 1 = vstup)
2	Režim brány B (0 = základní, 1 = strobovaný)
1	Směr brány B (0 = výstup, 1 = vstup)
0	Směr dolní poloviny brány C – bity C3 až C0 (0 = výstup, 1 = vstup)

Pokud některou z bran nepoužijete, je lepší nechat ji jako vstup (případný zkrat výstupů pak nezpůsobí problémy). Řídicí slovo pro nastavení všech bran jako vstupů je binárně 1001 1011, neboli hexadecimálně 9B.

Brány A a B se dají nastavit jako vstup / výstup pouze jako celek, ne po jednotlivých bitech. Bránu C lze nastavit nezávisle pro horní či dolní polovinu. Základní režim funguje tak, jak by člověk očekával – co pošleme na výstupní bránu, objeví se na výstupech a obráceně.

Máme-li brány nastavené na vstup a do základního režimu, pak hodnoty na nich čteme z příslušných registrů. Pokud jsou nastavené na výstup, zapisujeme požadované hodnoty rovněž do registrů bran. U Alphy to bude tedy tak, že zapíšeme hodnotu 9Bh na port 07 (=řídicí slovo) a pak hodnoty na branách A, B, C čteme z portů na adresách 4, 5 a 6.

Pomocí řídicího slova 10001011 (8Bh) nastavíme port A jako výstupní. Když pak začneme posílat hodnoty 00h a FFh na port 4, bude se měnit úroveň na všech výstupních vodičích brány A.

Druhý typ řídicího slova umožňuje nezávisle nastavovat a nulovat jednotlivé bity portu C. Má následující formát:

Bity	Funkce
7	Vždy 0
6, 5, 4	Ignorují se
3, 2, 1	Číslo nastavovaného / nulovaného bitu
0	Hodnota, která se zapíše do daného bitu

Pomocí řídicího slova 0E (binárně 0000 1110) se nastaví sedmý bit brány C (C7) na hodnotu 0. Řídicím slovem 0Fh se nastaví tentýž bit na hodnotu 1. Což se hodí třeba pro simulaci sběrnice SPI nebo pro nějaké řízení toku dat.

Obvod 8255 umožňuje použít porty A a B v takzvaném „strobovacím režimu". V praxi to znamená, že se k osmibitovému portu přidá část bitů z portu C, která slouží jako signály STROBE, ACK a INTR („Data připravena", „Data přijata" a „požadavek na přerušení"). U obousměrného režimu (pouze port A) jsou tyto signály zdvojeny (pro směr „dovnitř" a pro směr „ven"). Těmito režimy se ale nebudu zabývat, v případě potřeby jejich popis najdete v literatuře nebo datasheetech.

Já se rozhodl nasadit 8255 do Alphy proto, abychom mohli snadno připojovat další periferie. Například displej, klávesnici nebo SD kartu (teoreticky jde přes tento obvod připojit i Compact Flash karta). Vše si ukážeme v následujících kapitolách.

5.19 Displej ze sedmisegmentovek

Pokud bychom chtěli udržet řádného retro ducha, tak nasadíme sedmisegmentový displej. Stačí šest sedmisegmentovek s budiči a dva výstupní porty obvodu 8255. Ale upřímně: Je s tím spousta zapojování a pájení…

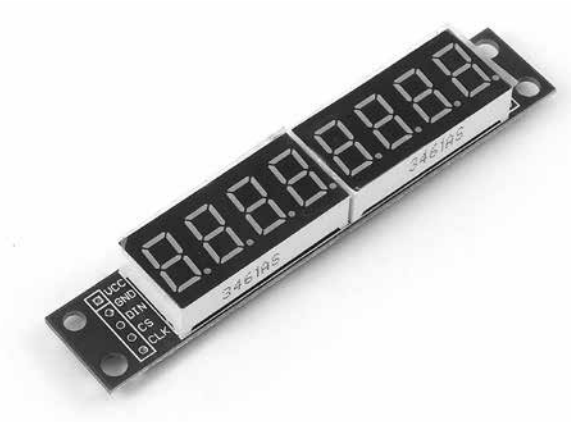
Lepší by bylo využít hotových modulů. Jsou k dispozici jako samostatné destičky s osmi sedmisegmentovkami a ovládají se jednoduchým sériovým rozhraním pomocí dvou či tří vodičů – jeden hodinový, druhý datový, případně vybavovací. K jejich ovládání nám postačí dva či tři bity, a já bych doporučil použít port C z obvodu 8255, protože u portu C lze velmi jednoduše přepínat hodnoty na jednotlivých pinech. Ale i na jiných portech to bude fungovat, jen musíme bity měnit

složitěji – buď posílat všech 8 bitů najednou, což může ovlivnit případné další periferie na témže portu, nebo načítat aktuální hodnotu, XORovat ji a zase posílat zpět.

Standardem u podobných modulů bývá obvod MAX7219. Existuje pro něj spousta knihoven, je dobře dokumentovaný a jeho ovládání je jednoduché.

Obvod MAX7219 dokáže ovládat až 64 segmentů / LED. Používá se pro ovládání maticových displejů 8x8 nebo pro ovládání až osmi sedmisegmentovek (sedmisegmentovka má samozřejmě osm segmentů, to je jak s těmi třemi mušketýry… Osmý segment je desetinná tečka).

Data se přenášejí po šestnácti bitech od nejvyššího. Na datový vstup pošlete nejvyšší bit a vzestupná hrana hodinového vstupu jej zaznamená do interního posuvného registru. Takto pošlete 16 bitů a vzestupnou hranou na vstupu CS (to je ten vybavovací, někdy se značí LOAD) se přenese celé řídicí slovo do obvodu.



Takových modulů můžete za sebe zapojit víc, protože obvod MAX7219 má kromě sériového vstupu i sériový výstup, takže když máte dva takové moduly za sebou, pošlete zkrátka dvakrát 16 bitů, ony se propíšou přes první obvod i do druhého a signálem CS pak najednou provedou.

Formát šestnáctibitového řídicího slova je následující:

Bity	Obsah
15-12	Ignorováno
11-8	Adresa registru
7-0	Osmibitová data

Na MAX7219 se tedy můžeme dívat jako na obvod s šestnácti osmibitovými registry. Jejich funkce je:

Registr	Funkce
0	Žádná
1	Pozice 0
2	Pozice 1
3	Pozice 2
4	Pozice 3
5	Pozice 4
6	Pozice 5
7	Pozice 6
8	Pozice 7
9	Dekódovací režim (pro každou pozici, 0=přímé ovládání segmentů, 1=BCD)
10	Intenzita svitu (0-15)
11	Scan (počet ovládaných pozic – 1)
12	Shutdown (0=vypnuto, 1=zapnuto)
15	Display test (0 = normální operace, 1=vše rozsvítit)

Pro přímé ovládání zvolte dekódovací režim 0, intenzitu svitu klidně 10, Scan limit 7 (všechny pozice), shutdown = 1 a test = 0. Bez dekódování ovládají bity 7 až 0 přímo segmenty v sedmisegmentovce.

7	6	5	4	3	2	1	0
DP	A	B	C	D	E	F	G

5.20 LCD displeje 16x2, 20x4

Podobný postup můžeme zvolit i pro známé znakové LCD displeje 16x2 nebo 20x4 (označované též jako 1602 a 2004).

Tyto displeje mají osmibitovou datovou sběrnici (D0-D7), signál Chip Select (zde E jako Enable), signál Read/Write a signál Register Select. Nepřipomíná vám to něco? Správně: je to téměř

identické s řídicími signály u periferních obvodů PPI, ACIA apod. Pomocí signálu RS určujete, jestli se zapisuje do datových nebo řídicích registrů, E aktivuje periferii a R/W říká, jestli se bude zapisovat, nebo číst.

Pokud připojíme takový displej přímo na datovou sběrnici, RS připojíme na A0, R/W třeba na /WR a E na nějaký vhodný volný výstup adresního dekodéru 74138, můžeme k displeji přistupovat jako k dalšímu zařízení.

Druhá možnost je využít toho, že tyto displeje umí pracovat i ve čtyřbitovém režimu. Pak potřebujeme celkem 4 datové + 3 řídicí signály, a to v pohodě obsloužíme jedním jediným portem obvodu 8255. Pokud oželíme čtení a nebudeme potřebovat přepínat data na vstup a výstup, tak můžeme použít klidně port A nebo port B ve výstupním módu.

5.21 Klávesnice 5x4

Každý správný jednodeskový počítač měl klávesnici, co vypadala jako (a někdy i doopravdy byla) z kalkulačky. 16 tlačítek pro hexadecimální znaky, a k tomu několik „funkčních" kláves, kterými se vyvolávaly základní funkce ovládacího programu.

Počítač KIM-1 měl tlačítek 24 (7 funkčních + jeden přepínač pro krokování), počítač PMI-80 jich měl 25 (hlavní důvod byl ten, že stejnou klávesnici používala i nějaká kalkulačka od stejného výrobce), počítač Heathkit ET-3400 si vystačil s šestnácti (a RESETem). U šestnáctitlačítkového ET-3400 byl použitý takový fígl, podobný zadávání příkazů u ZX Spectra: po RESETu či ukončení příkazů měly tlačítka 0-9 a A-F význam příkazu (B = Back, F=Forward, D = Do apod.)

Ať už bylo tlačítek kolik chtělo, platilo snad bez výjimky, že byly uspořádané do matice M řádků x N sloupců. Procesor postupně vybíral jeden řádek po druhém a četl hodnoty na vodičích sloupců. Ty byly zvedány pull-up rezistory k log. 1, takže v klidu jste přečetli samé jedničky.

Nejjednodušší by tedy bylo připojit řádkové vodiče k výstupnímu portu 8255, sloupcové vodiče ke vstupnímu portu, v klidu nechat na výstupech samé 1 a testovat řádek po řádku posláním log. 0.

Pokud bychom udělali klávesnici 5x4, tak bychom potřebovali pět řádkových vodičů a čtyři sloupcové. To je docela nešťastné, protože si tím zaplácneme přinejmenším jeden a půl portu. Na druhou stranu se nabízí opět stejný fígl jako u periferií, totiž vybavit klávesnici obvodem 74138 – dekodérem 1-z-8. Pro řádek teď budeme potřebovat tři vodiče, pro sloupce 4, takže celou klávesnici obsloužíme jediným portem C (spodní půlku necháme na čtení sloupců, horní polovinu uděláme jako výstup, kterým budeme vybírat řádek).

Stejný princip používaly i další počítače, jen s větší maticí. Sinclairy měly matici 5x8, ovšem fyzicky uspořádanou jako 10 tlačítek ve 4 řadách. Commodory měly matici 8x8, ale fyzicky uspořáda-

nou úplně jinak. S našim řešením bychom mohli ovládat až 8x4 = 32 kláves. Na plné QWERTY to je málo, ale kdybychom použili větší dekodér (1-z-16, 74HCT154), tak získáme 16x4=64 kláves. A s tím už se dá nadělat víc parády – a přitom všechno jen přes jeden port.

Samozřejmě je vždy otázka, co je lepší přístup a jestli zvolit o jeden obvod navíc, což jsou nějaké náklady a zvýšení složitosti, nebo jestli obětovat dva porty.

5.22 Systémový konektor

Pokud jste měli nějaký osmibitový počítač, pravděpodobně pamatujete na to, že měl hned několik konektorů, a ten největší a nejzajímavější se jmenoval „systémový". U Sinclairů to byl hranový konektor, respektive jen část plošného spoje, u jiných měl podobu třeba konektorů FRB nebo patice pro zasouvání rozšiřujících modulů…

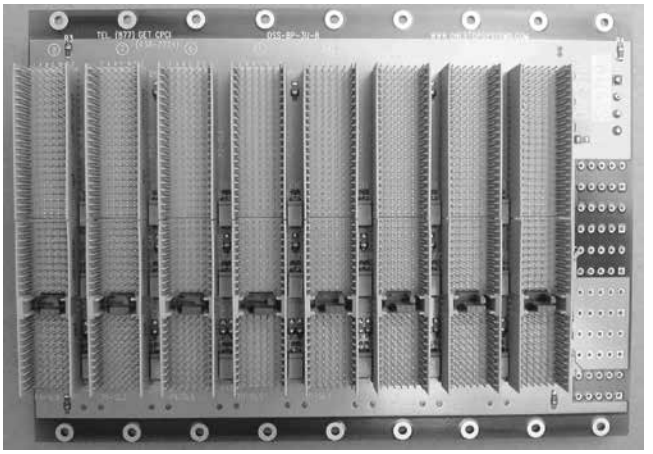
Bez ohledu na mechanické zpracování měly tyto konektory společné některé základní rysy: Připojovaly se přes ně další periferie (různá rozhraní u Spectra, paměť u ZX-81, cartridge s programy nebo periferiemi u Atari, …), a proto obsahovaly obousměrnou datovou sběrnici, adresní sběrnici (alespoň část), řídicí signály /RD, /WR, napájení a někdy i další signály (hodiny, RESET, přerušení atd.) U ZX Spectra byly na systémovém konektoru vyvedené všechny signály mikroprocesoru Z80, u PMD-85 pouze datová sběrnice, část adresní a některé systémové signály.

Díky takové sběrnici se snadno rozšíří systém o nové periferní obvody. Základní deska tak může zůstat malá, kompaktní, nemusíte kvůli nově vymyšlené periferii celý počítač předělávat, a navíc – pokud dodržíte formát sběrnice, můžete kdykoli změnit samotný počítač, a periferie budou beze změny fungovat dál.

Jen je potřeba dávat pozor na to, že podobný systém nelze nafukovat donekonečna. Že brzy narazí na elektrické limity – rychlé změny signálů na dlouhých vedeních postihne útlum a rušení, napájecí vodiče neutáhnou velký odběr… V takovém případě je lepší změnit koncepci a místo „počítače s moduly" navrhnout systém stylem „backplane".

„Backplane" počítače byly v osmdesátých letech oblíbené především v průmyslovém a poloprofesionálním nasazení. Tvořila je skříň, která měla na dně desku se sběrnicí a konektory. Ty měly navzájem spojené odpovídající vývody a do nich se zapojovaly jednotlivé desky.

Někdy byly pozice zaměnitelné, některé systémy měly ale například levou pozici vyhrazenou pro procesorovou desku a mezi touto pozicí a zbytkem pozic byly zapojené budiče sběrnice. Na této zadní desce (proto „backplane") byl často i napájecí zdroj. Při vypnutém systému pak šlo zapojit další rozšiřující moduly či stávající odpojit.



Obrázek 33: Backplane

Samozřejmě to nebylo bez problémů, někdy se spolu dvě desky „popraly“ o adresy či o signály a nešlo je dohromady použít, někdy se praly „jen tak“ a řešení bylo lehce magické, třeba jako „funguje to, pokud je deska A zasunutá vlevo od desky B“. Tedy přesně to, co zdravý rozum vyloučí jako první: „To na to přeci nemůže mít vliv!”

Nedá mi to a na tomto místě opět připomenu legendární počítač IQ-151. On nebyl nějak významně špatný, chybně navržený nebo tak něco. On byl koncepčně dobrý, ale zlikvidovalo ho provedení. Jednak úplně strašlivá klávesnice, tuhá a nespolehlivá, a jednak poddimenzované diody ve zdroji, které v prvních sériích nedokázaly dodat dostatek proudu pro víc modulů, takže jste zapojili BASIC a VIDEO, počítač fungoval, ale další periferii jste už nezapojili. A pokud ano, stalo se, že se ze zdroje zakouřilo a diody vlivem velkého proudu odešly. Ale v pozdějších sériích, s posíleným zdrojem a vylepšenou konstrukcí, už šlo o zajímavý systém. Bohužel pro něj přišel příliš pozdě.

I u Alphy jsem navrhl podobný konektor. Vyvedl jsem datovou sběrnici, tři adresní signály, sedm signálů z dekodéru 138 (na osmý je připojen obvod 8255 PPI), signály /RD, /WR, hodiny, RESET, přerušovací vstup a napájení. Díky tomu mohu snadno připojit třeba rozhraní pro CF kartu, o němž se ještě pobavíme, nebo další periferní obvody, pro něž není vhodné paralelní rozhraní.

Terminologická vsuvka: Takovému konektoru by se mělo říkat *aplikační*, nikoli *systémový* (nemá všechny systémové signály).

5.23 Programové vybavení

Neustálé vytahování a zasouvání pamětí EEPROM a jejich přeprogramovávání nesvědčí ani elektronice, ani konstruktérově psychice. Pojďme si tedy napsat nějaký základní ovládací program, který umožní snazší nahrávání.

Takovým základním programům se dnes říká například firmware, ale v duchu osmibitových dob ho nazvěme Monitor.

S velkým písmenem, prosím, abychom jej odlišili od zobrazovacího zařízení…

Monitor můžete znát možná z počítačů PMD-85 nebo IQ-151. Monitor se nazýval i obslužný program jednodeskových počítačů. Monitor poskytuje několik jednoduchých příkazů, pomocí nichž se počítač ovládá. Příkazy obvykle zahrnují změnu obsahu paměti, někdy i práci s registry a ladicí příkazy, a také pokyn pro spuštění uživatelských programů (respektive skok na zadanou adresu).

Příkazy se zadávaly buď z klávesnice, nebo přes sériové rozhraní. Obvykle sestávají z jednoho znaku, např. M je pro změnu paměti (Memory), G pro skok na zadanou adresu (Go to…) apod.

U Monitorů, které umí pracovat i se sériovým rozhraním, se často objevoval příkaz pro načtení HEX souboru. Důvtipný trik je, že Monitor implementuje příkaz ":" – což je počáteční znak u každého řádku HEX souboru. Pak načte jeden bajt délky, dva bajty adresy, jeden bajt typu záznamu a pak čte zadaný počet bajtů a ukládá je do paměti.

Takže vlastně stačí takový program napsat, nahrát do EEPROM, a zasunout zpátky do Alphy. Tím máte na dlouhou dobu po starostech – programy překládáte tak, aby fungovaly od adresy 8000h, a do Alphy je nahrajete tím, že soubor HEX pošlete přes sériové rozhraní.

Monitor zabere s trochou dobré vůle jeden kilobyte. Když budete hodně plýtvat, tak dva. Co s těmi zbývajícími třiceti? Já si třeba připravil interpret jazyka BASIC, přeložil jsem ho od adresy 1000h a nahrál taky do EEPROM. Stačí mi zadat příkaz G1000 a spustí se BASIC.

Zdrojové kódy mého Monitoru a Monitoru MON85 najdete na GitHubu:
https://github.com/osmibity/alpha.

Ale BASIC není jediný jazyk, který můžeme s Alphou použít – můžeme použít de facto cokoli, k čemu jsou dostupné zdrojové kódy pro 8080 / 8085. Nabízí se třeba FORTH…

Monitor Alpha MONv3

V době, kdy píšu tuto knihu, je aktuální Monitor na GitHubu MONv3. Tento Monitor komunikuje s obsluhou po sériovém portu (115200 Bd, 8 bitů, bez parity, 1 stop bit). Příkazy jsou jednoznakové:

- M (Memory) – změna / prohlížení obsahu paměti
- G (Go) – spuštění uživatelského programu
- D (Dump) – výpis obsahu paměti
- R (Registers) – výpis obsahu registrů
- X (Breakpoint) – nastavení bodu přerušení
- C (Continue) – pokračování po přerušení.
- : (Colon) – začátek dat ve formátu Intel HEX (vysvětlíme si později)

Příkazy M, G, D a X požadují zadání adresy. Adresa se zadává hexadecimálně, tedy pomocí znaků 0-9, a-f a A-F. Jakmile zadáte jiný znak, je vkládání adresy ukončeno. Monitor bere jen poslední čtyři znaky jako platné. Proto můžete překlep ignorovat a zadat adresu znovu a správně. Ze sekvence „112312331234“ si Monitor vezme jen poslední znaky: 1234.

U příkazu M zadáte adresu a Monitor ukáže obsah na této adrese. Nyní můžete buď zadat nová data (opět pomocí hexadecimálních znaků, platné jsou poslední dva) nebo ponechat původní. Stiskem Enter se posunete na další adresu, stiskem Backspace na předcházející a mezerou vkládání ukončíte.

Příkaz G spustí program od zadané adresy. Po ukončení běhu by měl program vrátit řízení zpět na adresu 0.

Příkaz D vypisuje paměť po větších blocích než příkaz M. Opět se pomocí Enter posouváte k dalším adresám, pomocí Backspace k nižším a mezerníkem ukončujete.

Dvojtečka slouží k nahrávání větších programů ve formátu HEX. Tento formát si představíme za chvíli. Je to jednoduchý datový formát, který rozděluje data do krátkých bloků, nejčastěji po šestnácti bajtech. Každý blok začíná znakem dvojtečka, pak následují servisní údaje (adresa, počet bajtů, typ záznamu), pak samotná data, kontrolní součet a znak konce řádku. Většina sériových terminálů umí poslat text po sériové lince, takže stačí pouze poslat HEX soubor přes terminál, a Monitor jej správně zpracuje a uloží do paměti.

Pomocí příkazu X nastavíte bod přerušení (breakpoint) na určitou adresu. Monitor si na tuto adresu uloží speciální instrukci. Jakmile na ni procesor při provádění programu narazí, zavolá se opět Monitor, který oznámí, že program je zastaven a vrátí na danou adresu původní obsah. Teď si můžete pomocí příkazů D a M prohlédnout obsah paměti, popřípadě ho změnit. Pomocí příkazu R si můžete prohlédnout nebo změnit obsah registrů. Když chcete pokračovat, zadáte příkaz C.

6 Základy programování v assembleru 8080

6 Základy programování v assembleru 8080

Ve světě mikroprocesorů nejsou žádné příkazy známé z vyšších programovacích jazyků. PRINT tu nenajdete. Jediné, co procesor umí, je vzít z paměti číslo – operační (strojový) kód instrukce – a podle něj provést nějakou činnost, třeba přesunout hodnotu odněkud někam nebo sečíst dvě čísla. Jediné, čemu mikroprocesor rozumí, je tedy *číslo*. A protože hovoříme o osmibitových mikroprocesorech, je přirozené, že ta nejčastěji používaná čísla budou právě osmibitová, tedy v rozsahu 0 až 255, hexadecimálně 00h až FFh.

Platí tedy. že každý osmibitový procesor zná přesně 256 instrukcí? Ani náhodou! Některé jich znají mnohem míň (třeba procesor 8080), takže některé operační kódy nemají žádnou přiřazenou instrukci. Jiné naopak mají mnohem víc instrukcí (Z80) a řeší to tím, že některé instrukce mají operační kód vícebajtový.

Program se skládá z instrukcí, které procesor vykonává po řadě tak, jak jdou po sobě v paměti, s výjimkou skoků. Program je zapsaný třeba takto:

21h 55h 3Ah

Tyto tři bajty jsou procesorem 8080 interpretovány tak, že do registrů H a L uloží hodnoty 3Ah a 55h. U procesoru Z80 mají tato tři čísla stejný efekt (protože Z80 je s 8080 na úrovni strojového kódu kompatibilní). U procesoru 6502 mají zcela jiný význam – vezme se obsah registru X, k němu se přičte číslo 55h, výsledek se ořízne na osm bitů a dostaneme adresu. Z paměti na téhle adrese se vezme jeden bajt, z paměti na adrese o jedničku vyšší druhý bajt. Tyto dva bajty dají dohromady 16bitovou adresu v paměti, kam procesor sáhne pro číslo, a nakonec provede operaci AND mezi hodnotou v registru A a tímto číslem (výsledek uloží do registru A). No a pak následuje prázdná instrukce (3Ah).

Různé procesory mají tedy odlišné vnímání stejných kódů, což je důvod, proč nelze vzít program pro jeden z nich a spustit bez úprav na druhém.

Jazyk symbolických adres (JSA) slouží k tomu, aby si programátor nemusel pamatovat tyhle číselné kódy (což o to, většinou si je stejně pamatují), a aby u složitějších programů nemusel počítat adresy, tj. na jaké adrese v paměti je která instrukce. Proto zavádí jednak symbolická návěští (takže se zapisuje nikoli JUMP 1234h, ale třeba JUMP START), a jednak zavádí symbolická jména pro instrukce. Takže třeba u procesoru 8080 nemusí programátor zapisovat zmíněný skok jako kód C3h, ale symbolicky: „JMP“ (jako že *jump*, rozumíme si…)

Assembler je nástroj, který překládá program, zapsaný v těchto symbolických jménech, do konkrétních kódů jednotlivých instrukcí.

Co to znamená, když se řekne „Program je napsaný v assembleru“? Ve skutečnosti to, že program je napsaný v jazyku symbolických adres. Zkrátka se stalo, že se tomuhle jazyku začalo říkat „assembler“, i když správně je assembler vlastně ten překladač.

A když někdo řekne, že napsal program „ve strojáku“? Pravděpodobně ho napsal v tom symbolickém jazyce a přeložil assemblerem do strojového kódu. „Stroják“ je takové označení, které se taky přeneslo na jazyk symbolických adres. *Ale mohlo se stát, že je dotyčný hardcore programátor a opravdu z hlavy nadiktoval instrukční kódy…*

6.1 Assembler ASM80

Dovolte mi, abych si na tomto místě přihřál trošku své vlastní polívčičky.

Totiž: pro to, abyste mohli napsat a přeložit program v assembleru, potřebujete dvě věci. Zaprvé: textový editor, a zadruhé: překladač.

Překladačů existuje velké množství. Pro každý typ procesoru hned několik. Navzájem se od sebe v některých detailech liší, především co do pohodlí zápisu nudného kódu – někde musíte vše vypisovat, jiné assemblery mají makra, co vám pomůžou. Každý si tak zkrátka může najít ten svůj oblíbený assembler.

Já jsem už před časem přemýšlel, že bych udělal univerzální assembler pro osmibitové procesory. Abych mohl překládat programy pro nejznámější stroje, a nemusel přitom řešit, že assembler pro jeden procesor funguje jen ve Windows 95, druhý zase pod Linuxem, třetí potřebuje Javu, každý má jinou syntax ovládání…

Proto jsem napsal webové IDE (Integrované vývojářské prostředí), které v době psaní této knihy umí překládat programy pro osm různých procesorů. Mám rád webová řešení – nemusíte se starat o aktualizace, nemusíte nic instalovat…

V IDE máte k dispozici pracovní prostor (workspace) – obdoba „projektů“ z jiných IDE. Pracovní prostor sídlí ve vašem prohlížeči. Pokud chcete pracovat se svými pracovními prostory z různých počítačů, přihlaste se (můžete k přihlášení využít účet u Google, GitHubu, Facebooku či Twitteru) a vaše pracovní prostory se budou synchronizovat všude tam, kde budete přihlášeni.

V assembleru tvoří program většinou jeden nebo několik málo souborů. Jejich přípona se liší podle požadovaného typu procesoru – pro procesor 8080 to je .A80, pro procesor Z80 to je .Z80, pro procesor 6502 to je .A65 apod.

Díky tomu, že se jedná o online nástroj, není, jak už jsem psal, nutné kamkoli cokoli instalovat. Jen píšete a překládáte.

Pro větší pohodlí vývojářů je v IDE zabudovaný i debugger, kde můžete programy pomocí zabudovaného emulátoru testovat, krokovat apod. Navíc je k dispozici několik simulátorů reálných počítačů (PMD-85, PMI-80, SAPI-1, ZX Spectrum apod.)

Všechny programy, které budou uvedené v této knize a doprovodných materiálech, používají právě syntax pro překladač ASM80. Neměl by být problém je upravit pro váš oblíbený překladač, pokud dáváte nějakému jinému přednost.

Příkazový řádek?

Vím, že online nástroje nejsou mezi programátory příliš oblíbené. Proto jsem vzal jádro překladače a udělal z něj nástroj pro příkazovou řádku. K běhu používá prostřední Node.js. Stačí toto prostředí nainstalovat (existuje pro nejrůznější OS) a pomocí nástroje npm (je součástí Node.js) nainstalovat překladač:

```
npm i -g asm80
```

Po instalaci můžete překladač volat tak jako ostatní: pomocí *asm80 [parametry] jméno_souboru*

Formát HEX

Výstupem assembleru jsou soubory ve formátu .HEX. Jedná se o formát, standardizovaný Intelem. V zásadě jde o binární data, ovšem v textové podobě, vždy s informací o tom, kam se mají v paměti umístit. Například takto (kód je uložen od adresy 0100h):

```
:10010000ED52898EDD8E05CE21ED7AFD29DD09DDEA
:10011000850010FDDDE9D7CD3602CD1101ED5EE39E
:0301200008FDE3F4
:00000001FF
```

Každý řádek začíná znakem dvojtečka a končí znakem „konec řádku“.

První dvě číslice za dvojtečkou říkají, kolik je na řádku dat. Zde to je 16 (10 hexadecimálně). Třetí řádek je kratší, obsahuje pouze 3 bajty dat, a poslední řádek s nulami je speciální ukončovací řádek.

Další čtyři číslice zapisují adresu, kam se mají data uložit. Na prvním řádku je to 0100h, na druhém 0110h, na třetím 0120h.

Dvě číslice po adrese udávají typ řádku. 00 jsou obyčejná data, 01 je *terminátor*, ukončovací řádek.

Pak následuje domluvený počet bajtů, každý vždy jako dvojice znaků. Poslední je kontrolní součet.

Procesory od Motoroly používají podobný formát, nazývaný Motorola S format.

Většina programátorů pamětí umí s daty ve formátu HEX pracovat. Stejně tak i některé systémy nebo zabudované monitory u některých jednočipových počítačů umí přímo tento typ souborů číst a ukládat do paměti. Dalo by se říct, že se jedná do jisté míry o standard ve světě programovatelné elektroniky. Ostatně i výstupem překladačů pro jednočipy AVR, PIC apod. je právě soubor s daty v HEX formátu.

S tímto formátem umí většinou pracovat i obslužné programy, Monitory. Pak stačí poslat HEX soubor po sériové lince a Monitor jej uloží do paměti.

6.2 Základy assembleru

Ano, já vím, terminologicky to není správné označení, ale *mí čtenáři mu rozumí*. Půjde o Jazyk symbolických adres, všeobecně (a nesprávně) nazývaný *assembler*.

Procesor, jak jsme si už řekli, zpracovává instrukce tak, že čte z paměti jejich strojový kód po bajtech, ty dekóduje a podle nich provádí požadované činnosti. Zápis programů v číselných hodnotách by byl velmi nepohodlný, proto se používá zápis pomocí mnemotechnických názvů instrukcí. Takovému zápisu se říká „jazyk symbolických adres“ (JSA), nebo (nesprávně, ale všeobecně srozumitelně) „zápis v assembleru“ (anglicky *assembly language*).

Program se v JSA zapisuje jako v jiných jazycích do textového souboru po řádcích. Řádky mají přesně definovaný formát a skládají se z následujících prvků:

- Návěští: řetězec znaků [A-Z, 0-9, podtržítko], který nezačíná číslicí. Některé překladače rozlišují jen prvních 6 znaků, jiné prvních 8, další pak všechny. Návěští je zakončeno dvojtečkou.
- Instrukce: mnemotechnická zkratka instrukce, popřípadě pseudoinstrukce (direktivy)
- Parametry: Některé instrukce a direktivy vyžadují další údaje, se kterými musí instrukce pracovat. Zapisují se za mnemotechnické označení instrukce, od které se oddělí mezerou (nebo mezerami). Pokud je parametrů víc, oddělují se od sebe čárkami.
- Poznámka: Cokoli, co si potřebujete poznamenat. Na začátku je znak středník (;), vše, co je za tímto znakem, překladač ignoruje.

Některé assemblery měly ještě přísnější pravidlo: cokoli začínalo v prvním sloupci, to bylo návěští. Pokud na řádku žádné návěští nebylo, musel takový řádek začínat tabelátorem nebo mezerou.

Ukažme si různé typy řádků.

Prázdný řádek překladač prostě ignoruje.

Řádek, kde je jen poznámka: Pokud je prvním znakem na řádku (mezery nepočítáme) znak středník, bere se zbytek řádku jako poznámka a překladač jej ignoruje.

Řádek s instrukcí: Na řádku je uvedena existující (pseudo)instrukce, popřípadě její parametry, má-li nějaké. Překladač ji přeloží na odpovídající operační kód. Za parametry může být ještě poznámka, kterou překladač ignoruje

Řádek s návěštím: Libovolný typ výše uvedených řádků může ještě začínat návěštím, tj. řetězcem ukončeným dvojtečkou. V takovém případě překladač přiřadí tomuto řetězci adresu, která odpovídá internímu počítadlu adres. (K tomu se ještě vrátím.)

Podívejme se na ukázku kódu:

```
1) ;Program začíná
2)          .ORG 8000H; nastavíme počátek kódu
3)
4) ZACATEK:
5)           NOP
6)           NOP
7) NAVEST:   NOP
8)           NOP
```

V tomto kódu najdete řádek s komentářem (1), prázdný řádek (3), řádky s instrukcemi (5, 6, 7, 8) i řádky s návěštím (4, 7).

Použil jsem jen jednu instrukci, totiž instrukci NOP, kterou znají snad všechny procesory a její význam je prostý: Nedělej nic. Možná se vám zdá zbytečné mít instrukci, která nedělá nic, ale ona se docela dobře hodí – když potřebujete někde „chvilku počkat", nebo když si chcete někde nechat místo pro budoucí změnu. *U procesorů 8080 / Z80 je její kód 0, u procesoru 6502 je to 0EAh.*

Instrukce „.org" není instrukcí v pravém slova smyslu. Nemá totiž žádný operační kód. Pouze říká překladači: Teď se chovej tak a tak. Patří tedy mezi pseudoinstrukce (nebo taky „direktivy"). Org je zkratka pro „origin", tedy počátek. Tady říká překladači: *Ber to tak, že tady, jak jsem já, bude nějaká konkrétní adresa, a následující instrukce tedy ukládej od té adresy dál.*

Pseudoinstrukce .org má jeden parametr – totiž tu adresu. Tady je zapsaná v hexadecimální podobě.

Jak lze zapsat konstanty v assembleru?

Máme následující možnosti:

Hexadecimální zápis

- Pomocí číslic 0-9 a znaků A-F. Číslo začíná vždycky číslicí a je ukončeno znakem H (h). Takže: 1234h, 00AAH, 23beh, 0cfh, 18H, …
- Pomocí číslic 0-9 a znaků A-F, před kterým je znak $. Takže: $1234, $00AA, $23be, $cf, $18, …
- Některé překladače umožňují i zápis v „céčkové“ syntaxi „0x…“

Binární zápis

- Řetězec znaků 0,1 ukončený písmenem „b“
- Řetězec znaků 0,1, před kterým je znak „%“

Desítkové číslo

- Zapisujte tak, jak jsme zvyklí.

Znaková konstanta

- Znak zapsaný v apostrofech nebo uvozovkách. Hodnotou je ASCII kód daného znaku. Např. „@“ = 64 = 40h

Tady je na místě připomenout, že konkrétní syntaxe jednotlivých překladačů se může lišit. V dalším popisu budu vycházet z tvaru, jaký jsem implementoval v překladači ASM80. Totéž platí i pro pseudoinstrukce – někde je správný zápis „org“, u jiného překladače je to důsledně „.org“ s tečkou. ASM80 povoluje oba tvary. Původní assemblery braly zápis bez tečky.

Listing – výpis přeloženého programu

Když si vezmete výše zapsaný kód a zkusíte ho přeložit překladačem – např. výše zmíněným ASM80 – získáte na výstupu jednak binární podobu (která nebude moc zajímavá), jednak takzvaný *listing*, což je výpis programu spolu s adresami a instrukčními kódy.

```
0000          ;Program začíná
8000                    .ORG 8000H; nastavíme počátek kódu
8000
8000          ZACATEK:
8000 00                 NOP
8001 00                 NOP
```

```
8002 00                 NOP
8003 00       NAVEST:   NOP
8004 00                 NOP

_PC 8004
ZACATEK 8000
NAVEST 8003
```

Každý řádek začíná adresou, na jaké je daná instrukce uložena. Vidíte, že na prvním řádku ještě překladač neví, kde chceme program mít, a tak počítá s tím, že bude uložený od adresy 0. Na druhém řádku pseudoinstrukcí .org říkáme, že program bude v paměti uložen od adresy 0100h. Překladač si proto nastaví interní počítadlo adres (_PC) na hodnotu 0100h. Třetí řádek je „prázdný řádek s návěštím". To znamená, že do tabulky návěští je zařazeno návěští se jménem ZACATEK a je mu přiřazena hodnota interního počítadla adres, tedy 0100h.

Čtvrtý řádek obsahuje konečně nějakou reálnou instrukci. Je to instrukce NOP, je přeložena na operační kód (00) a ten bude uložen na adrese 0100h. Počítadlo se posune na následující adresu, tedy 0101h.

Pátý řádek: NOP, kód je zase 00, je uložen na adrese 0101h a počítadlo se posouvá…

Šestý řádek: NOP, kód je zase 00, je uložen na adrese 0102h a počítadlo se posouvá…

Na sedmém řádku je návěští NAVEST. Je mu tedy přiřazena hodnota interního počítadla adres (0103h). Na tomtéž řádku je i instrukce. Její operační kód je uložen na tuto adresu a pokračujeme dál…

Na samotném konci listingu vidíte výpis všech návěští. _PC je interní počítadlo adres (které skončilo na hodnotě 0104h), no a návěští ZACATEK a NAVEST mají ty hodnoty, jaké jsme si popsali výše.

Základy jsou tedy jasné, pojďme na skutečné programování v assembleru.

6.3 První program v assembleru 8080

Napíšeme si první program. Nelžu, opravdu první program v assembleru pro procesor 8080!

Nebude to Hello world, bude to něco sofistikovanějšího. Sečteme dvě čísla. Třeba 18H a 23H. (Výsledek je 03BH, kdo nevěří, ať tam běží.) Nebojte se toho, že nezačínám od základů, že ještě žádnou instrukci neznáte (ne, znáte jednu, z minulé kapitoly víte o NOP) ani že neřeknu hned všechno (třeba zamlčím existenci „registru" M). Jen si odskočíme, abyste viděli, jak se v assembleru pracuje, a už v příští lekci to budeme brát systematicky instrukci po instrukci…

Ale teď: Jak sečíst dvě čísla v assembleru? Asi už tušíte, že to nebude prosté jako napsat „18+23“. Procesor 8080 totiž nic takového jako „sečti dvě čísla, která ti sem napíšu“ neumí. Co umí?

Umí sečíst dvě čísla, to je v pořádku. K sečtení slouží instrukce ADD, se kterou se seznámíme v dalších lekcích. ADD má jeden parametr, a tím je jméno registru. Vzpomeňte si na kapitolu o architektuře 8080: máme k dispozici registry B, C, D, E, H, L a akumulátor A. Instrukce ADD má symbolický tvar „ADD r“ (kde „r“ je právě to jméno registru) a její funkce je následující: Vezme obsah registru A, přičte k němu obsah zadaného registru a výsledek uloží zase do A. Nějak takhle:

INSTRUKCE	FUNKCE
ADD B	A = A + B
ADD C	A = A + C
ADD D	A = A + D
ADD E	A = A + E
ADD H	A = A + H
ADD L	A = A + L
ADD A	A = A + A

Postup bude tedy následující: Do jednoho registru si uložíme první číslo, tedy 18H, do druhého registru druhé číslo (23H), a pak je pomocí instrukce ADD sečteme. Protože ADD vyžaduje, aby jeden ze sčítanců byl v registru A, ušetříme si práci tím, že jedno z čísel umístíme rovnou do A, druhé třeba do B.

Jak do registrů nahrajeme nějaké číslo? Použijeme k tomu instrukci MVI (Move Immediate). Tato instrukce má dva parametry – první je jméno registru, druhý je hodnota 0-255, tedy 1 bajt. Symbolicky zapsáno je to „MVI r, n8“ (n8 se označuje osmibitové číslo). Všimněte si, že pořadí parametrů je podobné jako ve vyšších jazycích, tedy vlevo je to, KAM se ukládá, vpravo CO se tam má uložit. V našem případě použijeme dvojici instrukcí „MVI A, 23H“ a „MVI B, 18H“.

Nakonec výsledek uložíme do paměti na vhodnou adresu. Která to je? No, na adresách 0000h až 7FFFh je ROM, tam nelze zapisovat. Od adresy 8000h bude náš přeložený program. Ideální adresa je tedy „hned za programem“. Nevýhoda je, že pokud k programu něco přidáte, tak se tato adresa změní.

Poslední krok je návrat do monitoru. Aniž bych teď podrobněji vysvětloval důvod, tak řeknu, že použijeme instrukci JMP 0.

Celý program bude vypadat takto:

```
  ORG 8000h
  MVI A,023h
  MVI B,018h
  ADD B
  STA VYSLEDEK
  JMP 0
VYSLEDEK: DS 1
```

Direktiva DS na posledním řádku znamená: „Tady nechej prostor o velikosti X bajtů“ – v tomto případě 1 bajt. Tím jsme si udělali místo na uložení výsledku.

Co se tam děje? Nejprve říkáme překladači, ať si nastaví adresu 0, od té adresy se budou ukládat instrukce do paměti. Pak je instrukce, která do registru A uloží hodnotu 23h, pak do registru B uložíme 18h, a pak vykonáme ADD B, což je, jak už víme, ekvivalent „A = A + B“. Výsledek uložíme do paměti na adrese VYSLEDEK a skočíme zpět do Monitoru.

Ukažme si ještě listing – z něj, mimo jiné, zjistíme, která že adresa je ten VYSLEDEK:

```
8000             .ORG   8000h
8000   3E 23      MVI   A,023h
8002   06 18      MVI   B,018h
8004   80         ADD   B
8005   32 0B 80   STA VYSLEDEK
8008   C3 00 00   JMP 0
800B             VYSLEDEK: DS 1
```

Instrukce MVI zabírá vždy dva bajty. První bajt je kód instrukce (3Eh = MVI A; 06h = MVI B), druhý bajt je hodnota, která se má do registru nahrát. Instrukce ADD je jednobajtová, STA a JMP jsou tříbajtové – vždy operační kód (1 bajt) a adresa (2 bajty). A konečně magický VYSLEDEK – bude uložen do paměti na adresu 800Bh, což je adresa následující za poslední instrukcí.

Tak, to by bylo. A teď otázka nejdůležitější: Bude to fungovat? Bude. V tuhle chvíli vám nezbývá nic jiného, než mi věřit. Anebo si to vyzkoušet. Buď použijte svůj oblíbený assembler, spusťte si kód na svém počítači Alpha – no a nebo využijte ASM80, kde si můžete kód napsat, přeložit a odladit v emulátoru, a to všechno přímo v prohlížeči (tedy, pokud máte aspoň trošku modernější prohlížeč)!

Náš program nic moc nedělá. Když si ho spustíte, proběhne bleskurychle a uloží výsledek na adresu 800Bh. Pomocí Monitoru si můžete ověřit, že výsledek odpovídá očekávání…

Program můžete přenést do Alphy například tak, že pomocí příkazu M8000 začnete měnit obsah paměti od adresy 8000h a ručně budete zadávat jednotlivé hodnoty (vždy ukončené ENTERem): 3E 23 06 18 …

Nebo si necháte vygenerovat HEX soubor a pošlete jej po terminálu do počtíače.

Pomocí příkazu D8000 si ověříte, že program je v paměti tak, jak má být.

Příkazem G8000 jej spustíte.

Poté, co doběhne, zadejte opět M800B a prohlédněte si výsledek.

Ale co, Řím taky nepostavili za den!

6.4 Instrukce 8080 – adresní módy a registry

Jak instrukce ví, kde má vzít data, se kterými pracuje?

Některé instrukce (jako už zmíněný NOP) nevyžadují žádné parametry a nepracují s žádnými daty. Tam je to jasné. Ovšem většina instrukcí *něco* dělá, to znamená, že potřebuje odněkud vzít nějaké údaje, někam je uložit atd. Procesor 8080 má čtyři různé způsoby, jak říct, kde se data nacházejí.

Přímý operand

Tímto způsobem zapíšeme přímo hodnotu do kódu a procesor ji použije. S přímým operandem pracuje např. instrukce MVI, kterou jsme si ukázali v minulé kapitole. Součástí instrukce je přímo hodnota, která je použita.

Přímé adresování

Je podobné předchozímu. I v tomto případě je součástí instrukce hodnota, která ale není použita tak, jak je, ale je brána jako adresa (v paměti nebo vstupně-výstupní brány). Například instrukce „LDA 554Ah“ používá právě tento typ adresování. Procesor udělá to, že hodnotu 554Ah vezme jako adresu místa v paměti, přečte z něj jeden bajt a ten uloží do registru A.

Adresování registrem

Součástí instrukce je v takovém případě jméno registru, registrového páru nebo dvojice registrů, se kterými se operace provádí.

Nepřímé adresování registrem

Instrukce bere hodnotu v dvojici registrů (BC, DE, HL) nebo v registru SP jako adresu do paměti. Pracuje pak s obsahem na této adrese.

Některé instrukce pracují s tzv. „implicitním operandem“, což znamená, že operand (jeden z operandů) je daný už samotnou instrukcí. Například zmíněná instrukce ADD používá dva operandy – jeden je vždy A a druhý je určen parametrem.

6.5 Registry

U osmibitových instrukcí můžeme použít libovolný osmibitový registr – tedy A (akumulátor), B, C, D, E, H, L a M.

Moment! Registr M? Nikde jsem se o něm nezmínil… Není to nějaký renonc?

Není. „Registr M“ (memory) je ve skutečnosti buňka v paměti na adrese, která je uložena v registrech H a L. Pokud je v registru H hodnota 12h a v registru L hodnota 34h, bude instrukce, která pracuje s „registrem M“, pracovat ve skutečnosti s obsahem paměti na adrese 1234h. Instrukce ADD M přičte k obsahu registru A hodnotu z paměti na adrese 1234h.

Všimněte si, že nelze přímo přistupovat k registru F. Ono to většinou není k ničemu dobré, ale kdybyste opravdu potřebovali znát jeho hodnotu, dá se to obejít takovým trikem, který si popíšeme později.

Pokud instrukce pracuje s šestnáctibitovou hodnotou, využívá šestnáctibitový registr SP nebo dvojice registrů B, D, H. „Dvojregistr“ B jsou vlastně registry B a C, v B je vyšší bajt, v C nižší. Obdobně pro D a H. Instrukce pro práci se zásobníkem používají „šestnáctibitový registr“ PSW, což je ve skutečnosti dvojice registrů A a F (A vyšší byte, F nižší).

Nezapomeňte: Procesory Intel, Zilog Z80 i 6502 používají zápis vícebajtových čísel ve formátu „Little Endian“. Znamená to, že na nižší adrese je méně významná část čísla. Příklad: Na adresu 0100h uložíme naši obligátní hodnotu 1234h. Paměť je organizovaná po bajtech, hodnota je dvoubajtová (16 bitů), jak to tedy uložit? Použijeme dvě buňky paměti, 0100h a 0101h. Na tu nižší přijde nižší část čísla (34h), na tu vyšší zase vyšší část (12h). V paměti to tedy bude vypadat nějak takto:

0100h: 34h

0101h: 12h

Některé procesory (z těch známějších především procesory od Motoroly) používají opačný způsob („Big Endian“), kdy se části čísla zapisují do paměti od nejvýznamnějšího bajtu.

6.6 Přesuny dat

První skupina instrukcí, kterou si probereme, přesouvá data. Z registrů, do registrů, i z paměti a do paměti…

Až budete psát programy v assembleru, zjistíte, že nejvíc kódu nezabírají nějaké hyper super složité výpočty, ale přesouvání dat odněkud někam. V podstatě většina činností, které mikroprocesorový systém navenek dělá, je z valné části *přesouvání dat*. Samotného počítání je minimum. Takže je jen logické, že začneme právě u těchto instrukcí.

MOV

Univerzální instrukce pro kopírování obsahu z jednoho registru do druhého. Instrukce má dva operandy – názvy osmibitových registrů. Jako první se uvádí cílový, jako druhý zdrojový. MOV A,B tedy vezme obsah registru B a zkopíruje ho do registru A. MOV A,C zkopíruje obsah z registru C do registru A. MOV C,A přesně naopak – z registru A do registru C.

Ptáte se, kolik je kombinací? No všechny dostupné – tedy 8 x 8 = 64. Čímž neříkám, že jsou všechny smysluplné či funkční. Instrukce MOV A,A je samo sebou platná, zkopíruje obsah z registru A do registru A, ale upřímně: k čemu to je?

Znovu připomínám, že M je „pseudoregistr", který ve skutečnosti odkazuje na místo v paměti s adresou, která je uložená v dvojici registrů H, L. Takže MOV A,M znamená „vezmi obsah z paměti na adrese, která je v registrech H,L, a zkopíruj ho do registru A". MOV M,E znamená „vezmi obsah registru E a zkopíruj ho do paměti na adresu, která je v registrech H,L". No a MOV M,M – to je výjimka, a docela nebezpečná. Taková instrukce neexistuje. Její operační kód zaujímá instrukce HLT, která zastaví procesor. Opravdu. Procesor pak čeká až do chvíle, kdy přijde přerušení nebo RESET.

MVI

Tato instrukce vloží do osmibitového registru přímo zadanou hodnotu.

LXI

Tato instrukce vloží do registrového páru B, D, H nebo registru SP přímo zadanou šestnáctibitovou hodnotu.

LDA

Instrukce má jeden parametr – šestnáctibitovou adresu. Obsah paměti na téhle adrese zkopíruje do registru A

STA

Protipól předchozí instrukce – ukládá hodnotu z registru A do paměti na adresu zadanou jako operand.

A nyní si prosvištíme nektera sloviška a cele fěty – pardon, zkusíme si, jak tyhle instrukce pracují.

```
8000                .ORG 8000h
8000 3E 0A          MVI A,10
8002 06 14          MVI B,20
8004 21 40 80       LXI H,8040h
8007 77             MOV M,A
8008 3A 03 80       LDA 8003h
800B 32 41 80       STA 8041h
```

Kód přeložte a spusťte emulaci (pokud používáte ASM80). První instrukce je MVI A, 10. Po provedení prvního kroku bude tedy v registru A hodnota… aha! Je tam 0Ah. Emulátor totiž ukazuje všechny hodnoty hexadecimálně, zatímco v kódu je zapsaná konstanta 10, bez H na konci, takže je to tedy desítková hodnota, a 10 desítkově je v šestnáctkové soustavě 0A.

U reálného počítače můžete postupovat takto: nahrajte kód do paměti. Pomocí příkazu X8002 nastavte bod přerušení (breakpoint) na adresu 8002h. Spusťte program příkazem G8000. Procesor provede první instrukci na adrese 8000h, a na adrese 8002h narazí na breakpoint. V tu chvíli se opět zavolá Monitor, který oznámí, že běh programu byl zastaven na adrese 8002h. Příkazem R si teď můžete prohlédnout obsah registrů.

Takže tedy znovu: První instrukce uloží do registru A hodnotu 0Ah (=10), druhá instrukce uloží do registru B hodnotu 14h (=20). Třetí instrukce je LXI H, *číslo*– mrkněte se o kousek výš… jasně! Tahle instrukce uloží zadané číslo do dvojice registrů HL. Tady je to 8040h, takže do H půjde vyšší (80h) a do L nižší (40h) část čísla.

Na dalším řádku je instrukce MOV M,A. Z toho, co o ní víme, by se mělo stát následující: Obsah registru A (což je teď těch 0Ah) se zkopíruje do paměti na adresu, která je uložená v registrech HL. Tak schválně – v registrech HL je uložena hodnota 8040h, takže by se v paměti na adrese 8040h měla objevit hodnota 0Ah. Zkuste si sami – vidíte, že vlevo nahoře, v oblasti nadepsané MEMORY, se na adrese 8040h objeví právě ta hodnota 0Ah.

Bystřejší si všimli, že od adresy 8000h jsou v paměti nějaká čísla. Ti, co jsou ještě bystřejší, už vědí: To je přeci ten náš program! A právě do té oblasti sáhne další instrukce – LDA 8003h vezme obsah paměti na adrese 8003h, což je zrovna shodou okolností číslo 14h (které je součástí instrukce MVI B, 20) a uloží ho do registru A.

Poslední instrukce pak uloží obsah registru A na adresu 8041h, takže od adresy 8040h budete mít vedle sebe hezky 0Ah a 14h. Máte? Skvělé, pojďme na další instrukce!

LDAX

Tak jako LDA vezme hodnotu z paměti na adrese, kterou mu zadáme, a uloží ji do A, tak i LDAX vezme hodnotu z paměti a uloží ji do A. Adresu, se kterou se bude pracovat, najde v registrovém páru B nebo D (tedy v registrech B, C, resp. D, E).

STAX

Jako má LDA svoje STA, tak má LDAX svůj STAX. Funguje stejně jako LDAX, jen přenos dat probíhá v opačném směru.

LHLD

Tahle instrukce opět pracuje s dvojicí bajtů. Jako parametr dostane adresu a vykoná následující: Do registru L zkopíruje hodnotu z paměti na dané adrese a do registru H zkopíruje hodnotu z paměti na adrese o 1 vyšší.

SHLD

Už to tak je. Instrukce L*něco* mají svou obdobu v podobě *Sněco*. Tam, kde L z paměti čte (load), tam S do paměti zapisuje (store). Takže SHLD uloží obsah registru L na zadanou adresu, obsah registru H na adresu o 1 vyšší.

Všimněte si, že LHLD i SHLD dodržují pravidlo „malého indiána" – na nižší adresu jde méně významný bajt.

XCHG

Poslední instrukce pro přesun dat je tak trošku výjimka. Nemá žádné operandy – to, s čím pracuje, je dáno implicitně (jsou to dvojregistry D a H). A na rozdíl od předchozích instrukcí, které kopírovaly hodnoty, tato instrukce je vyměňuje. XCHG vymění obsah registrů D a E za obsah registrů H a L. Tedy to, co bylo v D, bude teď v H – a naopak. Kdybychom udělali třeba MOV H,D a MOV L,E, dosáhli bychom něčeho jiného obsah registrů D a E by se nevyměnil, ale ZKOPÍROVAL, takže by bylo v D totéž, co v H, a v E totéž, co v L.

Pokud chceme vyměnit údaje ve dvou registrech, musíme použít pomocný registr. Například výměnu obsahu registrů B a C zařídíme takto:

```
MOV A,B
MOV B,C
MOV C,A
```

Tedy přes registr A. Jeho původní hodnotu samozřejmě ztratíme…

A opět drobná ukázka. Až si pohrajete, můžete si zkusit jedno cvičení. První samostatný program. Poslyšte zadání, je jednoduché: Program prohodí obsah registrů BC a HL.

6.7 Příznaky a zásobník

Procesor potřebuje vědět, jak dopadla ta která instrukce, a občas si potřebuje nějaká data odložit na později.

Příznaky

O registru F jsem se už zmiňoval. Je to takový lehce mystický registr, do kterého se nedá přímo zapisovat a ze kterého se nedá přímo číst. Psal jsem, že v něm jsou uložené takzvané *příznaky*. Co to je?

Některé instrukce, převážně ty, co něco počítají, mohou dát najevo, že výsledek je v určitém formátu. Například že je nulový, záporný, jakou má paritu, nebo že při výpočtu přetekl výsledek z rozsahu 0-255. A přesně tyto informace ukládá jako *příznakové bity* do registru F.

U procesoru 8080 je příznakových bitů 5 a jsou v registru uloženy takto:

BIT	7	6	5	4	3	2	1	0
PŘÍZNAK	S	Z	0	AC	0	P	1	CY

- S (sign) informuje o znaménku výsledku. Je-li výsledek kladný, je to 0, je-li výsledek záporný, je to 1
- Z (zero) je roven 1 v případě, že výsledek je nula. Pokud je nenulový, je Z = 0
- AC (auxiliary carry) je roven 1, pokud při operaci došlo k přenosu přes polovinu bajtu (mezi nižší a vyšší čtveřicí)
- P (parity) je nastaven vždy tak, aby doplnil počet jedniček ve výsledku na lichý (tedy je roven 0, je-li počet jedniček lichý, 1, je-li sudý)
- CY (carry) je 1, pokud dojde k přenosu z nejvyššího bitu.

Termín „přenos“ je možná nejasný, pojďme si ho upřesnit. Co se stane, když budu sčítat čísla 0FAh a 0Ah (250 a 10) v osmibitovém registru? Výsledek je 260, hexadecimálně 104h – a to je číslo, které má devět bitů. Devátý (nejvyšší) bit se do registru nevejde, tam zůstanou jen bity 0-7 (tedy osm bitů). Pokud taková situace nastane, je příznak CY nastaven na jedničku.

Tento příznak je velmi užitečný, pokud sčítáte vícebajtová čísla. Sečtete nejprve čísla na nižších řádech, a pokud je náhodou větší než možný rozsah, *přenesete si* jedničku do vyššího řádu.

Tentýž příznak funguje i pro odčítání – zde zase funguje jako označení „výpůjčky“ do nejvyššího řádu v případě, že výsledek je menší než 0. Když od čísla 04h odečteme číslo 05h, bude výsledek 0FFh (nezapomeňte, že hodnoty jdou za sebou a po nule je 0FFh). Příznak CY bude nastaven („půjčili“ jsme si jedničku), příznak S bude nastaven (výsledek je záporný).

Ve skutečnosti je -1 a 255 jedno a totéž číslo a je jen na nás, jestli se na něj díváme jako na číslo se znaménkem, nebo na číslo bez znaménka. -2 je 254, -3 je 253 atd. Pokud chceme v binární soustavě převést číslo na jeho zápornou hodnotu, uděláme bitovou negaci (zaměníme 0 za 1 a opačně) a přičteme jedničku (procesor na to samo sebou má vhodnou instrukci).

```
8000                    .ORG 8000h
8000 3E FA              MVI a,0FAh
8002 06 0A              MVI b,0Ah
8004 80                 ADD b
8005 06 05              MVI b,5
8007 90                 SUB b
8008 00                 NOP
```

A k čemu jsou příznaky dobré? Jednak pomáhají s výpočty, a jednak se můžete podle výsledků rozhodovat a provádět podmíněné skoky a volání podprogramů.

Zásobník

Občas procesor potřebuje, jak jsem už psal, nějaká data „odložit" na později. Ať už to jsou data, co si potřebuje uložit programátor, nebo třeba adresa, kam se má program vrátit z podprogramu. Používá k tomu strukturu zvanou zásobník (stack). Stack je struktura typu LIFO (Last In – First Out, tedy co se poslední uloží, to se první vrátí). Implementace zásobníku je jednoduchá: V paměti si vyhradíte oblast, a adresu jejího konce +1 uložíte do registru SP (Stack Pointer). Zásobník je připravený.

Do zásobníku se vždy ukládají dva bajty, tedy buď adresa, nebo celý registrový pár. Pokud na zásobník uložíme pár HL, stane se toto: Procesor sníží SP o 1, na tuto adresu uloží obsah registru H, pak zase sníží SP o 1 a na tuto adresu uloží obsah registru L. Teď uložíme třeba pár BC: Procesor sníží SP o 1, na tuto adresu uloží obsah registru B, pak zase sníží SP o 1 a na tuto adresu uloží obsah registru C.

Když teď vyzvedneme hodnotu ze zásobníku – třeba do registrů DE – procesor vezme obsah z adresy SP, uloží ho do registru E, SP zvýší o 1, vezme obsah z adresy SP, uloží ho do registru D a opět zvýší SP o 1. SP teď ukazuje na předchozí položku a v DE máme tu poslední uloženou. Což byla, shodou okolností, zrovna hodnota z registrů B a C – nikde není psáno, že se musí údaje vybrat do téhož registru, z jakého byly uloženy.

Zásobník se tedy posouvá od vyšších adres k nižším. Proto se umisťuje na konec RAM, aby měl dost prostoru kam růst. Což s sebou nese riziko, že pokud uložíte do zásobníku příliš mnoho hodnot, klesne SP až tak, že další uložená data můžou přepsat váš program (tomuto stavu se říká *přetečení zásobníku*, stack overflow). Pokud umístíte zásobník POD program, může se zase stát, že poklesne až třeba do oblasti, kde je ROM. Oba stavy jsou chybné, znamenají téměř stoprocentně pád programu a většinou nastávají, když se program dostane do nekonečné smyčky, ve které se odskakuje nebo ukládá.

PUSH a POP

K uložení obsahu registrů do zásobníku slouží instrukce PUSH s jedním parametrem, a tím je jméno dvojice registrů (B, D, H), nebo „PSW“ – o kterém jsem se už zmiňoval. PSW (Program Status Word) je dvojice registrů A a F. Pro vybrání hodnot ze zásobníku slouží instrukce POP (a stejné parametry). Pojďme si zaexperimentovat: nastavíme si zásobník do paměti a budeme do něj ukládat a zase vybírat data.

```
8000                    .ORG 8000h
8000 3E 11              MVI a,011h
8002 01 33 22           LXI b,2233h
8005 11 55 44           LXI d,4455h
8008 21 77 66           LXI h,6677h
800B 31 40 80           LXI sp,8040h
800E                    ;ukládáme
800E F5                 PUSH psw
800F C5                 PUSH b
8010 D5                 PUSH d
8011 E5                 PUSH h
8012                    ;vybirame
8012 C1                 POP b
8013 E1                 POP h
8014 D1                 POP d
8015 3E 00              MVI a,0
8017 F1                 POP psw
```

Všimněte si nejprve toho, jak se do paměti ukládají data, když se vykonává instrukce PUSH. Později, při načítání zpět, je načítáme v jiném pořadí, takže registry nemají původní obsah.

Na konci minulé kapitoly jsem zadal úkol: vymyslet způsob, jak prohodit obsah registrových párů BC a HL. Předpokládám, že jste napsali něco takového:

```
MOV A,B
MOV B,H
MOV H,A
MOV A,C
MOV C,L
MOV L,A
```

Což je správně, ale přijdeme tím o obsah, který je v registru A. Pokud nám to nevadí, není co řešit. Ale pokud by nám to vadilo, dáme na začátek PUSH PSW a na konec POP PSW. Samozřejmě, spoléháme na to, že zásobník je správně nastavený, ale vzhledem k tomu, jak často je používaný, tak je to jedna z prvních věcí, co slušný programátor udělá.

No a s informacemi z této lekce můžete stejné zadání vyřešit třeba takto:

```
PUSH B
PUSH H
POP B
POP H
```

SPHL

Tato instrukce naplní registr SP obsahem dvojice registrů HL. Tedy SP = HL. Ptáte se, jak se zařídí opak, tedy jak zjistíte hodnotu registru SP a uložíte ji do HL? Tipujete instrukci HLSP? Ne, žádná taková není. Musíte použít konstrukci LXI H,0 a DAD SP (o instrukci DAD si řekneme víc v další kapitole, zde jen zmíním, že přičte k dvojici registrů HL obsah jiné dvojice registrů, popřípadě, jako zde, obsah registru SP).

XTHL

Další instrukce, která vyměňuje hodnoty dvojic registrů. XTHL vymění hodnotu dvojice registrů HL s „poslední uloženou hodnotou na zásobníku". Tedy: poslední hodnota, uložená na zásobníku, se ocitne v HL, a hodnota HL bude poslední hodnota na zásobníku.

6.8 Aritmetické operace

Je to počítač, ne? Tak to má taky počítat!

Procesor samo sebou není živ jen přesouváním dat tam a zpátky. Občas taky musí něco spočítat. K tomu mu slouží aritmetické instrukce pro sčítání, odčítání a porovnání. (Kde je násobení, dělení a hyperbolický sinus? Dozvíte se později!)

ADD

Instrukce ADD r vezme obsah v osmibitovém registru r (tedy A, B, C, D, E, H, L nebo M, což je ve skutečnosti obsah paměti, ale to už jsem psal asi pětkrát…) a přičte ho k registru A. Výsledek sčítání uloží do registru A. Pokud vám připadá, že se registr A používá nějak často, tak se vám to nezdá, je to opravdu tak, a protože se k němu vztahují skoro všechny aritmetické i logické instrukce, tak se mu říká „akumulátor".

Po sčítání nastaví patřičně příznaky S, Z, AC, P a CY – viz popis funkce příznaků v minulé kapitole.

Pojďme se podívat na pár příkladů sčítání a toho, jak ovlivní příznaky. Všechna čísla budou hexadecimálně.

Sčítanec 1	Sčítanec 2	Výsledek	S	Z	AC	CY	POZNÁMKA
03	05	08	0	0	0	0	Nic zvláštního, vešli jsme se dokonce do jedné pozice
09	08	11	0	0	1	0	Při sčítání bylo třeba přenést jedničku z nejnižší pozice do vyšší
FF	01	00	0	1	1	1	Výsledek je 0 (Z), přetekl rozsah registru (CY) a přenášela se jednička mezi čtveřicemi bitů (AC)
F0	05	F5	1	0	0	0	Pokud používáme aritmetiku se znaménkem, tak je výsledek záporný (S)

Se znaménkem…?

Jak to vlastně je, když chceme použít místo čísel bez znaménka (0-255) čísla se znaménkem (-128 až 127)? Kde to dáme procesoru najevo? Odpověď zní: Nikde! Procesoru to je úplně jedno. Vezměme si poslední řádek z předchozí tabulky. Pokud se bude jednat o čísla bez znaménka, tak sčítáme hodnotu 240 (F0h) a 5. To je dohromady 245, tedy hexadecimálně F5h. Když je budeme vnímat jako čísla se znaménkem, tak sčítáme -16 (F0h) a +5. Výsledek je -11, což je hexadecimálně F5h.

Předposlední řádek je podobný: při sčítání bez znaménka je FFh+01h=100h, což se do osmi bitů nevejde, máme tedy výsledek 00 a nastavený přenos (1). Pokud sčítáme se znaménkem, sčítáme -1 (FFh) a +1 a výsledek je – opět nula!

To, jestli pracujeme s čísly se znaménkem nebo bez znaménka, si musíme určit coby programátoři sami. Procesor je zpracovává stále stejně a je jen na nás, jak je interpretujeme. Jen je důležité je interpretovat vždy stejně.

ADI

Často potřebujeme přičíst nějakou předem známou konstantu. Pomocí ADD bychom si ji museli nejprve uložit do některého registru, a ten pak přičíst. Naštěstí to jde rychleji, pomocí jedné instrukce ADI. Ta má jediný parametr, a tím je osmibitová konstanta. Tu přičte k registru A. Zbytek, tedy nastavování příznaků apod., platí stejně jako u instrukce ADD.

ADC

Funguje stejně jako instrukce ADD, ale k výsledku přičte ještě hodnotu příznaku CY. Hodí se to například při sčítání větších čísel. Představme si, že máme dvě šestnáctibitová čísla, jedno v registrech BC, druhé v registrech DE, a chceme je sečíst a výsledek uložit do registrů BC. Takhle jednoduše to nejde, žádnou instrukci na to nemáme, takže to musíme udělat krok po kroku. Nejprve

si sečteme čísla na nižších pozicích (C + E) a výsledek uložíme do C. Ovšem pokud součet těchto číslic bude větší než 255, tak musíme při sčítání vyšší pozice přičíst (přenesenou) jedničku! Což je bez problémů, protože, jak víme: ADD nastaví správně příznak CY, a ADC ho dokáže započítat. Součet tedy naprogramujeme třeba takto:

```
MOV     a,c
ADD     e
MOV     c,a
MOV     a,b
ADC     d
MOV     b,a
```

Naštěstí instrukce MOV nijak neovlivňuje příznaky, takže stav příznaku CY tak, jak ho nastaví instrukce ADD, se až do provedení instrukce ADC nezmění.

Funguje to? Ověříme v emulátoru! Sečteme si čísla 1000 a 2000 (03E8h a 07D0h), výsledek by měl být 3000 (0BB8h)… Všimněte si, že při součtu nižších řádů (E8h+D0h) je nastaven bit přenosu, protože výsledek je větší než FFh.

```
8000                    .ORG 8000h
8000 01 E8 03           LXI b,1000
8003 11 D0 07           LXI d,2000
8006 79                 MOV a,c
8007 83                 ADD e
8008 4F                 MOV c,a
8009 78                 MOV a,b
800A 8A                 ADC d
800B 47                 MOV b,a
```

ACI

Jako má ADD své ADI, tak má ADC své ACI. K registru A přičte zadanou konstantu a hodnotu příznaku CY.

SUB, SUI

Instrukce SUB od hodnoty v registru A odečte hodnotu zadaného registru a výsledek uloží do A. SUI dělá totéž, ale s osmibitovou konstantou. Pokud je výsledek menší než 0, tak se pokračuje od FFh dolů (-1 = FFh, -2 = FEh, …) a je nastaven příznak CY, který má tentokrát význam „výpůjčky“ (borrow).

SBB, SBI

Totéž jako SUB, SUI, ale tentokrát od výsledku odečte 1, pokud je nastaven příznak CY.

CMP, CPI

Tyto instrukce porovnají obsah registru A s jiným registrem (CMP) nebo konstantou (CPI) a nastaví příznakové bity takto:

POROVNÁNÍ	S	Z	CY
A=n	0	1	0
A>n	0	0	0
A<n	1	0	1

Podmíněné instrukce skoku se pak mohou podle těchto hodnot rozhodnout.

Zajímavost: Instrukce CMP a CPI vnitřně dělají totéž jako instrukce SUB, SUI, tedy odečtou od A hodnotu parametru, nastaví příznaky, ale výsledek neuloží do A, ale zahodí ho.

INR, DCR

Velmi častá operace je zvýšení hodnoty o 1 nebo naopak odečtení 1. Vyšší jazyky, které vyšly z jazyka C, na to mají i operátory „++“ a „–“. INR přičte k registru 1, DCR odečte 1. Instrukce ovlivní příznaky S, Z, AC a P, CY ponechá tak jak je (v tom se liší od ADI 1 / SUI 1)

INX, DCX

INR a DCR pracují s osmibitovým registrem, INX a DCX pracují s dvojicí registrů, jako by byl jeden šestnáctibitový. Parametrem může být B, D, H nebo SP.

DAD

Sčítání v šestnácti bitech. K obsahu registrů HL přičte BC, DE, HL nebo SP.

A jak se tedy násobí a dělí? Dostaneme se k tomu, nebojte, ale teď si odskočíme. Jump!

6.9 Skoky

Kdyby nebyly skoky, tak by program prostě jen jel, jel, jel… a na konci by jel zase od začátku. Něco jako flašinet.

Předpokládám, že znáte nějaký vyšší jazyk. Pravděpodobně ten jazyk obsahuje příkaz, kterému se vyhýbáte jako čert kříži, i když tam z nějakého důvodu je. Pokud jste mladší ročníky a začali jste s nějakým moderním jazykem, tak před vámi možná dokonce existenci toho příkazu úplně zatajili. Každopádně vám řekli, že to je zlo, nesystémovost, vede to ke špatným věcem a vůbec – pokaždé, když použijete GOTO, tak Bůh zabije koťátko.

Vítejte ve světě, kde je GOTO tím hlavním nástrojem.

Totiž, aby bylo jasno: V assembleru nejsou objekty, třídy, metody… Dokonce tam nejsou ani řídicí struktury. Není tam cyklus FOR, WHILE, není ani IF / ELSE. Ba co hůř – nejsou ani datové typy. Zapomeňte na řetězce, zapomeňte na datové struktury, jediné, co máte, je číslo. Zapomeňte na lokální proměnné. Nic z toho v assembleru nedostanete. Je to tvrdý svět – člověk proti křemíku. Nic vám neodpustí, nic za vás nedomyslí. Ale zase má jiné výhody: můžete program optimalizovat do nejmenších podrobností a detailů a za týden práce máte vymazlený program, co byste ve vyšším jazyce psali jedno odpoledne. Možná je rychlejší, možná je kratší, každopádně v něm můžou číhat velmi zákeřné chyby, které nemáte ve vyšším jazyce šanci udělat.

„Ale jak tedy udělám smyčku? Jak cyklus? Jak vnořím podmínky? Vždyť bez těchto základních konstrukcí nelze programovat!" To si povíme za chvíli, teď si ukážeme, co máme k dispozici.

JMP

To je ono, to zlé GOTO. Tady se jmenuje JMP (jako že „jump") a má jeden parametr. Tím je šestnáctibitové číslo – adresa, na kterou se skočí, tj. ze které bude procesor číst další instrukci. Vnitřně funguje tak, že to šestnáctibitové číslo, které je součástí instrukce, uloží do registru PC. Ten obsahuje adresu, ze které se čte aktuální instrukce. Když tedy do PC uložíme číslo 0123h, bude se další instrukce číst z adresy 0123h.

Když takový program píšete, musíte znát adresu, kde bude ta cílová instrukce uložena. Naštěstí si to nemusíte počítat, to za vás dělá překladač. Vy jen překladači řeknete, kde bude ležet první instrukce (.org, vzpomínáte?), a zbytek adres si dopočítá za vás. Vy si můžete jednotlivé body pojmenovat symbolickými názvy a překladač je za vás správně vyhodnotí. Nějak takhle:

```
8000                    .ORG 8000h
8000 3E 01      COLD:   MVI a,1
8002 3C         LOOP:   INR a
8003 C3 07 80           JMP skip

8006 3D                 DCR a
8007            SKIP:
8007 C3 02 80           JMP loop
```

Což je samo sebou zcela smyšlený příklad, ve skutečnosti by ho takhle nikdo nenapsal, ale je hezky vidět, jak instrukce JMP skáčou dopředu i dozadu. Zkuste si ho chvíli krokovat a uvidíte…

CALL, RET

Víte, nebyl jsem tak úplně upřímný. GOTO není jediné, co máte k dispozici. Ještě můžete použít GOSUB / RETURN. Pokud znáte BASIC, víte, co tyto instrukce dělají – volají podprogram. Ve vyšších jazycích je to *něco jako procedura / funkce*, ale opravdu jen velmi vzdáleně: nejsou žádné lokální

proměnné, nejsou parametry, nejsou návratové hodnoty. Je tedy konvence, která říká, že podprogram, pokud něco vrací, tak by to měl vracet v registru A nebo HL, ale není to vůbec nezbytné – když na to přijde, tak váš podprogram může vrátit údaje v registru E, B a návdavkem ještě v příznaku CY.

Pokud v podprogramu změníte hodnotu registru, zůstane změněná i po návratu do hlavního programu. Musíte s tím počítat, a pokud si nechcete přepisovat hodnoty v registrech, tak si nezapomeňte na začátku podprogramu potřebné hodnoty uložit na zásobník (PUSH) a před návratem zase ze zásobníku vybrat (POP). Ale pozor! Vždycky musíte vybrat přesně tolik údajů, kolik jste jich uložili, protože jinak se stanou strašlivé věci. Jaké? Hned si ukážeme.

Jak taková instrukce CALL funguje? CALL má jeden parametr, kterým je šestnáctibitová adresa, jako u instrukce JMP. CALL nejprve vezme adresu následující instrukce (říká se jí „návratová adresa"), tu uloží na zásobník, a pak provede skok na zadanou adresu, stejně jako JMP. Instrukce RET vezme hodnotu ze zásobníku (měla by tam být ta návratová adresa, kterou tam uložila instrukce CALL) a skočí na ni. provádění programu tak pokračuje za tou instrukcí CALL, která volala podprogram.

Pokud si v podprogramu uložíte obsah registru (třeba HL pomocí PUSH H), bude na zásobníku HL a pod ním návratová adresa. Před návratem tedy musíte HL zase odebrat (nejčastěji pomocí POP něco), aby se RET vrátila tam, kam má. Pokud byste hodnotu neodebrali, RET by obsah zásobníku, tj. uloženou hodnotu registrů HL, považovala za návratovou adresu a skočila by tam. Naopak pokud byste si ze zásobníku vyzvedli víc údajů, než jste do něj vložili, byla by mezi nimi i návratová adresa, takže RET by se vrátila úplně někam jinam.

Ano, OBČAS se takové triky dělají, ale upozorňuju dopředu: dělají je programátoři, kteří vědí, co dělají a proč. Mají své místo ve chvíli, kdy hledáte, jak ušetřit nějaký ten bajt / nějaký ten takt procesoru, ale v naprosté většině případů je přebývající POP nebo PUSH chyba, která povede k pádu celého systému.

```
8000                    .ORG 8000h
8000 31 60 80           LXI sp,8060h
8003             LOOP:
8003 CD 0C 80           CALL neco
8006 CD 0C 80           CALL neco
8009 C3 03 80           JMP loop
800C 3C          NECO:  INR a
800D 05                 DCR b
800E C9                 RET
```

PCHL

Prosím, nečíst P-Ch-L (podle vzoru *Pche*!), ale P-C-H-L. Když to přečtete správně, napoví vám, jak funguje. Do registru PC uloží obsah HL (PC=HL). Je to tedy něco jako JMP na adresu, uloženou v HL.

RST

RST je taková zkratka pro CALL na některé oblíbené adresy. Instrukcí RST je 8, číslovaných 0-7, a skáčou na adresu 8*N (kde N je to číslo). RST 0 je tedy totéž co CALL 0, RST 1 je CALL 8, RST 2 je CALL 16 (CALL 0010h) a tak dál… až RST 7 je totéž jako CALL 0038h. Oproti instrukci CALL, která zabírá 3 bajty (1 bajt instrukční kód 2 bajty adresa) má instrukce RST jen 1 bajt, navíc se provede rychleji než CALL. V mnoha systémech jsou proto na těchto adresách připravené často používané podprogramy. Namátkou ZX Spectrum používá RST 2 (v mnemonice procesoru Z80 označená jako RST 10h) pro vypsání znaku s ASCII kódem, který je v registru A, na výstup (obrazovku).

V Monitoru V3 slouží RST 0 pro návrat do monitoru, RST 1 pošle znak z registru A na sériovou linku, RST 2 čeká na znak ze sériové linky a vrátí ho v registru A, RST 3 jsou obecná systémová volání a RST 4 implementuje mechanismus ladicích bodů (breakpoint).

Podmíněné skoky

Je hezké, že program skáče jako srnka, ale mnohem lepší a užitečnější by bylo, kdyby mohl nějak reagovat na to, co se počítá. K tomu slouží podmíněné instrukce Jcond, Ccond a Rcond. „Cond" je označení podmínky, která se má testovat. Vždy se testuje hodnota některého příznaku S, Z, CY nebo P.

COND	PROVEDE SE, POKUD…
C	CY = 1
NC	CY = 0
Z	Z = 1
NZ	Z = 0
M	S = 1
P	S = 0
PE	P = 1
PO	P = 0

Podmínky mají i své mnemotechnické názvy: Carry, Not Carry, Zero, Not Zero, Minus, Plus, Parity Even, Parity Odd.

Podmíněné skoky, odvozené od JMP, jsou tedy JC, JNC, JZ, JNZ, JM, JP, JPE a JPO. Pracují tak, že otestují příslušný podmínkový bit. Pokud platí podmínka, provede se skok, pokud není splněna, pokračuje se dál. Například instrukce JZ 0123h zkontroluje nejprve, jestli je příznak Z=1. Pokud ano, skočí na adresu 0123h, pokud ne, pokračuje dál.

Ano, takhle nějak funguje IF cond THEN GOTO x. „Pokud je splněno, skoč tam a tam." V praxi budete chtít často zapsat něco jako „Pokud je výsledek 0, udělej ještě to a to…" (tedy chování obdobné známé konstrukci IF) V assembleru to vyřešíte tak, že zapíšete „Pokud výsledek není 0, tak přeskoč až za blok příkazů".

```
; IF (CY) {INC A, DEC D}
  JNC navesti ; Pokud podmínka neplatí, přeskoč
  INC A       ; blok příkazů
  DEC D
navesti:      ; a tady se pokračuje dál.

; IF (NZ) {INC A, DEC D} else {DEC A, DEC E}

  JZ else     ; Pokud podmínka neplatí, přeskoč
  INC A       ; blok příkazů
  DEC D
  JMP endif   ; bez toho by se provedl i blok ELSE
else:
  DEC A
  DEC E
endif:        ; a tady se pokračuje dál.
```

Takto tedy funguje konstrukce IF – THEN a IF – THEN – ELSE.

Podobně jako jsou podmíněné instrukce skoku JMP, tak můžeme vytvořit podmíněné skoky do podprogramu (Ccond – CZ, CNZ, CC, CNC, CM, CP, CPE, CPO) a podmíněné návraty (RZ, RNZ, RC, RNC, RM, RP, RPE, RPO). Fungují stejně jako CALL a RET, ale provedou se pouze pokud je splněná podmínka.

While

Dobře, ukážeme si jednu konstrukci z vyššího jazyka, přepsanou do assembleru. Představte si takový počítaný cyklus WHILE:

```
while (--B) {
   A = ctiKlav();
   pisObr(A+1);
}
```

Tedy dokud je hodnota B vyšší než 0, tak od něj odečti 1, zavolej funkci ctiKlav, výsledek ulož do A, a zavolej funkci pisObr s parametrem v A.

Řekněme si na rovinu – nic takového jako „výsledek ulož do registru A" nemáme. Je na autorovi podprogramu ctiKlav, aby vrátil požadované v registru A. Stejně tak nemůžeme podprogramu

pisObr předávat nějaké parametry – pokud podprogram očekává nějakou hodnotu v registru A, je na nás, abychom ji do toho registru připravili. Nemluvě o tom, že budeme používat registr B jako počítadlo průchodů, takže pokud nám ho některý z podprogramů přepíše, tak algoritmus nebude fungovat.

Proto je, a to vám kladu na srdce, dobrým zvykem u každého podprogramu napsat do dokumentace, ve kterém registru očekává parametr, ve kterém vrací hodnotu, a zmínit, pokud nějaký registr přepíše, ať s tím může ten, kdo ten podprogram použije, správně naložit.

(Kdysi jsem četl historku o tom, jak se kdosi v 80. letech bál použít podprogramy monitoru, protože v dokumentaci u nich psali poznámky jako: *Ničí registry D, E*. Čímž autor samo sebou myslel, že přepíše hodnoty v nich uložené, ovšem nebohý juniorní programátor se bál toho výrazu *ničí* – počítač je tak drahý stroj, tak si přeci nevezme na svědomí, aby se vevnitř něco zničilo.)

Tak jak tedy to výše zmíněné přepsat do assembleru? Řešení:

```
while:
    DCR B          ; --B
    JZ endwhile    ; je už 0? Skok na konec…

    CALL ctiKlav   ; volání
    INR A          ; A = A + 1
    CALL pisObr    ; volání
    JMP while      ; další průchod smyčkou
endwhile:
```

Budete používat nejrůznější varianty této konstrukce – s podmínkou na konci (do {…} while()) nebo s podmínkou uprostřed, ale vždy to bude plus mínus něco podobného.

Ony totiž všechny ty konstrukce, které znáte a máte rádi (for, while apod.), jsou nakonec často přeloženy do strojového jazyka, a to nějakým způsobem, který je velmi podobný tomuto. Tedy samé IF…THEN GOTO a GOTO.

GOTO!

6.10 Rotace

„Cože? Rotace? Proč ne něco Opravdu Důležitého?“ – ale ony jsou docela důležité, věřte mi, a když je správně použijete, tak ušetří spoustu času.

Procesor 8080 oplývá sadou instrukcí, která dokáže provádět takzvané bitové rotace registru A. Co to znamená?

Představme si registr A jako osm bitů, očíslovaných (od nejnižšího) 0-7. Představme si je napsané vedle sebe.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0

To je on. S ním budeme teď pracovat a jako první si ukážeme operaci „rotace vlevo“:

RLC

Instrukce RLC posune bity o 1 doleva. To znamená, že bit 0 se přesune na pozici 1, bit 1 na pozici 2, bit 2 na pozici 3 a tak dál, a bit 7, který nám vypadne zleva ven, se zase vrátí zprava na pozici 0. (A právě proto, že takhle „krouží“, se té operaci říká „rotace“. Kdyby vypadl a byl zahozen, byl by to „posuv“ – ale takové instrukce 8080 nemá). Bit 7 je zároveň zkopírován do příznaku CY.

Operace	**Pozice v registru A**								**Příznak**
	7	**6**	**5**	**4**	**3**	**2**	**1**	**0**	**CY**
Výchozí stav	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	CY
RLC	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Bit 7	Bit 7

RRC

Instrukce RRC posune bity o 1 doprava. To znamená, že bit 7 se přesune na pozici 6, bit 6 na pozici 5, bit 5 na pozici 4 a tak dál, a bit 0, který tentokrát vypadne vpravo, se vrátí zleva na pozici 7, a návdavkem je zkopírován do příznaku CY.

Operace	**Pozice v registru A**								**Příznak**
	7	**6**	**5**	**4**	**3**	**2**	**1**	**0**	**CY**
Výchozí stav	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	CY
RRC	Bit 0	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0

RAL

Další dvě instrukce berou v úvahu i obsah příznaku CY. RAL funguje stejně jako RLC, ale s jedním rozdílem – bit 7, který zleva vypadne, je zapsán do příznaku CY, a hodnota z CY je přesunuta do pozice 0 v registru A. Graficky to vypadá nějak takto:

Operace	Pozice v registru A								Příznak
	7	6	5	4	3	2	1	0	CY
Výchozí stav	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	CY
RAL	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	CY	Bit 7

RAR

RAR je obdoba předchozí funkce RAL, jen směr je opačný – bity se posouvají doprava, bit 0 jde do CY a „staré CY“ jde na pozici 7.

Operace	Pozice v registru A								Příznak
	7	6	5	4	3	2	1	0	CY
Výchozí stav	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	CY
RAR	CY	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0

Víc rotací procesor 8080 nemá.

Jaké mají tyto operace smysl? Když se nad tím zamyslíte, je to analogické jako u desítkové soustavy: když máte číslo (123) a posunete ho o jednu pozici doleva (a zprava doplníte nulou), dostanete jeho desetinásobek (1230). Když ho posunete doprava, dostanete celočíselný výsledek dělení deseti. U dvojkové soustavy platí totéž – posun doleva je totéž jako vynásobit dvěma, posun doprava je dělení dvěma.

Trochu tam hapruje to rotování kolem dokola, ale to lze obejít pomocí vhodného domaskování nebo tím, že si předem nastavíte příznak CY na požadovanou hodnotu.

Dají se použít ještě na spoustě dalších míst – ale to si ještě ukážeme.

STC, CMC a jak nastavit příznak CY?

Přiznám se, že jsem váhal, kam zařadit instrukce STC a CMC, a nakonec je zařadím sem. Instrukce STC nastaví CY na 1, instrukce CMC neguje jeho hodnotu. Pokud chcete CY nastavit na nulu, musíte buď udělat STC a CMC, nebo (pokud vám nevadí, že přijdete o hodnotu v přízna-

cích S, Z, P) zkusit třeba CMP A – tato instrukce porovná registr A se sebou samotným. Výsledek je, nepřekvapivě, „hodnota se sobě rovná“, takže instrukce nastaví Z na 1 a S a CY nuluje.

Rotace 8080 – souhrn

Operace	Pozice v registru A								Příznak
	7	6	5	4	3	2	1	0	CY
Výchozí stav	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	CY
RLC	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Bit 7	Bit 7
RRC	Bit 0	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
RAL	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	CY	Bit 7
RAR	CY	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0

6.11 Násobení

Když se sejdou tři sobi, tak se radost násobí…

Už jsme si řekli, že násobení nepatří mezi základní operace, kterými procesor 8080 oplývá. A rovnou můžu prozradit, že to ani u Z80, ani u 6502 není v tomto směru o moc lepší. Po pravdě si nevzpomínám kromě 6809 a jednočipů 8051 na žádný další osmibitový procesor z té doby, který by měl instrukci pro násobení. Neříkám, že neexistoval, jen si na něj nevzpomínám.

Pokud chceme násobit, a taková potřeba na každého někdy přijde, musíme si vystačit s tím co máme a známe. A taky trošku zavzpomínat na základní školu.

Násobení poprvé – naivní

Víme, že 3 krát 5 je vlastně 5 + 5 + 5. Třikrát se sečte pětka. Mohlo by to vypadat nějak takhle – násobíme obsah registru B obsahem registru C a výsledek ukládáme v A:

```
8000                    .ORG 8000h
8000 0E 05              MVI C,5
8002 06 03              MVI B,3
8004 3E 00              MVI A,0
8006 81         LOOP:   ADD C
8007 05                 DCR B
8008 C2 06 80           JNZ loop
```

Rovnou upozorňuju, že to je jeden z nejzoufalejších kódů, co tu budou zveřejněné. Je špatný z mnoha důvodů. Zaprvé – když násobím dvě osmibitová čísla, může být výsledek až šestnáctibitový (prostý test: 255 * 255 = 65025 (FE01h)). Já tu sčítám hodnoty v registru A, který je osmibitový a přeteče už u čísla 255. Což, to by se dalo napravit snadno – místo sčítání v registru A můžu použít registry HL, přičítat k nim číslo z registrů DE a bude to… Nějak takto (opět násobíme B * C, výsledek jde do HL):

```
8000                    .ORG 8000h
8000 0E 05              MVI C,5
8002 06 03              MVI B,3
8004 21 00 00           LXI H,0
8007 54                 MOV D,H
8008 59                 MOV E,C
8009 19         LOOP:    DAD D
800A 05                 DCR B
800B C2 09 80           JNZ loop
```

Teď už můžeme násobit v plném rozsahu, tenhle problém jsme vyřešili, ale je tu jiný problém. Totiž ten, že násobení trvá dlouho. Představte si, že B bude třeba 250 a C dvě. Budeme 250krát přičítat dvojku, stále dokola – ostatně zkuste si to v emulátoru nasimulovat, uvidíte sami. Co s tím?

Můžeme čísla na začátku prohodit (místo „250x přičti dvojku“ udělat „2x přičti 250“), ale to je řešení na půli cesty. Vlastně ne *na půli cesty*, ve skutečnosti *na pytel*! Správný postup je zahodit tenhle algoritmus a jít na to jinak.

Násobení podruhé – rafinované

Spousta lidí tváří v tvář dvojkové soustavě strne a očekává, že přestanou fungovat postupy, které znají ze soustavy desítkové. Ale ony fungují docela dobře. Třeba to násobení se dá obstojně zařídit stejně, jako jsme se to učili ve třetí třídě ZŠ, když jsme násobili na papíře dvojciferná čísla pod sebou. Pamatujete?

```
    26
*   34
------
   104   (4 * 26)
+  78    (3 * 26)
======
   884
```

Postup popíšu slovy. Začneme od nejnižšího řádu druhého činitele (4). Tím vynásobíme prvního činitele (4*26) a dostaneme první mezivýsledek (104). Ten platí pro nejnižší řád (je zapsaný úplně

vpravo, zarovnaný s tím řádem, který právě zpracovávám). Pokračuju k vyššímu řádu druhého činitele. Vida, je to 3. Postup opakuju: vynásobím tím číslem první činitel (3*26) a druhý mezivýsledek (78) si zapíšu na pozici desítek, tedy o řád výš než předchozí. A takhle budu pokračovat, dokud nepronásobím všechny řády druhého činitele. Nakonec mezivýsledky sečtu.

Tak. A jde to tak i ve dvojkové soustavě? Zkusíme to. Kolik je dvojkově 13 * 9?

```
     1101  13
0    1001   9
---------
     1101  1 * 1101
+   0000   0 * 1101
+  0000    0 * 1101
+ 1101     1 * 1101
=========
  1110101  117
```

Postup je naprosto stejný – taky násobíme a posouváme o řád, ovšem situaci máme o mnoho jednodušší: Protože pracujeme ve dvojkové soustavě, tak násobíme vždy jen buď jedničkou (tedy mezisoučet je roven prvnímu činiteli), nebo nulou (a mezisoučet je 0). Mezisoučty postupně posouváme doleva a přičítáme.

Tak, s touhle vědomostí by určitě šlo napsat násobící algoritmus, který nemá tu nectnost, že by závisel na hodnotě činitele. Jeho hlavní smyčka proběhne tolikrát, kolik má druhý činitel bitů. Ale předtím, než se do něj pustíme, si řekneme ještě pár tipů.

K posunu registru o jeden bit doleva můžeme samozřejmě použít instrukci rotace. Ovšem tady budeme rotovat rovnou dva registry najednou. Nejjednodušší způsob bude použít instrukci DAD H. Ta, jestli si vzpomínáte, přičte k registrovému páru HL hodnotu svého parametru – tedy HL. Sečíst HL a HL znamená vlastně udělat 2*HL, a z hlediska bitového je to totéž jako posunout obsah registru doleva o 1 bit a zprava doplnit nulou. Tedy přesně to, co se děje v tom algoritmu výše.

Troufnete si napsat algoritmus, který vynásobí obsah registrů D a E a výsledek uloží do HL? Zkuste to…

Řešení je následující:

```
8000                    .ORG 8000h
8000 31 80 80           LXI sp,8080h
8003 16 0D              MVI d,13
8005 1E 09              MVI e,9
8007 CD 0D 80           CALL mul8
```

```
800A C3 00 00         JMP 0
800D           MUL8:
800D 21 00 00         LXI h,0
8010 06 08            MVI b,8
8012 4A               MOV c,d
8013 54               MOV d,h
8014           MUL8LOOP:
8014 79               MOV a,c
8015 0F               RRC
8016 4F               MOV c,a
8017 D2 1B 80         JNC Mul8Skip
801A 19               DAD d
801B           MUL8SKIP:
801B EB               XCHG
801C 29               DAD h
801D EB               XCHG

801E 05               DCR b
801F C2 14 80         JNZ Mul8Loop
8022 C9               RET
```

Může vám připadat trošku zmatené, ale nebojte, hned si to projdeme.

Na začátku nastavím zásobník na rozumnou pozici. Připravím si do registrů D a E hodnoty, které budeme násobit (13 a 9). Mohl bych použít LXI D, 0D09h, ale pro názornost je to takhle rozepsané. Pak volám podprogram Mul8 – jako že „Multiplication, 8 bits".

V podprogramu si nejdřív vynuluju registry HL. Tam se budou průběžně sčítat mezivýsledky. Do B si připravím hodnotu 8 – to je počítadlo, kolikrát provedu hlavní smyčku. Budu ji provádět osmkrát, pro každý řád jednou, tak proto ta osmička. Teď bych potřeboval, aby v DE byl jen druhý činitel (tedy v D 0 a v E to, co tam je). Přesunu si tedy obsah D do registru C (mám teď činitele v registrech C a E) a registr D si vynuluju. A teď může začít vlastní kouzlo…

Podívám se, jakou hodnotu má nejnižší bit registru C, a posunu si ho o jednu pozici doprava (v příštím průchodu tak budu kontrolovat pozici 1, pak pozici 2 atd.) To ale nemůžu udělat přímo, takže si jej musím nejprve přesunout do registru A, tam provést RRC a pak uložit zpátky do C. Instrukce RRC dělá obě věci, co potřebuju, najednou – uloží nejnižší bit (do příznaku CY) a rotuje o jednu pozici doprava.

Podle hodnoty nejnižšího bitu (mám ho teď v CY) se rozhodnu, jestli mezisoučet (DE) přičtu k celkovému výsledku (pokud je 1), nebo nepřičtu (CY=0). Použiju podmíněný skok: Pokud není CY (JNC), tak skoč dál. Pokud je CY, pokračuj a přičti DE k HL (HL=HL*DE)

Tím mám jeden krok skoro za sebou. Teď už jen musím posunout mezisoučet v DE o jednu pozici doleva. Vyřeším to tak, že ho přičtu k sobě samotnému (tedy DE = DE + DE). A protože na to instrukci nemáme, tak si pomocí XCHG na chvíli prohodíme DE a HL a provedeme HL = HL + HL.

Pak už jen snížím počet průchodů smyčkou (DCR B), a pokud ještě nejsme na nule, tak to celé provedeme znovu.

A věřte nebo ne, na tomhle principu je založena naprostá většina softwarových násobiček.

Násobení dvaapůlté – optimalizované

Můžeme u předchozího algoritmu ušetřit pár bajtů, dva registry a nějaký ten čas, a to tím, že druhý činitel umístím do registru H, zatímco mezisoučet bude neustále v HL. Zní to divně? Divné to teprve začne být!

Ve skutečnosti při tomto postupu nepůjdeme od nejmenšího bitu, ale od nejvyššího. Náš postup nebude ten, že při každém průchodu přičteme první činitel, patřičně posunutý doleva (*2), ale při každém průchodu naopak vynásobíme mezisoučet dvěma a případně přičteme první činitel. Matematicky místo $(A * 2^{0*} b_0) + (A * 2^{1*} b_1) + (A* 2^{2*}b_2) + \ldots$ (A je první činitel, b_0-b_7 bity druhého činitele) budeme provádět $(((A*b_7)*2+(A*b_6))*2+(A*b_5))*2\ldots$

Pro zájemce doporučím nastudovat něco, čemu se říká Hornerovo schéma. *(Náš učitel programování tomu říkal tvrdošíjně „Hornerovo šéma“ – ale on říkal taky místo while cosi, co znělo jako [hwajl]…)*

Využijeme toho, jak se chová instrukce DAD H. Už jsme si řekli, že udělá HL = HL + HL, tedy HL * 2. Tedy vlastně posune obsah HL o jednu pozici doleva. Zároveň nastaví příznak CY, pokud hodnota přeteče – v tomto případě přeteče, pokud je nejvyšší bit registru H = 1… takže jako by zkopíroval ten nejvyšší bit do příznaku CY.

Takže jednou instrukcí tu máme:

- posun hodnoty v registru H o jeden bit doleva (tedy můžeme jít bit po bitu od b_7 po b_0),
- při každém posunu udělá vlastně jeden bit prostoru pro mezisoučet v HL (to, že H v průběhu výpočtu ve vyšších bitech obsahuje nějaký zmatek, to nevadí, protože „zmatek“ se postupem času odsune pryč),
- vynásobí mezisoučet dvěma.

Tolik dobra za jednu cenu, že?

```
8000                    .ORG 8000h
8000 31 80 80           LXI sp,8080h
8003 16 13              MVI d,0x13
8005 1E 09              MVI e,0x9
8007 CD 0D 80           CALL mul8
800A C3 00 00           JMP 0
800D            MUL8:
800D 62                 MOV h,d
800E 2E 00              MVI l,0
8010 06 08              MVI b,8
8012 55                 MOV d,l
8013            MUL8LOOP:
8013 29                 DAD h
8014 D2 18 80           JNC Mul8Skip
8017 19                 DAD d
8018            MUL8SKIP:

8018 05                 DCR b
8019 C2 13 80           JNZ Mul8Loop
801C C9                 RET
```

Vyšší dívčí? Nebojte se, vpravíte se do toho. V očích vám teď vidím otázku: *Zešílels? Tohle přeci není normální!*

Takže: *Ano, zešílel, ale to s tím nijak nesouvisí, a ne, v assembleru je tohle naprosto normální.* Když chcete, hrajete o každý bit v registru, takt procesoru, bajt v paměti… A někdy ani nechcete, ale musíte!

Násobení potřetí

V assembleru má optimalizace vždycky dvě fáze. V té první vyhazujete instrukce, které jste napsali zbytečně – program bude kratší i rychlejší. V té druhé si musíte vybrat: Chcete kratší kód? Musíte dělat triky, co vás budou stát nějaký ten čas. Chcete kód rychlý? Musíte obětovat prostor. I u algoritmu násobení stojíte před stejným rozhodováním: rychleji? Bude to něco stát…

Úplně nejrychlejší algoritmus by byl takový, který by měl v paměti výsledky všech možných kombinací pro násobení (v případě osmibitových činitelů by jich bylo 65536) a prostě by si jen sáhnul tam, kam potřebuje. Taková tabulka by ale zabrala spoustu paměti a pro vlastní program by v adresním prostoru běžného osmibitového procesoru zůstalo… ehm… přesně 0 bajtů. Naštěstí existují kompromisní řešení, kdy si do tabulky uložíme jen pár konstant (třeba 1024) a zbytek dopočítáme. Je to opět kompromis mezi rychlostí a velikostí.

6.12 Logické instrukce

– „Milý autore, tvá kniha je zmatená a nemá žádnou logiku! Logika v něm chybí! Tvůj čtenář."

– „Milý čtenáři, logika tu nechybí. Logika je tu právě teď! Tvůj autor."

Protože předpokládám, že všichni, co to čtete, jste už nějaký programovací jazyk zvládli, tak si nemusíme, doufám, vysvětlovat, co je to AND, OR a XOR. Nebo aspoň naznačím: &, | a ^.

ANA, ANI

Provede operaci AND mezi registry A a vybraným registrem (ANA), nebo mezi registrem A a zadanou osmibitovou konstantou (ANI). Hezky bit po bitu, a výsledek jde zpátky do A. Pokud se na parametr budeme dívat jako na masku, kterou je třeba aplikovat na registr A, tak vězte, že kde má maska 1, tam zůstane A nedotčeno, kde má 0, tam bude vynulováno. Instrukce ovlivňují všechny příznaky.

ORA, ORI

Provede operaci OR mezi registry A a vybraným registrem (ORA), nebo mezi registrem A a zadanou osmibitovou konstantou (ORI). Hezky bit po bitu, a výsledek jde zpátky do A. Pokud se na parametr budeme dívat jako na masku, kterou je třeba aplikovat na registr A, tak vězte, že kde má maska 0, tam zůstane A nedotčeno, kde má 1, tam bude nastaveno na 1. Instrukce ovlivňují všechny příznaky.

XRA, XRI

Provede operaci XOR mezi registry A a vybraným registrem (XRA), nebo mezi registrem A a zadanou osmibitovou konstantou (XRI). Hezky bit po bitu, a výsledek jde zpátky do A. Pokud se na parametr budeme dívat jako na masku, která se bude aplikovat na registr A, tak vězte, že kde má maska 0, tam zůstane A nedotčeno, kde má 1, tam bude změněna hodnota příslušného bitu v A. Instrukce ovlivňují všechny příznaky.

CMA

Vezme obsah registru A a zneguje ho, tj. všechny 1 prohodí za 0 (a obráceně). Instrukce nijak neovlivní příznaky.

Praktické použití

Takhle odsud to možná vypadá jako nějaká zajímavost pro nerdy, ale věřte, že v assembleru budete tyhle instrukce používat často. Například pro nastavení konkrétního bitu použijete ORI 01h (02h, 04h, 08h, 10h, 20h, 40h, 80h). Pro vynulování bitu použijete AND, ale v negované podobě – tam, kde chcete bit nastavit na 0, nechte 0, všude jinde 1. Chcete-li vynulovat bit 3 (To je on: „….3…"), použijete konstantu, která má všude 1, s výjimkou pozice 3: „11110111". Tedy F7h.

Operace XOR slouží krom změny hodnot bitů i k jednomu hezkému fíglu: pokud potřebujete vynulovat registr A a nezáleží vám na tom, že se změní hodnota příznaků, použijte XRA A. Taková instrukce zabere jen jeden bajt a je i proti MVI A,0 rychlejší.

Instrukce CMA má efekt stejný jako XRI FFh – ale nemění příznaky a zabere jen jeden bajt.

Teď jsme se dostali do bodu, kdy zbývá probrat už jen šest instrukcí procesoru 8080, takže je opět vhodná chvíle na drobné odbočení, ať to máme pestřejší…

6.13 Instrukce a operační kód

… a když na vás o půlnoci zakřičím: „C3h!“, tak vy vyskočíte!

Jistě jste se v předchozích kapitolách dívali na to, jaké kódy patří k jakým instrukcím a zaujalo vás, že „MVI A, něco“ má vždycky první bajt 3Eh. (*Cože? Nevšimli? Nezaujalo? A zvážili jste, že byste se místo assembleru učili třeba Javu?*)

Nebo taková instrukce MOV – ať už je jakákoli, je její kód vždycky mezi 40h a 7Fh. Má to svoji logiku. Schválně, podívejte se sami (ten program je naprosto nesmyslný, proto se nedívejte, co dělá, ale jak je přeložený!)

```
0000 40              MOV b,b
0001 41              MOV b,c
0002 42              MOV b,d
0003 43              MOV b,e
0004 44              MOV b,h
0005 45              MOV b,l
0006 46              MOV b,m
0007 47              MOV b,a
0008 48              MOV c,b
0009 50              MOV d,b
000A 58              MOV e,b
000B 60              MOV h,b
000C 68              MOV l,b
000D 70              MOV m,b
000E 78              MOV a,b
```

Dá se v tom vypozorovat určitá pravidelnost, že? Instrukce MOV obecně vypadá totiž po jednotlivých bitech takto:

0 1 d d d s s s

Trojice bitů D a S kódují jednotlivé registry (D = cílový, S = zdrojový). Registry jsou kódované následujícím způsobem:

R2	R1	R0	REGISTR
0	0	0	B
0	0	1	C
0	1	0	D
0	1	1	E
1	0	0	H
1	0	1	L
1	1	0	M
1	1	1	A

Dva nejvyšší bity jsou 0 a 1. Pokud má instrukce nejvyšší dva bity takto nastavené, ví dekodér instrukcí, že jde o MOV, že se bude přesouvat z registru do registru, a informaci o tom, ze kterého do kterého najde v bitech 5-3, resp. 2-0. (Ano, je tu výjimka – přesun z registru M do registru M není možný a operační kód, který by taková instrukce měla (76h) je vyhrazen pro jinou instrukci.)

Co třeba instrukce s takovýmto bitovým obrazem?

0 0 d d d 1 1 0

Možné kódy jsou 06h, 0Eh, 16h, …, 3Eh. Tato bitová kombinace říká dekodéru, že se jedná o instrukci MVI, tedy přesun následujícího bajtu do některého z registrů. Který registr to bude, to je opět zakódované v operačním kódu – jsou to ty tři bity „d“.

A co takováto instrukce?

0 0 d d d 0 1 0

Možné kódy jsou 04h, 0Ch, 14h, …, 3Ch. Tentokrát se jedná o instrukci INR a v operačním kódu je opět určeno, o jaký registr půjde.

Když se podíváte do tabulky instrukcí, zjistíte, že instrukce nejsou rozmístěny nahodile. V první čtvrtině (00-3Fh) jsou nejrůznější přesuny, rotace, inkrementace a dekrementace. Druhá čtvrtina

(40h-7Fh) patří přesunům (MOV), třetí třetina (80h-BFh) obsahuje aritmetické instrukce, které pracují s registry (ADD, ADC, …) a poslední čtvrtina (C0h-FFh) obsahuje převážně skoky, operace s přímým operandem a operace se zásobníkem.

Takováto výstavba instrukcí je dána potřebou mít co nejjednodušší dekodér. Později, až se podíváme na operační kódy procesorů 6502 nebo 680x, zjistíme, že jsou v dodržování tohoto principu mnohem důslednější a že i přes své zdánlivé limity nabízejí programátorovi mnohem větší svobodu.

6.14 Pseudoinstrukce

A jestli vám vadí to „pseudo“, nemáte rádi věci, co jsou jen „jako“ a raději chcete mít vše jasně nařízené, tak to nazývejte třeba „direktivy“.

Překladač (správně nazývaný assembler, ale už jsme si vysvětlili, že tohle slovo se tak nějak přemigrovalo i na Jazyk Symbolických Adres, a já tentokrát potřebuju zdůraznit, že mám na mysli ne ten jazyk, ale opravdu překladač) plní hlavně dva úkoly:

1. Překládá instrukce, zapsané v mnemotechnickém tvaru, tj. v podobě nějakých snadno zapamatovatelných názvů, na jejich instrukční kódy (MOV B, C = > 41h).
2. Počítá pozici v paměti a dovoluje je nazvat nějakým lidským jménem, abychom mohli napsat „CALL podprogram“ a nemuseli přemýšlet, na jaké že adrese ten podprogram je.

Kromě těchto instrukcí („pravých“ instrukcí) pracuje překladač ještě s takzvanými pseudoinstrukcemi. „Pseudo-“ proto, že se zapisují jako instrukce, ale ve skutečnosti je procesor nezná. Zná je překladač a nějak se podle nich zachová. Některé už jsme si ukázali.

org adresa

Oznámí prohlížeči, že toto místo programu bude na dané adrese a další instrukce se uloží za něj. Díky tomu si překladač správně spočítá adresy.

db číslo [,číslo, …]

Uloží do výsledného kódu bajt (nebo bajty). Místo čísla můžeme zapsat i řetězec do uvozovek – překladač s ním naloží jako se seznamem ASCII kódů.

dw číslo [,číslo, …]

Uloží do výsledného kódu šestnáctibitové číslo jako dva bajty. Pokud je víc parametrů, uloží víc dvojbajtových hodnot.

ds počet

Řekne překladači, že má vynechat určitý počet bajtů. Používá se v případech, že chceme nechat nějaký prostor volný (např. pro buffer) a je nám jedno, jaký bude jeho obsah na začátku, stačí, když bude N bajtů dlouhý.

návěští equ hodnota

Přiřadí jménu návěští nějakou hodnotu. Třeba takto:

```
HELLO EQU 8000h
```

Od této chvíle už nemusíme řešit, jestli jsme nespletli adresu – prostě napíšeme LXI H, HELLO a překladač zjistí, že jsme HELLO nadefinovali takovouto hodnotu, tak ji použije.

Poznámka k návěštím

Každé jméno návěští je unikátní. To znamená, že pokud použijete stejné jméno znovu k označení nějaké pozice v paměti, nebo pokud už nadefinovanému jménu budete přiřazovat hodnotu pomocí EQU, překladač vám vyhodí chybové hlášení. Je to logické – kdybyste si definovali dvakrát stejný název, překladač by nevěděl, jaká adresa platí…

Následující pseudoinstrukce jsou popsané tak, jak je implementuje překladač ASM80. V jiných překladačích se jejich syntax může lišit.

.include jméno-souboru

Na místo této instrukce vloží obsah zadaného souboru, jako byste ho tam vložili pomocí copy-paste. Ale asi není potřeba vysvětlovat víc. Všimněte si tečky na začátku pseudoinstrukce… Správně by se měla psát u všech pseudoinstrukcí (je to taková docela užitečná konvence), ale z historických důvodů se u těch výše zmíněných nepoužívá.

.if výraz – .endif

Překladač vyhodnotí výraz, a pokud je nenulový, pokračuje v překladu. Pokud je nula, vynechá všechny řádky až do nejbližšího .ENDIF – obdoba podmíněných bloků třeba u preprocesoru jazyka C.

.ifn výraz – .endif

Stejná konstrukce, jen obrácená podmínka: následující instrukce až do .ENDIF se překládají, pokud je výraz nulový!

Aktuální seznam pseudoinstrukcí naleznete v dokumentaci k ASM80.

6.15 BCD a přerušení

Kód BCD (Binary Coded Decimal) je způsob zápisu desítkových čísel v šestnáctkové soustavě tak, že se místo znaků 0-F používají jen znaky 0-9 a když k 09h přičtete jedničku, nedostanete

0Ah, ale 10h. Odborné vysvětlení si můžete přečíst na Wikipedii, já se přiznám, že mi to z tamního popisu moc jasné nebylo. Vezmu to tedy „vlastními slovy".

Někdy se hodí vypsat číslice takovým způsobem, jakému rozumějí normální smrtelníci. Ti, jestli si ještě vzpomínáte, považují za číslice jen ty znaky od nuly do devítky, a kdybyste na ně vyrukovali s tvrzením, že A až F jsou taky číslice, tak vám nebudou rozumět. A ačkoli se někteří programátoři (pohříchu často vyrostlí na assembleru) domnívají, že člověk se má buď naučit rozumět počítačům tak, jak je přirozené strojům, nebo na ně nešahat a jít pracovat do marketingu, je převažující postoj opačný. Takže ani v tom assembleru nezaškodí čas od času nasimulovat starou dobrou desítkovou soustavu.

Jestli jste si zkusili převést strojově osmibitové číslo na desítkovou hodnotu, dáte mi zapravdu, že to není úplně procházka růžovou zahradou. Dokonce ani počet bitů na číslici není vždy stejný… Proto vznikl kód BCD, kde se do jednoho osmibitového čísla vejde desítkové číslo z rozsahu 0 – 99. Pokud je číslo v kódu BCD, tak 99h (1001 1001) neznamená 153 desítkově, ale 99 desítkově.

Což, jako vyjádření čísla dobré, ale teď – jak se s tím pracuje? Procesor na to má nástroj.

Představte si, že sečtete dvě čísla, řekněme 13h a 28h. To máme 3Bh (je to totiž desítkově 19 + 40 = 59). A přesně to nám spočítá následující kód:

```
MVI A, 13h
ADI 28h
```

Pokud budeme obě čísla považovat za čísla v kódu BCD, tedy vlastně třináct a dvacet osm, budeme očekávat, že výsledek bude 41 – v BCD kódu 41h. Jak tedy sečíst dvě čísla v kódu BCD? Zcela normálně: sečteme je běžnou instrukcí ADD/ADI, a po ní použijeme instrukci DAA.

DAA

Zkratka znamená „Decadic Adjust after Addition", neboli Desítková úprava po sčítání. Tahle instrukce vezme příznakové bity (zejména bit AC, který se nikde jinde nepoužívá) a pomocí nich upraví výsledek (3Bh) tak, aby byl v kódu BCD (41h).

Při odčítání platí

Požadavek na přerušení!

V procesorových systémech se občas stane něco významného a důležitého, na co je potřeba okamžitě zareagovat. Třeba někdo zmáčkne klávesu na klávesnici. (Aspoň vidíte, co je ve světě procesorů Opravdu Důležité – kdyby se náhodou vzňal a hořel, bylo by mu to jedno, ale zmáčknutý čudlík zasluhuje okamžitou reakci!) Nebo přijdou nějaké údaje ze sériového rozhraní a je třeba

je přečíst, než tam někdo pošle další a tenhle se ztratí. Nebo se stane to, že uplyne nějaký předem nadefinovaný čas – tedy něco jako minutka na vaření vajíček. V takovém případě požádá to zařízení, ve kterém nastala ona výjimečná událost, procesor o přerušení. Ten všeho nechá, odskočí si do podprogramu, který obslouží, co je potřeba, a pak se zase hezky vrátí zpátky k tomu, co měl rozdělané.

KONEC PŘERUŠENÍ!

… totéž jako při sčítání. Stačí použít instrukci DAA, která upraví výsledek tak, aby zase obsahoval číslo v kódu BCD.

Po pravdě řečeno – s instrukcí DAA se člověk moc nesetkává. Kupodivu i spousta algoritmů na převod čísel z desítkové do šestnáctkové soustavy a zpátky se bez téhle instrukce obejde. Proto je to takový docela otloukánek, většinou ho vysvětlují všichni až na konci, kromě špatných knih o programování v assembleru x86, které probírají instrukce ne podle jejich složitosti nebo funkce, ale podle abecedy. Tam se totiž ta samá instrukce jmenuje AAA! Představte si, co byste říkali, kdybych na vás jako první instrukci vytáhnul právě DAA a BCD kód!

Požadavek na přerušení!

Přerušení je něco jako telefonát – taky vás zaskočí v nejnevhodnější chvíli a vy se mu musíte hned teď věnovat. Proto je dobré, aby jeho obsluha byla co nejkratší („*Ano, pane řediteli! – Jistě pane řediteli! – Tak to jste asi magor, pane řediteli! – Ano, na personální si dojdu zítra ráno! Nashledanou!*“) Vyřídit to nezbytné, ať se můžeme vrátit k poctivé práci.

KONEC PŘERUŠENÍ!

BCD kód bych, s dovolením, považoval tímto za probraný, víc už asi nemá smysl na tomto místě probírat. Namísto toho se vypořádáme s přerušením…

DI, EI

Jak přerušení vypadá, to jste názorně mohli vidět v předchozím textu. Naštěstí existují instrukce DI (ta přerušení zakáže) a EI (ta ho opět povolí). DI použijeme ve chvíli, kdy procesor potřebuje dodělat svou práci a je to důležitější než obsloužit periferie – třeba něco časově kritického. V zásadě by mohl mít zakázané přerušení neustále (a některé systémy to tak mají), ale zase na druhou stranu, když zůstanu u alegorie s telefonem: *Nikdo se vám nedovolá!*

Teď tedy používám jedno virtuální DI, aby už další výklad žádné přerušení nerušilo, a řekneme si, jak to vlastně funguje.

Procesor má vstup INT. U 8085 má i vstupy RST5.5, RST6.5, RST7.5, které slouží k témuž. U jiných procesorů se jmenují ještě jinak (IRQ třeba), ale princip je stejný. Za normálních okolností by na takovém vstupu měla být klidová hodnota, a taky tam je. (U 8080 a vstupu INT je to 0, u vstupů RST nebo IRQ to je 1, záleží na procesoru) Pokud některá část počítače (klávesnice, sériový port, časovač, …) požaduje přerušení, nastaví příslušné obvody tento vstup na hodnotu opačnou. Procesor dokončí aktuální instrukci a zkontroluje stav tohoto vstupu. Když je klidová, pokračuje dál, pokud je ale aktivní, a zároveň je přerušení povolené, udělá následující kroky:

- Zakáže přerušení
- Potvrdí, že požadavek převzal (tj. ve stavovém slovu pošle informaci INTA – Interrupt Acknowledge)
- Přečte z datové sběrnice kód instrukce, kterou má vykonat. Je to jedna z instrukcí RST 0 – RST 7, nebo instrukce CALL.
- Načtenou instrukci pak provede.

Všimněte si, že jsem nepsal, že ji čte „z paměti“, ale „ze sběrnice“. Je na systému, aby v takovém stavu připravil procesoru tu správnou instrukci.

Asi nejjednodušší způsob, který lze dosáhnout zapojením jednoho rezistoru, je ten, že vždy, když procesor vyšle INTA a čeká na instrukci, tak mu pomocný obvod 8228 vrátí RST 7. Vzpomeňte si, co jsme si říkali o instrukci RST n: *Je to volání podprogramu na adrese 8*n*. RST 7 tedy zavolá podprogram na adrese 38h a tam by měla proběhnout obsluha přerušení.

Tuto metodu jsem zmiňoval už při představení architektury systémů s procesorem 8080.

Složitější způsoby, které využívají třeba obvod 8214, dokážou obsloužit osm různých požadavků podle jejich priorit a adekvátně k nim zavolat různé instrukce RST. Specializované řadiče přerušení (8259) dokážou například cyklicky měnit priority nebo volat ne osm pevně daných instrukcí, ale pomocí instrukce CALL zavolat libovolnou adresu.

Ale u těch jednodušších systémů je většinou vše zapojeno tak, že INT znamená vykonání instrukce RST 7.

Obslužná rutina by měla být co nejkratší a nejrychlejší. Nezapomeňte na konci zase povolit přerušení instrukcí EI, jinak obsluha proběhne jen jednou. A pokud náhodou budete psát obsluhu přerušení pro různé RST, uvědomte si, že na to máte jen osm bajtů – nejčastěji ty první tři použijete pro JMP a pět zůstane volných.

HLT

Víte, co dělá procesor v počítači většinu času? Čeká. Čeká, až mu něco řeknete, pak to chvíli zpracovává, a pak zase čeká. Mezi zmáčknutím kláves čeká. Když zmáčknete dvě klávesy za sekundu, tak za tu dobu provede 8080A rovné dva miliony taktů. Obsluha každého zmáčknutí zabere, no, ať nežeru, i třeba 500 taktů! Procesor tedy 1000 taktů pracoval a 1 999 000 taktů jen čekal, jestli se nestane něco zajímavého. Váš den, kdybyste byli procesor, by vypadal tak, že byste 43 sekund dělali něco zajímavého, a zbývajících 23 hodin 59 minut a 17 sekund by se nedělo nic. Vůbec nic. (A tohle jim děláme neustále, mimochodem! Skoro pořád na nás s něčím čekají!)

Instrukce HLT zastaví procesor. Stojí na místě, neděje se nic, žádné instrukce neprovádí, zkrátka stojí. Stojí a stojí. Nic nedělá. Šmitec, šlus, finito. Z téhle letargie ho probudí buď RESET, nebo právě požadavek na přerušení. Představte si, jak ten procesor zpracovává instrukci HLT, stojí stále na místě, jakoby v nekonečné smyčce, když v tom přijde požadavek na přerušení. Procesor udělá ty kroky, které jsem popsal, a provede (třeba) instrukci RST 7. Ta vykoná nějaké operace a vrátí se za instrukci HLT. Program tak může pokračovat. Často se tenhle fígl používá např. k tomu, že si nastavíme časovač (onu „minutku“) a procesor pomocí HLT zastavíme. Až doběhne požadovaný časový interval, přijde přerušení, a to nás vyvede ze stavu HALT.

Sledujte: HLT!

(Jo a taky je dobrý nápad si před tím, než uděláme to HLT, zkontrolovat, jestli jsme nezapomněli povolit přerušení…)

6.16 Práce s periferiemi

Pracovat s pamětí je fajn, ale pro uživatele je paměť neviditelná. Uživatel ocení třeba nějaké to písmeno na displeji, nějaký ten zvuk, chce zmáčknout tlačítko, zakvrdlat joystickem či tak něco…

Počítačové periferie jsou „všechno co není procesor nebo paměť“. Takže třeba ty klávesnice, výstupní zařízení, magnetofony, reproduktory, … Ty se nějakým způsobem připojují do systému (jakým přesně, to záleží na tvůrci systému) a procesor s nimi komunikuje.

Některé procesory (6502 například) nerozlišují periferie od paměti. Zkrátka a dobře určité adresy z adresního rozsahu neobsadí paměť, ale periferie. „Pošli znak 12 sériovou linkou“ je pro ně stejná operace jako „ulož znak 12 do paměti na konkrétní adresu“. Má to svoje výhody, ale i nevýhody.

Procesor 8080 (nebo Z80) naproti tomu mají speciální vývody, které říkají: Teď se nepřistupuje do paměti, ale ke vstupně-výstupnímu zařízení, k periferii. Paměť může zůstat neaktivní a adresa

čtení nebo zápisu se vztahuje na periferní zařízení (je pro to takové slovo „port“ – „přistupujeme na porty“ třeba). U procesoru 8080 může být až 256 periferních zařízení, protože pro tyto operace používá osmibitovou adresu.

IN, OUT

Procesor 8080 má dvě instrukce pro práci s periferiemi. IN slouží ke čtení, OUT k zápisu. Obě instrukce mají jeden přímý parametr, a tím je osmibitová adresa periferie. Instrukce IN F8h přečte bajt z periferie na adrese F8h a uloží ho do registru A. OUT 23h vezme obsah registru A a pošle ho na periferii na adresu 23h.

Víc k těmto instrukcím asi nelze říct. Jsou opravdu tak jednoduché, jak vypadají. Konkrétní adresy už jsou na tvůrci toho kterého systému.

Například v počítači PMI-80 je na adresách F8h-FBh připojen obvod 8255, který slouží jako programovatelný obvod pro řízení vstupů a výstupů. Tentýž obvod v PMD-85 sídlí na adresách F4h-F7h (a tam obsluhuje klávesnici, LED a reproduktor), další stejného typu je na adresách F8h-FBh (a tam se stará o načítání dat z ROM modulu), no a na adresách 1Eh a 1Fh je připojený obvod 8251, který pro změnu slouží pro komunikaci s kazetovým magnetofonem.

RIM, SIM

Instrukce, které jsme si ukazovali až dosud, fungují v procesoru 8080 i v procesoru 8085. Oba procesory jsou kompatibilní nejen na úrovni zdrojového kódu, ale i na úrovni binárního kódu (to znamená, že všechny instrukce mají stejné operační kódy).

Procesor 8085 přidává k instrukční sadě oficiálně dvě další instrukce, totiž RIM a SIM. Tyto instrukce v procesoru 8080 nefungují, nejsou implementovány…

Instrukce SIM vezme obsah registru A a podle něj nastaví přerušovací subsystém a sériový výstup. Takto:

D7	D6	D5	D4	D3	D2	D1	D0
SOD	SDE	…	R7.5	MSE	M7.5	M6.5	M5.5

Bit D7 určuje hodnotu, která se má zapsat na výstup SOD (Serial Output Data). Bit D6 (SDE, serial data enable) říká procesoru, jestli má bit D7 ignorovat (0), nebo jestli má podle něj změnit stav SOD.

Bit D5 je ignorovaný.

Bit D3 (MSE – Mask Set Enable) říká, jestli procesor má změnit maskování přerušení. Pokud je 0, ignoruje se, pokud je 1, nastavuje se podle bitů D2-D0.

Bity D2 – D0 maskují vstupy RST 7.5, RST 6.5 a RST 5.5. Pokud je odpovídající bit v 1, procesor přerušení z příslušného vstupu ignoruje, pokud je bit 0, procesor s přerušením pracuje.

Bit D4 nuluje klopný obvod na vstupu RST 7.5. V kapitole o přerušení procesoru 8085 jsem psal o tom, že tento vstup má v sobě klopný obvod, který se krátkým pulsem nastaví, a procesor pak může ve chvíli, kdy kontroluje stav přerušení, správně zareagovat. Pomocí instrukce SIM s D4 v logické 0 se tento klopný obvod opět vynuluje a je připraven na další příchod přerušovacího signálu.

```
MVI A, 0
SIM
```

Tato dvojice instrukcí nastaví klopný obvod u vstupu RST 7.5 do výchozího stavu.

```
MVI A, 11000000b
SIM
```

Tato dvojice nastaví výstup SOD na log. 1.

Instrukce RIM naopak čte stav přerušení a sériového vstupu:

D7	D6	D5	D4	D3	D2	D1	D0
SID	I7.5	I6.5	I5.5	IE	M7.5	M6.5	M5.5

Bit D1 odpovídá stavu na vstupu SID (Serial Input Data).

Bity D6 – D4 říkají, jestli na příslušných vstupech RST čeká na obsluhu nějaké přerušení.

Bit D3 informuje, jestli jsou přerušení povolena instrukcí EI (1) nebo zakázána instrukcí DI (0)

Bity D2 až D0 jsou ekvivalentní stejným bitům u instrukce SIM.

6.17 Ahoj světe!

Konečně! Konečně jsme se dostali do bodu, kdy si vypíšeme ono známé a populární AHOJ SVĚTE, a pak už nás nic nezastaví!

Použijeme k tomu, jak jinak, OMEN Alpha a vypisovat nebudeme přímo na obrazovku (ona taky žádná není), ale na sériové rozhraní. Jako sériové rozhraní nám slouží obvod 6850 a jeho programování jsme si už popisovali. Nyní tedy doopravdy – jak s ním budeme pracovat?

Jak pracovat s 6850?

Na začátku musíme nastavit řídicí registr 6850 tak, jak potřebujeme. Už jsme si řekli, že to bude hodnota 15h. Tím je obvod nastaven a připraven k přijímání a vysílání dat. Rozšíříme tedy naši inicializační sekci:

```
      .ORG    8000h ;inicializace()
COLD: DI
      LXI     SP,9000h ; bezpečná adresa

; adresace sériového rozhraní
ACIA  EQU     0DEh
ACIAC EQU     ACIA
ACIAS EQU     ACIA
ACIAD EQU     ACIA+1
;inicializace ACIA
      MVI     A,15h
; 115200 Bd, 8 bit, no parity, 1 stop bit, no IRQ
     OUT     ACIAC
```

Na začátku jsou pojmenované adresy ACIA (bázová adresa obvodu 6850 v našem počítači), ACIAC(ontrol), ACIAS(tatus) a ACIAD(ata). Vyhneme se tak programátorskému moru, „magickým konstantám".

Do registru A uložíme požadované řídicí slovo (15h) a instrukcí OUT ho zapíšete na adresu ACIAControl (tj. 0DEh). Vnější logika našeho počítače se postará o to, že data neskončí v paměti, ale tam, kde mají, tj. v obvodu 6850.

Co dál? Obvod je nastaven, teď je zapotřebí vypsat ono obligátní HELLO WORLD. Někde v paměti tedy bude tenhle řetězec a my ho budeme bajt po bajtu procházet a vysílat na sériový výstup.

Když se řekne „vysílat", tak si na to uděláme podprogram. Bude se jmenovat třeba SEROUT (jako že SERial OUTput) a jeho funkce bude, že vyšle hodnotu v registru A. Předtím si ale zkontroluje, jestli je vysílač volný. Musí si tedy načíst hodnotu stavového registru SR a zkontrolovat bit 1 (TDRE). Pokud je nulový, musí počkat, až bude 1.

```
ACIA_TDRE   EQU     02h
SEROUT:
            PUSH    PSW
SO_WAIT:    IN      ACIAS
            ANI     ACIA_TDRE
            ;bit TDRE - pokud lze vysílat, je =1
            JZ      SO_WAIT
```

```
POP     PSW
OUT     ACIAD
RET
```

Na začátku si uložím obsah registru A. Mám v něm ten bajt, co chci vyslat, ale budu ten registr potřebovat, protože si do něj načtu hodnotu stavového registru. Takže si jeho hodnotu uložím na zásobník.

Na dalším řádku načtu hodnotu stavového registru do registru A. Pak provedu logický součin (AND) s hodnotou 2. Hodnota 2 totiž binárně vypadá takto: 00000010 – jsou to tedy samé nuly, jen na pozici bitu 1, který potřebuju testovat, je jednička. Výsledkem logického součinu bude buď hodnota 2, pokud je bit 1 nastaven, nebo 0, pokud je nulový.

Připomeňme si: pokud je bit 1 stavového registru nulový, znamená to, že obvod 6850 ještě vysílá předchozí data a my musíme počkat, dokud to nedokončí. Tedy pokud je (hodnota stavového registru AND 02) rovna nule, čekáme. A přesně to zajišťuje další instrukce JZ. Pokud je příznak Z=1 (tedy předchozí operace skončila s výsledkem 0), tak JZ skáče. Tady se skáče opět na načtení stavového bajtu a vše se opakuje, dokud není výsledek nenulový. V tu chvíli už víme, že má 6850 volno a můžeme vysílat.

Pokud je tedy volno, přečteme si ze zásobníku zpět hodnotu, co byla původně v registru A a pomocí OUT ji zapíšeme do datového registru 6850 – ACIAData.

Správná otázka je: Co se stane, když náhodou bude obvod 6850 vadný, nebo nebude zapojený správně a bude vracet pořád hodnotu 0? V takovém případě, ano, tušíte správně, jste právě vygenerovali nekonečnou smyčku, ve které se bude procesor točit do skonání věků – pardon, do vypnutí napájení, do RESETu nebo do přerušení.

Výpis řetězce

Už zbývá jen detail – dát to všechno dohromady. Máme inicializovaný obvod ACIA, máme někde podprogram, který umí poslat znak na sériové rozhraní, teď už jen stačí vytvořit smyčku, která bude posílat jednotlivé znaky nejznámějšího nápisu v historii programování.

Buď můžeme zvolit postup mechanický, tedy postupně volat SEROUT s hodnotami pro znaky H, E, L, L, O… v registru A, nebo si ty znaky můžeme někam nějak zapsat a ve smyčce je postupně načítat.

Nejjednodušší způsob je uložit si nápis někam do paměti jako posloupnost znaků. K tomu slouží direktiva DB. Na začátku programu si do registrů HL uložíte adresu prvního znaku nápisu. Pak stačí jen opakovat následující postup:

1. Přečíst hodnotu z adresy, na kterou ukazuje HL, do registru A: MOV A, M
2. Zavolat SEROUT: CALL SEROUT
3. Zvýšit hodnotu HL o 1: INX H
4. Opakovat od bodu 1: JMP 1

Ale pozor – takto zapsáno to je nekonečná smyčka, která bude posílat stále dokola kompletní obsah paměti. Potřebujeme říct, kdy má přestat.

Používají se v zásadě tři postupy. První je ten, že za poslední znak dáme ještě znak 0. Někdy se tomu říká také ASCIIZ (ASCII s ukončovací nulou – Zero), někdy se tomu říká CSTR (protože takový způsob ukládání řetězců volí jazyk C). Pokud v bodu 1 načteme hodnotu 0, tak je vypsaný celý řetězec a můžeme skončit. Nevýhoda: Součástí řetězce nesmí být samotný kód 0.

Druhý způsob je ten, že jako první hodnotu řetězce dáme jeden byte, který bude obsahovat délku řetězce v bajtech. Tomuto způsobu se také někdy říká PSTR, protože takto ukládá řetězce Pascal. Algoritmus je o něco složitější:

1. Přečíst hodnotu z adresy HL do registru C: MOV C, M
2. Pokud je C=0, tak se vrátit
3. Zvýšit hodnotu HL o 1: INX H
4. Přečíst hodnotu z adresy, na kterou ukazuje HL, do registru A: MOV A, M
5. Zavolat SEROUT: CALL SEROUT
6. Snížit hodnotu C (Counter) o 1: DCR C
7. Skočit na bod 2: JNZ 2

Nevýhoda tohoto zápisu je jasná: řetězec může mít maximálně 255 znaků. Existují i varianty, které mají délku řetězce dvoubajtovou nebo čtyřbajtovou.

Třetí způsob využívá toho, že se většinou u takových počítačů používá jen prostá znaková sada ASCII, která má definované znaky jen v rozsahu 00 – 7Fh. Tedy žádné speciální, semigrafické, přehlásky, … Pokud je tomu tak, tak se použije způsob, podobný tomu prvnímu, ale s tím rozdílem, že místo toho, abychom za poslední znak přidali 0, tak my zapíšeme poslední znak tak, že mu nastavíme nejvyšší bit na 1 (tedy ho posuneme do rozmezí 80h-FFh). Výhoda je, že řetězec nezabírá ani o jediný byte navíc. Nevýhoda je, kromě toho, že nelze použít speciální znaky, třeba i složitější obsluha, ale zas tak hrozné to není. Představme si podprogram STROUT, který tento postup implementuje:

```
STROUT: MOV A,M
        ANI 7Fh
        CALL SEROUT
        MOV A,M
        ANI 80h
        RNZ
```

```
        INX H
        JMP STROUT
```

Načtený bajt nejprve ošetříme a případný nastavený bit 7 vynulujeme. Pak výsledek pošleme na výstup.

Znovu si načteme stejný bajt, a tentokrát se podíváme na nejvyšší bit. Pokud je nenulový, máme vše za sebou a vracíme se pryč (RNZ). Jinak standardní postup: Přejít na další adresu a celé opakovat znovu!

Možná by pomohlo uložit si hodnotu do nějakého jiného registru, nemuseli bychom pak číst dvakrát z paměti. Jenže uložení do jiného registru a navrácení zpět v tomto případě nepřinese žádnou výraznou úsporu – na velikost to bude o jeden bajt delší, na čas o jeden takt pomalejší… Proto jsem zvolil postup „čtení téhož údaje z paměti dvakrát“.

Všechny tři postupy mají svá pro a proti a své výhody a omezení. Který způsob použijete, to záleží na vás. V dobách osmibitů se hojně používal poslední (s negovaným bitem 7), ale pokud počítáte s tím, že například budete zobrazovat i jiné znaky než standardní sedmibitové ASCII, musíte zvolit jiný postup. Koncová nula je široce používaná dnes, právě díky tomu, že takto implementuje řetězce standardní Céčko. Takový řetězec může mít libovolnou délku, ale nesmí sám obsahovat nulu. Navíc to každý řetězec prodlouží o jeden byte.

Hotový program

```
            .ORG    8000h
                      ; inicializace()
COLD:       DI
            LXI     SP,8000h
                      ; adresace sériového rozhraní
ACIA        EQU     0DEh
ACIAC       EQU     ACIA
ACIAS       EQU     ACIA
ACIAD       EQU     ACIA+1
ACIA_TDRE   EQU     02h

;inicializace ACIA
; 115200 Bd, 8 bit, no parity, 1 stop bit, no IRQ
            MVI     A,15h
            OUT     ACIAC
WARM:
            LXI     H,HELLO
```

```
            CALL    STROUT
            JMP     WARM
SEROUT:     PUSH    PSW
SO_WAIT:    IN      ACIAS
            ANI     ACIA_TDRE
;bit TDRE - pokud lze vysílat, je =1
            JZ      SO_WAIT
            POP     PSW
            OUT     ACIAD
            RET
STROUT:     MOV     A,M
            ANI     7Fh
            CALL    SEROUT
            MOV     A,M
            ANI     80h
            RNZ
            INX     H
            JMP     STROUT
HELLO:
            DB      „HELLO WORLD“,0Dh,0Ah+80h
```

Na začátku proběhne inicializace – studený (cold) start. Nastaví se zásobník (bez něj by nefungovaly podprogramy) a obvod ACIA. Po inicializaci startuje vlastní obslužný program (teplý start). Tedy program: uloží do HL adresu řetězce, zavolá funkci STROUT a jede se znovu.

Zvolil jsem nakonec ukládání řetězců „tradičně osmibitově“. Na posledním řádku vidíte znaky k výpisu. Znak 0Dh je řídicí znak pro terminál, konkrétně CR – návrat kurzoru na začátek řádku. 0Ah je přechod na nový řádek. A protože znak přechodu na nový řádek je poslední, musí být jeho hodnota zvýšena o 80h (což je ekvivalent nastavení bitu 7).

Assembler ASM80 s těmito triky počítá, tak nabízí pseudoinstrukce .pstr, .cstr a .istr – příklad:

.PSTR „HELLO WORLD“
.CSTR „HELLO WORLD“
.ISTR „HELLO WORLD“

Všechny tři uloží do paměti řetězec HELLO WORLD, ale první ho uloží v „Pascal style“, tedy první byte bude délka a za ním znaky. Druhý ho uloží v „C style“ s nulou na konci. Třetí ho uloží v „inverted style“ – tedy poslední znak s nastaveným bitem 7.

6.18 Periferie: Klávesnice

Když jsem v předminulé kapitole psal o tom, že stisk klávesy vyvolá přerušení, tak, po pravdě řečeno, spíš ne… Totiž, ne že by to nešlo, ne že by to nebyl dobrý nápad, ale v dobách největšího rozmachu osmibitů snad žádný z nich nepoužíval přerušení při stisku klávesy (vzpomínám jen na jednu zvláštnost, a tou byla konstrukce klávesnice z nějakého Amatérského Radia, která vyvolala přerušení při stisku klávesy). Naprostá většina systémů v té době používala zcela jiný princip.

Klávesy na klávesnici se zapojovaly do matice N x M, kdy každá klávesa sídlila v nějakém průsečíku. Jedno z těchto čísel bylo často 8, aby se líp připojovala k periferním obvodům. Například klávesnice u ZX Spectra byla organizovaná do 8 řádků po 5 sloupcích. Řádky byly připojeny na vstupní port (s klidovým stavem 1), sloupce na výstupní. Když chtěl programátor vědět, jaká klávesa je stisknuta, poslal postupně 0 na jednotlivé vodiče sloupců a pokaždé si přečetl hodnotu řádků. Pokud byly všude 1, znamenalo to, že v tom sloupci není žádná klávesa stisknuta, když našel 0, věděl, že našel nějakou stisknutou klávesu.

Takto jednoduchý princip vedl k tomu, že při určité kombinaci stisknutí tří tlačítek se klávesnice tvářila, jako by bylo stisknuté i čtvrté. Stačilo stisknout klávesu, k ní pak některou ve stejném sloupci a další ve stejném řádku, a díky vzájemnému propojení se objevil falešný stisk klávesy ve čtvrtém rohu pomyslného obdélníku. Tento jev, nazývaný „ghosting“, se řeší různě. Buď tím, že mřížka neodpovídá fyzickému rozložení a minimalizuje se tak riziko tří stisknutí, popřípadě se ke každému tlačítku zapojuje sériově dioda. Krom toho se speciální tlačítka, u nichž se očekává, že budou stisknuta s jinými, zapojují buď do samostatných řádků, nebo zcela mimo matici.

U jednodeskových počítačů platilo totéž. Stejně byla zapojena klávesnice PMI-80, PMD-85, IQ-151 a dalších. U Atari 800 třeba taky, ale tam se o tohle testování stisknuté klávesy nestaral procesor, ale specializovaný obvod POKEY. U Commodore C64 byla rovněž matice a specializovaný obvod CIA.

6.19 Displej

Některé osmibitové počítače, zejména jednodeskové, měly jako displej nejrůznější sedmisegmentovky. Ty se většinou připojovaly přes nějaký port. U počítače BOB-85 měly dokonce i latche, tj. „paměti“ aktuálního stavu; naopak u PMI-80 bylo potřeba postupně vysílat informace o tom, co se má na které sedmisegmentovce zobrazovat. Displej byl připojen podobně jako klávesnice a během čtení jednotlivých sloupců se také rozsvěcely jednotlivé sedmisegmentovky…

Displeje u „větších“ počítačů byly řešeny buď jako znakové (IQ-151, SAPI-1), nebo grafické (PMD, Spectrum, …) – některé systémy měly inteligentní řadiče, které se dokázaly přepnout do

různých módů. Ať tak či onak, většinou se k displeji přistupovalo jako k paměti (přesněji řečeno: řadič displeje si sahal do vyhrazené oblasti v paměti RAM a zobrazoval odtamtud data).

6.20 Trocha assemblerové teorie

Kdysi jsem do rozhovoru pro programátorský magazín říkal, že *ASM80 je dvouprůchodový assembler* a že v dobách osmibitů byl „dvouprůchodový assembler" de facto standard. Prý to mám vysvětlit, co to znamená…

Musíme si říct, jak vlastně překladač funguje.

Bere řádek po řádku a kouká se, jestli tam je instrukce nebo pseudoinstrukce. Pokud ano, začlení si její kód do připravovaného výsledného útvaru, popř. provede to, co pseudoinstrukce nařizuje. Tím, že už v téhle fázi připravuje kód, tak vlastně ví, jak dlouhá která instrukce bude. Takže si může průběžně vytvářet i tabulku adres, kde má ke každému symbolickému jménu uloženou jeho adresu (nebo hodnotu). To je první průchod.

Ve druhém průchodu tohle všechno už má připravené, takže jede znovu, a jen na místech, kde je potřeba vyčíslit nějakou konkrétní hodnotu s odkazem, spočítá jeho hodnotu a doplní do kódu.

Takhle tedy funguje ASM80. Fungoval stejně i GENS (např.) Něco podobného (podobného!) dělal třeba assembler Prometheus, ale ten, nakolik jsem pochopil, dělal část prvního průchodu už při editaci textu, kdy si jednotlivé instrukce „předpřekládal".

Co jiné počty průchodů? Tříprůchodový? Jasně, existují. Dokážou třeba vyřešit problém několikanásobné dopředné reference, navíc by mohly trošku optimalizovat (plné skoky nahradit relativním, dlouhé varianty instrukcí těmi se zero page u 6502 a podobně).

Šel by napsat jednoprůchodový assembler? No jasně, šel – představte si první průchod, při něm to vyhodnocování výrazů… Když narazí na adresu, kterou už zná, tak ji použije, když ji ale ještě nezná, zapíše si do tabulky symbolických názvů tenhle název s poznámkou: „Až na tohle návěští narazím, doplním ho tam a tam".

Počet průchodů tedy dokáže ovlivnit některé aspekty chování překladače (dopředné reference by jednoprůchodový zvládnul).

A kolikaprůchodové se používají dneska?

Turbo assembler, poslední, který jsem na PC platformě používal, je „multiprůchodový". Ale vzhledem k jeho komplexnosti to ani jinak nejde. Ten hlavní rozdíl proti „starým pákám" je v tom, že assembler dneška nepřipraví hotový binární kód pro spuštění. Sestaví něco, čemu se říká „ob-

jektkód“, což je v podstatě výsledek posledního průchodu, ale nevadí mu, když nějaké adresy nerozpoznal. Pravděpodobně patří nějaké knihovní rutině. Objektkód pak dostane do spárů program, kterému se říká „linker“, a ten spojí vše potřebné – všechny části kódu, knihovní rutiny, data, vše. Teprve když tady nějaký symbol chybí, tak je zle. Překladač oddělený od linkeru má tu výhodu, že program může být napsaný v různých jazycích, v jednom třeba výpočty, v druhém UI, knihovny ve třetím – a linker to poslepuje dohromady. Toto rozdělení samozřejmě není nijaká novinka. Není to dokonce ani výmysl šestnáctibitového světa. I osmibity měly překladače a linkery zvlášť, např. u CP/M. Tohle rozdělení funguje tam, kde není problém pracovat se soubory. U osmibitového domácího počítače s kazeťákem by to moc smysl nedávalo.

6.21 Algoritmy v assembleru 8080

Algoritmy! Neseme čerstvé algoritmy! Berte, paní, jsou zadarmo, a přitom tak užitečné!

Přesun bloku dat

Máme 32 bajtů na nějaké adrese a chceme je zkopírovat na adresu úplně jinou. Jak na to? Jednoduše – použijeme instrukci LDIR a… cože? A jo vlastně, instrukci LDIR má až procesor Z80, u 8080 nic takového není, tam si to musíme pořešit jináč. Třeba takhle:

```
8000                    .ORG 8000h
8000 21 00 80           LXI h,8000h
8003 11 60 80           LXI d,8060h
8006 01 20 00           LXI b,32
8009 7E         LDIR:   MOV a,m
800A 12                 STAX d
800B 13                 INX d
800C 23                 INX h
800D 0B                 DCX b
800E 78                 MOV a,b
800F B1                 ORA c
8010 C2 09 80           JNZ ldir
```

Nejprve si naplním registry patřičnými údaji – do registrů HL dám adresu, kde se blok dat nachází, do registrů DE adresu, kam chci blok přesunout, do registrů BC počet bajtů, které chci přesunout. Ve smyčce LDIR (která se víceméně chová shodně s instrukcí LDIR procesoru Z80, až na to, že tady je pomalejší a přepisuje obsah registru A) se děje následující: Do registru A vezmu bajt z adresy HL, uložím ho na adresu DE, obě adresy zvýším o 1, od počítadla BC odečtu jedničku, a když je BC nenulové, opakuju smyčku.

Bohužel (tedy někdy spíš bohudík) instrukce INX, DCX nemění stav příznaků. Což je dobře, jak uvidíme v dalších algoritmech, ale v případě, jako je tento, by se nám hodilo, kdyby DCX dokázala

nastavit příznak Z, jako že je výsledek nula. Ale nedělá to, takže si to musíme vyřešit jinak (zvykejte si, to je assembler!)

Každý podobný případ lze řešit několika způsoby, ale časem se ustálí vždy jeden *návrhový vzor*, který má nejméně nevýhod. V případě „zkontroluj, jestli je ve dvojici registrů 0" se používá ten, který jsem ukázal v kódu: do A si zkopíruju jeden registr z dvojice, udělám OR s druhým (vzpomeňte – výsledkem OR je nula, pokud všechny bity obou operandů jsou nulové) – a protože OR už příznakový bit Z změní podle výsledku, tak jsme získali, co jsme chtěli.

Jak bychom to ale řešili, kdyby v A bylo něco, co chceme zachovat? Řešit by to samo sebou šlo, jen – jinak a náročněji.

Přepsání bloku dat

V procesoru Z80 se instrukce LDIR používá i k jiné operaci, než je přesun bloků dat – totiž k mazání (přesněji k nastavení celého bloku na konkrétní hodnotu). Využívá se toho, že nastavíme registry HL a DE určitým způsobem – HL na začátek bloku, který se má smazat, DE na adresu o 1 vyšší. Ručně pak nastavíme první bajt bloku (ten, kam ukazuje HL) na požadovanou hodnotu a LDIR postupně první bajt bloku nakopíruje na další a další adresy. BC je v takovém případě (délka bloku – 1). Pokud nastavíme DE na adresu větší než HL+1, třeba HL+4, vyplní LDIR blok paměti vzorkem dat (HL, HL+1, HL+2, HL+3).

Fungovalo by to i s tím naším LDIRem? Ale určitě fungovalo, jen je to v případě nastavení nějaké oblasti v paměti na konkrétní hodnotu trošku kanón na vrabce (a to v assembleru znamená vždy: stojí to paměť a čas). Jednodušší postup je zde:

```
8000                    .ORG 8000h
8000 21 40 80           LXI H,8040h
8003 01 20 00           LXI B,0020h
8006 1E 55              MVI E,55h
8008 73          FILL:  MOV M,E
8009 23                 INX H
800A 0B                 DCX B
800B 78                 MOV A,B
800C B1                 ORA C
800D C2 08 80           JNZ fill
```

Všimněte si, že hodnota, která se do bloku paměti zapisuje, není v registru A, ale v registru E. Je to právě kvůli výše popsanému postupu na testování BC na nulu – využívá registr A. Kdybychom v něm měli tu hodnotu, přepsali bychom si ji. Abychom si ji nepřepsali, museli bychom si ji někam uložit (pravděpodobně třeba do toho registru E), pak zase načíst zpátky… Výsledek by byl sice funkční, ale pomalejší a delší. A to jen kvůli tomu, že chceme číslo v jiném registru?! Kdepak.

Dělení

V jedné z minulých kapitol jsme si ukazovali, jak násobit dvě celá čísla bez znaménka. Dělení můžeme napsat analogicky – buď jako postupné odčítání, nebo pomocí rotací a odčítání. Ten druhý postup opět můžeme napsat bez triků, nebo s trikem…

Tady je postup pro dělení dvojregistru HL registrem D. Matematicky: HL = HL / D, zbytek po dělení je v registru A. Je to postup s trikem (používá se „trojitý registr" AHL, do A se postupně nasouvají bity z HL a kontroluje se, jestli už je mezistav větší než dělitel).

```
8000                    .ORG 8000h
8000 21 03 14           LXI H,5123
8003 16 0A              MVI D,10
8005            DIV:
8005 AF                 XRA A
8006 06 10              MVI B,16
8008            DIVLOOP:
8008 29                 DAD H
8009 17                 RAL

800A BA                 CMP D
800B DA 10 80           JC DIVNEXT
800E 92                 SUB D
800F 2C                 INR L
8010            DIVNEXT:
8010 05                 DCR B
8011 C2 08 80           JNZ DIVLOOP
```

Ukázali jsme si pár triků a předvedli pár algoritmů. Ovšem v programování obecně a pro assembler zvlášť platí, že:

1. Není důležité si tyhle algoritmy pamatovat, důležitější je umět je napsat, když je potřeba
2. Je dobré si tyhle algoritmy pamatovat, aby je člověk nepsal zbytečně znovu
3. Nejdůležitější je dokázat se rozhodnout, jestli pro daný konkrétní případ platí 1 nebo 2!

6.22 Konstrukce z vyšších programovacích jazyků

Assembler je tak blízko procesoru, jak jen může být. Když se jej naučíte, zjistíte, jak pracuje procesor, a hned vzápětí si uvědomíte, jak moc odlišné to je od všeho toho komfortu, co známe dnes. Strukturovaným programováním počínaje a objektovým či funkcionálním konče.

Většina osmibitových procesorů vlastně žádné strukturované konstrukce nemá. Všechno se vždycky dá nějak obejít, odněkud někam skočit, něco někde přepsat, a i z nejvnitřnějšího cyklu můžete skočit ven a doprostřed podprogramu… Zkrátka procesor pracuje spíš stylem programů v BASICu nebo FORTRANu než v čemkoli strukturovaném.

Na datové struktury můžete zapomenout taky. Paměť je jen jedno velké pole a všechny datové struktury se v něm emulují pomocí ukazatele na strukturu a znalosti offsetů jednotlivých položek. O tohle všechno se vám naštěstí postará překladač, takže ani nevíte, že dole, tam úplně dole, je to jen nestrukturovaný chaos a jedno velké pole.

Samozřejmě se i v assembleru dá programovat strukturovaně, ale nic vás nenutí. Jen vaše sebekázeň, pokud ji v sobě najdete. Na druhou stranu programovat striktně strukturovaně v assembleru znamená, že vaše programy budou pravděpodobně zbytečně delší a pomalejší, než by mohly být. Přesto se hodí vědět, jak konstrukce z vyšších programovacích jazyků do assembleru převést. Už jsem některé ukazoval, ale pojďme si je pro jistotu shrnout a zopakovat:

if (podmínka) {blok příkazů}

V assembleru nemáme žádné podmínky. Jediné, co je, jsou příznakové bity – příznak nulového výsledku, záporného výsledku, přenosu, … U podmíněné konstrukce použijeme nějaké porovnání, pravděpodobně akumulátoru s registrem nebo přímou hodnotou. Pokud podmínka neplatí, přeskočíme příkazy.

Dejme tomu, že chceme zapsat podmínku if (A=15) {…něco…}

```
   CPI 15
   JNZ SKIP
   … něco…
SKIP:
   … pokračování programu …
```

Instrukce CPI porovnává akumulátor s přímou hodnotou, a pokud jsou obě hodnoty stejné, nastaví příznak Z (Zero). Pokud nejsou, bude příznak Zero nulový a program skočí na návěstí SKIP.

if (podmínka) {blok1} else {blok2}

Jen o trošku složitější, než předchozí případ.

```
   CPI 15
   JNZ ELSE
   …blok 1 …
   JMP SKIP
```

```
ELSE:
   …blok 2 …
SKIP:
   …pokračování…
```

do {blok} while (podmínka)

Nejjednodušší cyklus je ten s podmínkou na konci. Pokud podmínka platí, skáče se opět na začátek. Pro příklad opět podmínka A=15

```
DO:
   … blok příkazů …
   CPI 15
   JNZ DO ; podmínka neplatí -> skok na začátek
   … pokračování programu …
```

while (podmínka) {blok}

Cyklus s podmínkou na začátku. V zásadě jde jen o rozšířený IF, kde na konci bloku příkazů je zase skok na začátek – na vyhodnocení podmínky.

```
WHILE:
   CPI 15
   JNZ SKIP
   … blok …
   JMP WHILE
SKIP:
   … pokračování …
```

Break a continue

První příkaz je vlastně skok za konec cyklu – u předchozího příkladu by to bylo tedy na návěstí SKIP. Příkaz *continue* vyvolá další iteraci cyklu a ve výše uvedených příkladech je ekvivalentní skoku na návěstí WHILE, resp. DO.

for (inicializace; podmínka; změna) {blok}

Pro převod takovéhoto cyklu do assembleru je dobré si uvědomit, že cyklus FOR můžeme přepsat pomocí cyklu WHILE takto:

```
… inicializace …
while (podmínka) {
   … blok …
   změna
}
… pokračování …
```

Použití strukturovaných konstrukcí

Stačí vlastně jen tyto konstrukce poctivě zanořovat a pamatovat si, že kromě *break* a *continue* nejsou přípustné žádné jiné skoky dovnitř či ven. Každý podprogram (ekvivalent *procedury* či *funkce*) musí mít jeden vstupní bod a jeden výstupní bod. Pokud tato pravidla budete dodržovat, neznamená to, že vaše programy budou jen díky tomu funkční a správné. Jen budou podstatně lépe udržovatelné. Za což ovšem budete platit daň v podobě neefektivity. Na velkých a výkonných strojích dneška vás pár taktů navíc většinou nebude moc trápit, ale u osmibitu může několik ušetřených taktů ve vnitřním cyklu zrychlit výsledný program o celé řády.

6.23 Nedokumentované instrukce 8085

Nedokumentované instrukce jsou takový drobný dárek, který dostanete od výrobce procesoru ve chvíli, kdy už všechny instrukce znáte a říkáte si: *A co se stane, když pošlu operační kód instrukce, co není v tabulce popsaná?*

Stát se může cokoli. U některých procesorů a některých kódů ho můžete klidně zablokovat, ale někdy se dočkáte zajímavých efektů. Někdy nesmyslných, ale někdy užitečných.

Jak takové instrukce vznikají? Někdy to je vedlejší efekt vnitřní architektury procesoru, jako třeba u 6502 (těm s dovolením věnuju samostatnou kapitolu), někdy to ale působí dojmem, že výrobce instrukce zabudoval, ale pak se z nějakého důvodu, který většinou neznáme, rozhodl, že je nenapíše do dokumentace. To je třeba případ procesoru Z80, kde se pomocí prefixů dá pracovat i s polovinami indexových registrů, popřípadě se ukáže série nedokumentovaných instrukcí pro posuvy.

U 8085 také existuje několik nedokumentovaných instrukcí a dva nedokumentované příznakové bity. Pojďme si je představit.

Ještě než si je představíme, musím říct jednu zásadní věc: Tím, že jsou instrukce nedokumentované, jsou vlastně tak trochu i „bez záruky“. Může se tedy stát, že použijete procesor od jiného výrobce, a tam ty instrukce fungovat nebudou, nebo budou fungovat jinak, to vám dopředu nikdo nezaručí. Proto je lépe si nejprve ověřit, že nedokumentované instrukce implementovali i výrobci „druhých zdrojů“ a „klonů“.

Přehled nedokumentovaných instrukcí 8085

V kapitole o příznacích jsem psal: „V procesoru 8080 je příznakových bitů 5 a jsou v registru F uloženy takto…“

BIT	7	6	5	4	3	2	1	0
PŘÍZNAK	S	Z	0	AC	0	P	1	CY

- S (sign) informuje o znaménku výsledku. Je-li výsledek kladný, je to 0, je-li výsledek záporný, je to 1
- Z (zero) je roven 1 v případě, že výsledek je nula. Pokud je nenulový, je Z = 0
- AC (auxiliary carry) je roven 1, pokud při operaci došlo k přenosu přes polovinu bajtu (mezi nižší a vyšší čtveřicí)
- P (parity) je nastaven vždy tak, aby doplnil počet jedniček ve výsledku na lichý (tedy 0, je-li lichý, 1, je-li sudý) – viz též paritní bit.
- CY (carry) je 1, pokud dojde k přenosu z nejvyššího bitu.

V dokumentaci procesoru 8085 je to napsáno stejně, ovšem díky pečlivé práci všímavých programátorů se postupně přišlo na to, že bity 5 a 1 příznakového registru nejsou vždy konstantní a že mají nějakou funkci.

Ukázalo se, že bit 1 udává přetečení (Overflow) výsledku u operací sčítání a odčítání při práci s čísly se znaménkem. Proto byl (neoficiálně) označen jako V. Bit 5 má funkci složitější. U inkrementu a dekrementu dvojice registrů (INX H třeba) udává přetečení/podtečení. Při obyčejných aritmetických instrukcích lze jeho hodnotu popsat jako „znaménko výsledku XOR příznak přetečení". Takto nedává hodnota příznaku moc smysl, ale ve skutečnosti je to užitečná hodnota, která dává smysl při odečítání či porovnávání čísel se znaménkem. Tento příznak dostal označení K, popřípadě X5.

Ken Shirriff, který se zabývá reverzním inženýrstvím křemíkových čipů, odhalil přesné zapojení obvodů, které tyto příznaky nastavují. Díky tomu víme, jak přesně tyto příznaky fungují. V následující tabulce jsou shrnuté různé kombinace odčítání dvou čísel se znaménkem. Představte si, že levý operand je v registru A, pravý v registru B a provedete SUB B (popřípadě CMP B). Je vidět, že K=1 v případě, že B>A (se znaménkem).

Operace	Čísla se znaménkem	C	S	V	K
50h – f0h = 60h	80 – -16 = 96	0	0	0	0
50h – b0h = a0h	80 – -80 = -96	0	1	1	0
50h – 70h = e0h	80 – 112 = -32	0	1	0	1
50h – 30h = 120h	80 – 48 = 32	1	0	0	0
d0h – f0h = e0h	-48 – -16 = -32	0	1	0	1
d0h – b0h = 120h	-48 – -80 = 32	1	0	0	0
d0h – 70h = 160h	-48 – 112 = 96	1	0	1	1
d0h – 30h = 1a0h	-48 – 48 = -96	1	1	0	1

Kromě dvou nových příznakových bitů máme u procesoru 8085 i pár nových nedokumentovaných instrukcí. Instrukce RIM a SIM jsme si už představovali, ale ty v datasheetech jsou. Ukažme si ty, co v nich chybí.

Poznámka: Názvy instrukcí jsou neoficiální, takže některé mají dva různé, podle toho, jaká konvence byla při tvorbě jejich pojmenování použita.

Operační kód	Takty	Název	Popis	Ovlivňuje příznaky
08	10	DSUB	HL = HL – BC	S, Z, AC, P, CY, K i V
10	7	ARHL (RRHL)	Aritmetický posun HL doprava. Nejvyšší bit registru H zůstane nezměněn, nejnižší bit registru L je přesunut do příznaku CY	CY
18	10	RDEL (RLDE)	Rotace DE doleva přes příznakový registr. Bit 7 registru D je přesunut do CY, CY je přesunut do bitu 0 registru E	CY
28 nn	10	LDHI n8 (ADI HL,n8)	DE = HL + n8 – přičítá osmibitovou konstantu	
38 nn	10	LDSI n8 (ADI SP,n8)	DE = SP + n8. Přičítá osmibitovou konstantu	
CB	6/12	RSTV (OVRST8)	RST 8 (na adresu 0040), pokud je nastaven příznak V	
D9	10	SHLX (SHLDE)	Na adresu v registrovém páru DE uloží obsah registru L, na adresu o 1 vyšší uloží obsah registru H	
DD addr	7/10	JNK addr (JNX5) addr	Skok na zadanou adresu, pokud příznak K=0	
ED	10	LHLX (LHLDE)	Do registru L uloží obsah paměti na adrese DE, do registru H na adrese o 1 vyšší	
FD addr	7/10	JK addr (JX5 addr)	Skok na zadanou adresu, pokud K=1	

Některé z těchto instrukcí mohou být velmi užitečné. SHLX a LHLX nabízejí velmi rychlý přístup k paměti (přenesou se dva bajty během 10 T). ARHL je rychlé dělení HL dvěma se zbytkem v registru CY, DSUB se někdy hodí pro šestnáctibitové odečítání… Ale jak jsem varoval výš: Nelze se stoprocentně spolehnout, že všechny procesory 8085 mají tyto instrukce implementované. Měly by mít, ale to víte: vše je neoficiální!

7 Intermezzo zvukové

7 Intermezzo zvukové

Kdo by neznal jednobitovou muziku! Tedy vlastně, hm, všichni Ataristi a Commodoristi, jejichž počítače měly mnohakanálové zvukové generátory. To my na Spectru jsme si museli vystačit s jedním drátem, kde byla buď 0, nebo 1, a podle toho, jak rychle se s ním (programově) cvakalo, tak takový zvuk to dělalo…

Tak samozřejmě nebyla to žádná extra sláva, ale šlo to, a když se tomu dalo trošku péče, tak byly výsledky opravdu zajímavé.

Máme hezký osmibit postavený, ten má vývod jako stvořený pro nějaký *pípák*, tak proč tedy něco jednoduchého nezapojit? Technicky bude stačit oddělovací kondenzátor, tak 10M, a za něj zesilovač. Buď z tranzistoru, nebo klidně jako hotový modul, co se koupí v obchodě za pár desetikorun. Ale zas to prosím nepřehánějte s kvalitou zesilovače, byly by to vyhozené peníze, protože jednobitová muzika není žádné HiFi.

7.1 Trocha nezbytné teorie na úvod

Zvuk je, jak známo, vlnění, přenášené vzduchem nebo jiným médiem. Jako základní typ vlnění se bere sinusoida. Ne snad proto, že by se na tom sympozium hudebníků usneslo, ale proto, že v přírodě ty nejjednodušší kmity probíhají právě po sinusoidě.

Tón je základní stavební jednotkou hudby. Jeho nejdůležitějším parametrem je frekvence, udávaná v hertzech, neboli „počet cyklů za sekundu“.

Komorní A je tón, který má většinou frekvenci 440 Hz. (Proč většinou? Protože jsou ladění, které mají komorní A někde jinde, např. na 442 Hz, Händel používal třeba i 409 Hz, La Scala v 18. století zase měla komorní A na frekvenci 451 Hz…)

Oktáva je podle klasických hudebních teoretiků interval osmi tónů. Podle Františka Fuky to je 12 půltónů a vřele doporučuju si jeho článek přečíst.

https://8bt.cz/zvuk

Pokud to z jakéhokoli důvodu neuděláte, tak aspoň vězte, že „o oktávu vyšší tón“ má dvojnásobnou frekvenci v Hz. Pokud má komorní A frekvenci 440 Hz, tak o oktávu vyšší A bude mít 880 Hz. Vzhledem k tomu, že frekvence vibrace je nepřímo úměrná délce vibrujícího předmětu, tak by mohlo platit, že struna délky L vydává nějaký tón a struna délky L/2 vydává tón o oktávu vyšší. Máte-li kytaru, račte si přeměřit vzdálenosti od pražce ke kobylce a ověřit, jestli pražec v poloviční vzdálenosti vyvolá o oktávu vyšší tón.

Samplování je postup, při kterém ze spojitého průběhu, třeba té sinusovky, uděláme nespojitý. Děje se to tak, že v pravidelných intervalech bereme vzorek (*sample*) aktuální hodnoty jako číslo a ukládáme si ho. Pokud pak ve stejných intervalech nastavíme výstup na tuto hodnotu, dostaneme průběh, který s určitou mírou tolerance odpovídá původnímu.

Samplovací frekvence je frekvence, s jakou vzorkujeme. Pro nás u jednobitové muziky to bude frekvence, s jakou jsme schopni změnit hodnotu na výstupu. Z čehož vyplývá, že pokud budeme s touto rychlostí měnit jedničky a nuly, dostaneme obdélníkový průběh o poloviční frekvenci – a to bude zároveň maximální dosažitelná frekvence hraní. (Čímž jsme si, mimo jiné, hezky odvodili Shannonův, též Nyquistův, teorém).

Sinusoida je základní zvukový průběh. Naštěstí žádný nástroj nemá zvuk, který by byl přesně sinusový, což je dobře, protože sinusoida je velmi plochý, tupý a bezbarvý zvuk.

Amplituda je pro nás totéž, co hlasitost. Čím větší jsou výchylky vln od neutrální hodnoty, tím hlasitější zvuk vnímáme. Ucho vnímá jen absolutní odchylku, nerozlišuje, jakým je (ta odchylka) směrem, jestli kladným, nebo záporným (v případě přenosu zvuku vzduchem: nerozlišuje, jestli přišlo zředění nebo zhuštění, vnímá jen rozdíl oproti normálu).

Barva zvuku je dána v reálném světě stavbou nástroje, materiálem, ozvučnicí – každý z těchto faktorů přidá k základnímu tónu nějakou další složku, jiné chvění s jinou intenzitou, čímž dodá zvuku charakteristické zabarvení. U složitých hudebních nástrojů rezonují různé součástky na různých frekvencích a kombinace těchto rezonančních efektů je unikátní a pro daný nástroj charakteristická. Jednoduché nástroje mají jednoduchý zvuk – například nejrůznější píšťaly. Složité nástroje, třeba takové, kde vzniká tón například drnkáním, mají zvuk složený z nejrůznějších frekvencí, které v čase mění svou amplitudu…

Pila (sawtooth) je průběh zvuku, který vypadá – inu, jako pila. Je zvláštní, že pilu můžeme snadno sestrojit ze sinusovek – vezmeme tu základní, třeba 440, k ní přidáme dvakrát rychlejší (880) s poloviční amplitudou, k tomu třikrát rychlejší (1320) s třetinovou amplitudou… A když to uděláme donekonečna, dostaneme pilový průběh.

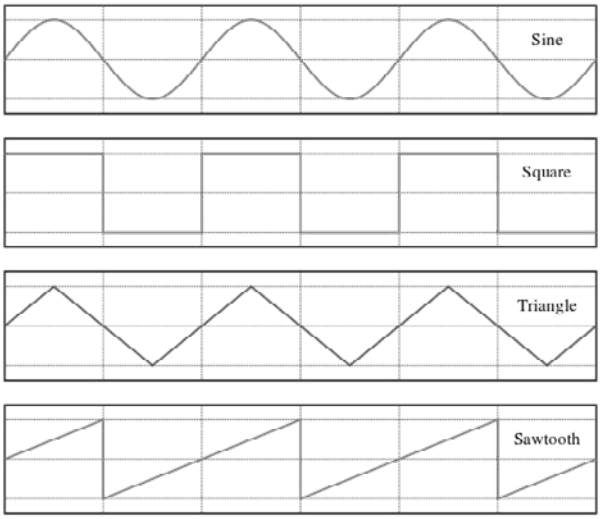
Harmonická frekvence je taková, jejíž velikost je nějakým celočíselným násobkem základní frekvence. První harmonická je rovna základní frekvenci, druhá harmonická dvojnásobné, třetí trojnásobné… Předchozí definici pily bychom mohli zjednodušit tak, že to je součet všech harmonických (první, druhé, třetí, …) s lineárně klesající amplitudou.

Trojúhelník je další oblíbený průběh, který uvádím vlastně jen tak do počtu. Stejně jako pilu lze zkonstruovat i trojúhelník součtem harmonických – tentokrát lichých harmonických (1., 3., 5., …) s exponenciálně klesající hlasitostí.

Obdélník je poslední ze „svaté čtveřice" základních průběhů. Namísto plynulého klesání nebo stoupání má pouze dvě hodnoty – maximum a minimum. Konstruuje se stejně jako trojúhelník,

tedy součtem lichých harmonických, ale hlasitost tentokrát neklesá exponenciálně, tedy s druhou mocninou, ale lineárně.

Šum je náhodný průběh.



Obrázek 34: základní zvukové průběhy

To rozkládání základních periodických funkcí na součty sinusovek je samo o sobě velmi zajímavá část matematiky, a pokud by vás to zajímalo víc, tak klíčová slova jsou Fourierova transformace a Fourierovy řady.

7.2 Obdélník vládne všem

Jednobitová hudba je generována tak, že měníme hodnotu jediného bitu – buď je 0, nebo 1. Výsledkem je tedy, když se zamyslíme, obdélníkový průběh. On nakonec nebude úplně přesně obdélníkový, protože na cestě od procesoru k zesilovači jsou ještě různé obvody, které ten obdélník trošičku zdeformují, ale to není teď pro nás podstatné. Podstatné je, že můžeme zapomenout na sinusoidy a trojúhelníky, protože vygenerujeme vždy a pouze obdélník.

Když chceme zahrát třeba tón o frekvenci 1000 Hz, bude to znamenat, že tisíckrát za sekundu (=každou milisekundu) musí proběhnout oba cykly, tedy 1 a 0. Nejjednodušší tedy bude, když necháme půl milisekundy logickou 1, další půl milisekundy logickou 0, dohromady to dá jednu milisekundu, a když to budeme opakovat, dostaneme tisícihertzový obdélník jako víno!

Když je polovinu doby 1 a polovinu 0, říkáme tomu, že signál má střídu 1:1 (též 50%). Stejnou dobu je zapnuto, jako vypnuto. Spectrum 48 nám takový signál vygeneruje na požádání – ano, je to to, co zahraje BEEP.

Co když ale zmenšíme dobu, po kterou máme tu logickou 1, o polovinu? Výsledek bude tedy mít střídu 1:3 (25%, tj. čtvrtinu doby 1, tři čtvrtiny 0) a zvuk bude znít jinak, bude ostřejší – i když ho stále budeme vnímat jako „stejně vysoký".

Zvláštní, takovou *bzučivou*, hudbu měl na Spectru třeba Heartland nebo hry od Special FX (Firefly). Použitá byla i v hudebním editoru Orfeus. Má střídu 1:N – tedy pouze na začátku cyklu na malou chvíli zapneme log. 1 a pak je logická 0 až do konce požadované doby. (Ta *malá chvíle* je dána samplovací frekvencí, tedy rychlostí, s jakou jsme schopni v naší rutině přepínat 1 a 0.)

7.3 Dělíme frekvence

Jak tedy budeme hrát jednotlivé tóny? Když chci zahrát 440 Hz, co to přesně znamená?

OMEN Alpha běží na frekvenci 1,8432 MHz. Tedy necelé dva miliony taktů za sekundu. Známé T – doba jednoho cyklu hodin – je tedy 0,54253 mikrosekundy.

Frekvence tónu *f* je 440. Za jednu sekundu proběhne tedy 440 cyklů, to znamená, že na jeden cyklus vyjde… 1843200/*f*= 4189 cyklů T. Když každých zhruba 2094 T změníme výstup z log. 0 na log. 1 (resp. obráceně), bude nám Alpha hrát krásné obdélníkové komorní A.

Jenže my nemůžeme měnit výstup s frekvencí hodinového signálu – nejrychlejší instrukce má 4 takty, navíc potřebujeme k tomu nějaké počítání… No, dejme tomu, že jeden průběh smyčkou našeho hracího programu, tedy tou částí, kde se rozhodne, jestli teď 0 nebo 1, zabere třeba 50 T. Prostým podělením 1843200/50 zjistíme, že tahle smyčka proběhne 36864x za sekundu. Když budeme pravidelně střídat 1 a 0, vygenerujeme tón o frekvenci 18,5kHz – to je nepříjemné vysoké pištění. Ultrazvuk to není, ale je to hodně vysoké a člověk by to měl slyšet (horní hranice slyšitelné frekvence je okolo 20 kHz, v mládí víc, k stáru se zhoršuje).

Hrací smyčka funguje v zásadě – a hrubě zjednodušeně – tak, že počítá průběhy, tj. kolikrát proběhla. Když je dosaženo hodnoty D (dělitel), jede se zase od nuly, protože to znamená nový průběh. Během té doby je tedy potřeba změnit výstup podle požadovaného průběhu. Příklad: pro „heartlandovský" průběh 1:N bude na výstupu log. 1 pouze když je počítadlo rovno 0, pak bude na výstupu log. 0. Logická jednička bude na výstupu tedy po dobu jednoho průběhu, 50T.

Kolik je tedy dělitel? Je to v zásadě převrácená hodnota frekvence (1/*f*) a vzorec je: D = M / *f* / Tcyc. M je frekvence hodin (1843200), *f* je požadovaná frekvence tónu, Tcyc je trvání smyčky v taktech hodin (T), a výsledkem je číslo, které udává, kolik průchodů smyčkou připadá na jeden cyklus vý-

sledného tónu. Pro komorní A nám to vychází 83,78…, a protože máme registry celočíselné, tak 84. Tón o oktávu vyšší bude mít dělitele 42 (čím menší dělitel, tím vyšší tón), o oktávu nižší pak 168.

Jaká je maximální a minimální frekvence našeho hypotetického *hradla*? Maximální frekvence je taková, která se ozve s dělitelem 2 (pokud by byl 1, hodnota na výstupu by se neměnila) – po dosazení do vzorce vychází 18,432 kHz. Minimální frekvence bude, při osmibitovém počitadle, 145 Hz.

Teď už zbývá jen spočítat hodnoty pro jednotlivé půltóny. Podíl frekvencí dvou sousedních tónů je dvanáctá odmocnina ze dvou – naštěstí si to nemusíte počítat a můžete využít tabulky frekvencí. (Z ní se taky dozvíme, že naše nejnižší frekvence je cca tón D3, nejvyšší pak zhruba B8.)

http://www.phy.mtu.edu/~suits/notefreqs.html

Dostaneme tabulku dělitelů:

Nota	Frekvence	Dělitel	Reálná frekvence	Rozladění
D3	146.83	251	146.8685259	1
D#3/Eb3	155.56	236	156.2033898	1.004
E3	164.81	223	165.309417	1.003
F3	174.61	211	174.7109005	1
F#3/Gb3	185	199	185.2462312	1.001
G3	196	188	196.0851064	1
G#3/Ab3	207.65	177	208.2711864	1.002
A3	220	167	220.742515	1.003
A#3/Bb3	233.08	158	233.3164557	1.001
B3	246.94	149	247.409396	1.001
C4	261.63	140	263.3142857	1.006
C#4/Db4	277.18	132	279.2727273	1.007
D4	293.66	125	294.912	1.004
…				
A6	1760	20	1843.2	1.047
A#6/Bb6	1864.66	19	1940.210526	1.04
B6	1975.53	18	2048	1.036

Vzhledem k tomu, že používáme celočíselné operace, tak nebudou frekvence přesné a v místech, kde máme hodně tónů na málo hodnot (tedy u vysokých tónů) bude už velmi slyšitelné rozladění. Ostatně už v sedmé oktávě připadá na jednoho dělitele několik tónů a šestá bude taky slyšitelně rozladěná.

7.4 Vícehlas

Probrali jsme si teorii jednoho tónu. Jenže jednobitová hudba dokáže hrát i víc tónů naráz. Jak se to dělá?

Tak, buď můžete rychle měnit frekvence tónů a hrát chvilku C a chvilku E a takhle to střídat, anebo můžete oba tóny skládat dohromady – tj. ve smyčce máte dvě počítadla, každé pro jeden tón, takže vám vznikají dva průběhy, a výsledek je pak OR nebo AND těchto průběhů. (U střídy 1:N nemá AND smysl.)

Takhle jednoduché že by to bylo? No ano, je to tak. Teoreticky. V praxi narazíte na spoustu zádrhelů. Například na to, že když použijete generátor se střídou 1:N a zahrajete dva tóny, které jsou od sebe přesně o oktávu, tak vám impulsy toho vyššího splynou s impulsy toho nižšího a vy uslyšíte jen ten vyšší. Řešit to lze – buď jeden tón decentně rozladíte, tj. posunete jeho frekvenci o kousíček jinam, např. přičtete k děliteli 1 (což dělejte s tím nižším tónem), nebo mu posunete fázi (tj. nebudete posílat log. 1 ve chvíli, kdy je počítadlo rovno 0, ale třeba 4). Tím sice docílíte toho, že budou znít oba, ale navíc si přidáte do výsledku parazitní frekvence. Na druhou stranu – v tom obdélníkovém šumu se to ztratí…

Nebo že u jiných stříd (1:1 například) zní dva tóny jako by měly poloviční hlasitost proti jednomu tónu. Mnojo…

Ještě takový detail – když u střídy 1:N měníme tu první hodnotu (tedy 2:N, 3:N, 4:N), tak do určité meze můžeme simulovat změnu hlasitosti. Pokud ale zvolíme první číslo příliš velké, tak se u vyšších frekvencí změní střída na něco blízké 1:1 – a zase máme jiný problém.

Když budou hrát dva tóny, jen mírně rozladěné, vzniknou ve výsledném průběhu takzvané zázněje. Je to tím, že při hraní dvou frekvencí naráz vnímáme i třetí, která je rovna jejich rozdílu – toho lze taky využít při generování zvuků.

7.5 Ke čtení a inspiraci

- http://1bit.i-demo.pl/forum/2/zx-spectrum-48k-timex-2048-music/
- http://shiru.untergrund.net/1bit/
- http://sysel.webz.cz/sound/zx10.html

8 OMEN Bravo

Vhodné pro výuku!

OMEN Bravo

Chcete se stát systémovým programátorem?

Mikroprocesor 6502 stále žije! Tento legendární obvod, který poháněl řadu špičkových počítačů své doby, včetně takových hvězd, jako byl Apple I, Apple II, britský BBC nebo nejprodávanější osmibit Commodore C64, najdete i ve výukovém počítači OMEN Bravo! Spolu s ním je na desce i 32 kB RAM, 8 kB ROM, sériové a paralelní porty. Jednoduché zapojení, jednoduché programování. Vyzkoušejte si sami doma práci s tímto procesorem. OMEN Bravo je ideální vstupní brána.

8 OMEN Bravo

8.1 Architektura procesoru 6502

Doufám, že jste nepřeskočili kapitolu o architekturách procesorů, kde jsem popisoval základní rozdíly mezi architekturou procesorů 8080 a 6800. Téměř všechno, co jsem psal o architektuře 6800, lze vztáhnout na procesor 6502, s jedinou výjimkou, a tou je zápis vícebajtových hodnot. Na rozdíl od 6800 používá 6502 zápis „Little Endian“, tedy stejný jako 8080 (nejprve je nižší byte, pak vyšší).

Procesor 6502, stejně jako jeho vzor 6800, nesází na velký počet interních registrů. Kromě programového čítače (PC), ukazatele zásobníku (SP) a registru příznaků má pouhé tři registry. Jeden je akumulátor, vůči němu se provádějí matematické a logické operace, a dva mají funkci indexových registrů pro přístup do paměti (registry X a Y). Kromě PC jsou všechny registry osmibitové (PC má 16 bitů).

Registry 6502

15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00	(pozice bitu)
Akumulátor																
—								A								Accumulator
Indexové registry																
—								X								X index
—								Y								Y index
Ukazatel zásobníku																
0	0	0	0	0	0	0	1	SP								Stack Pointer
Programový čítač																
PC																Program Counter
Stavový registr (registr příznaků)																
—								N	V	1	B	D	I	Z	C	P Processor flags

Jak je to tedy zařízeno se zásobníkem, jestliže ukazatel má jen 8 bitů? Vyšší byte je vždy roven 01h, takže zásobník sídlí na adresách 0100h-01FFh. Ano, má maximálně 256 bajtů. Při zápisu se, stejně jako u 8080, postupuje směrem k nižším adresám. Když dojde na konec, tj. na adresu 0100h, pokračuje se zase od začátku (od 01FFh). Zásobník tedy funguje ve stránce 1.

Princip paměťových stránek jsme si popisovali. Zmiňoval jsem, že u 6800 má paměťová stránka 0 speciální význam – existuje adresovací mód, který místo kompletní 16bitové adresy používá pouze osmibitovou (a horní byte je vždy 00h). Tento mód je, nepřekvapivě, nazván „zero page".

Takže u 6502 máte k dispozici tři vnitřní registry (A, X, Y) a 256 bajtů nulté stránky (v praxi si ale nějakou část této oblasti sebere pro sebe operační systém, takže aplikačním programátorům už moc nezbývá). Tento koncept vychází z možností tehdejší techniky, kdy paměti byly rychlejší než procesory a dobře navržený čip (jako je třeba právě 6502) zvládne na tři takty procesoru přečíst operační kód, přečíst parametr (adresu v zero page) a přečíst nebo zapsat hodnotu z/do paměti. Srovnejme si časování instrukce, která přenese do akumulátoru obsah paměti na adrese 1234h:

- Intel 8080: LDA 1234h – 3 byte, 13 taktů procesoru
- Z80: LD A, (1234h) – 3 byte, 13 taktů
- 6502: LDA 1234h – 3 byte, 4 takty

U nejobvyklejšího časování zvládne 8080 tímto způsobem přenést 141.784 bajtů za sekundu (taktováno na 1,8432MHz), Z80 za tu dobu při stejné frekvenci přenese stejné množství dat, ale 6502 přenese 460.800 bajtů. Omlouvám se, že tu explicitně počítám takovou samozřejmost, ale je na místě ukázat a připomenout, že frekvence procesoru není všechno a nelze říct: V Atari má procesor 1,79MHz, ve Spectru 3,5MHz, je tedy jasné, co je rychlejší (a to ani nezmiňuju, že ve starých počítačích nejedou procesory neustále na *plnej kotel*, občas je okolní systém zpomaluje). Nahradit registry přímo na čipu (které jsou sice nejrychlejší, ale poměrně drahé) rychlým přístupem do paměti byl v sedmdesátých letech docela důvtipný nápad.

Jak vlastně procesor dosahuje takové datové propustnosti? Ukažme si o něco složitější instrukci, totiž LDA 1234h,X – načtení bajtu z adresy 1234h zvýšené o obsah registru X.

- V prvním taktu načte procesor operační kód (a na pozadí možná dokončuje předchozí instrukci).
- V druhém taktu procesor načítá další bajt instrukce, což je nižší část adresy (34h) a zároveň dekóduje operační kód. Z něho pozná, že jde o LDA a adresa se skládá z absolutní šestnáctibitové adresy a indexového registru, tedy že právě načtený byte je nižší část adresy.
- Ve třetím taktu procesor načítá další bajt instrukce, což je vyšší část adresy (12h), a zároveň počítá součet „34h + X".
- Pokud došlo k přenosu, přičte se 1 k vyšší části adresy. Toto přičtení zabere jeden takt. Pokud k přenosu nedošlo, tento takt se vynechá.
- Ve čtvrtém (nebo pátém) taktu se čte obsah paměti na zadané adrese.

LDA s absolutní adresou a indexem trvá čtyři takty, případně pět, pokud výsledná adresa leží za hranicí stránky. V každém cyklu se přistupovalo do paměti. Pokud použijeme procesor CMOS s frekvencí 4MHz, přistupuje se do paměti se stejnou frekvencí, což znamená 250ns na přístupový cyklus. Pomalejší paměti s tím mohou mít problém…

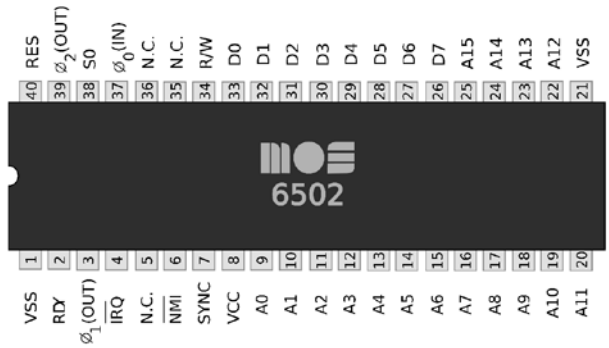
U procesorů 808x nebo Z80 trvá jen načtení a dekódování instrukce 4 takty…

Pro zajímavost – instrukce ADC #51, což je obdoba instrukce ACI 51h (sčítání akumulátoru s konstantou, 7 taktů) se skládá z několika kroků: 1. Vyzvednout operační kód, 2. dekódovat operační kód, 3. načíst operand, 4. sečíst operand a obsah registru A, 5. uložit výsledek do registru A.

Procesor tuto instrukci vykoná reálně ve dvou taktech: v prvním načte operační kód, ve druhém paralelně dekóduje kód a načítá operand. Kroky 4 a 5 dělá zároveň s načítáním operačního kódu další instrukce…

Procesor 6502, stejně jako 6800, nemá speciální systém pro periferie. Návrhář systému musí tedy tyto obvody (klávesnice apod.) namapovat do prostoru paměti. Nevýhodou je, že se tím připravíme o prostor pro paměť, výhodou je, že s periferiemi může programátor pracovat stejně jako s pamětí a využít k tomu libovolnou instrukci, která dokáže číst nebo zapisovat z/do paměti.

8.2 Zapojení 6502



Obrázek 35: 6502 (NMOS verze) (Autor Bill Bertram, CC-BY)

Procesor 6502 má v historii osmibitových procesorů ikonickou pozici. U nás, v bývalém Československu to tak možná nevnímáme, tento procesor se k nám moc nedostal, pouze v dovezených do-

mácích počítačích Atari či Commodore, ale naši konstruktéři, profesionální ani amatérští, s ním moc zkušeností neměli, orientovali se na snadněji získatelné procesory 8080 a Zilog Z80.

Ovšem na západ od našich hranic, v zemích, kde koupit mikroprocesor bylo podstatně snazší, se 6502 dočkal obrovské popularity i mezi amatérskými konstruktéry (a klidně si mezi ně započítejme i pana Wozniaka a jeho počítače Apple). Procesor 6502 je totiž extrémně jednoduchý a konstrukce počítače s ním je snadno pochopitelná.

Jeho programování je pro člověka, zvyklého na svět procesorů od Intelu, trošku překvapivé, ale má své výhody. Kupodivu i zásobník s 256 bajty je pro většinu použití a jazyků dostatečný (zapomeňte ale na předávání parametrů přes systémový zásobník jako v C).

Procesor 6502 se stále vyrábí (respektive jeho vylepšená CMOS varianta 65C02) a prodává a na jeho jádru jsou založené i některé složitější konstrukce, kde slouží např. v roli vnitřního procesoru. V syntetizované podobě, například ve FPGA, může dosahovat pracovních frekvencí v řádech stovek megahertzů a jako řídicí jednotka může být velmi efektivní. Dnešní vyráběné verze používají technologii CMOS a vychází z typu 65C02, který má kromě vyšších frekvencí a nižší spotřeby opravené i některé chyby, které měl původní procesor NMOS, a přidané některé často požadované instrukce, které v původní verzi nebyly.

Asi nejznámější výrobce 65C02 v dnešní době je společnost WDC, založená jedním z návrhářů procesoru 6502 Billem Menschem. Portfolio WDC nabízí kromě procesoru W65C02 i šestnáctibitovou verzi 65C816, a k oběma procesorům dodává i jejich „mikrořadičové" verze – s integrovanou ROM, ve které je Monitor, a několika periferiemi. Navíc vyrábí i několik podpůrných obvodů.

Další výrobce, Rockwell, vyrábí také lehce vylepšenou verzi R65C02, a k nim i verze R65C102 a R65C112. Verze 102 a 112 nabízejí některé vylepšené funkce (DMA), podobně jako verze od WDC.

65C02 má lehce odlišné významy některých pinů od standardní 6502, navíc se liší i různé verze „vylepšených CMOS verzí".

Není 65 jako 02!

Ještě jednu věc je potřeba zdůraznit. 6502 má několik verzí, které se od sebe docela podstatně liší.

- MOS6502 – „původní", originální 6502, použitá např. v Commodore PET nebo KIM-1.
- 6502C – 6502 s vývodem HALT (použitá v počítačích Atari)
- 6510 – 6502 s přidaným portem, využitá v Commodore C64
- 8500 – CMOS verze 6510 (Commodore C64C a C64G)
- 8502 – rychlejší 8500 (až 2 MHz – C128)
- 7501 – HMOS-1 verze 6502 (C16/C116/Plus4)
- 8501 – HMOS-2 verze 6502

Tyto procesory jsou softwarově kompatibilní. Zajímavost je, že procesor, původně vyvinutý jako univerzální společností MOS Technology, se po koupi této společnosti firmou Commodore vyvíjel především tak, aby vyhovoval autorům domácích počítačů od Commodoru.

Na trhu se objevily i další procesory, odvozené z 6502:

- 65C02 – neplést s výše zmíněným 6502C! Tato verze přidala některé instrukce. Tento procesor jsem použil i v OMEN Bravo.
- 65SC02 – zmenšená verze 65C02, která má opět některé instrukce odebrané.
- 65CE02 – rozšířená verze 65C02 (použita v počítači Commodore C65 – tento počítač jste pravděpodobně nikdy neviděli, jejich počet se odhaduje na 50 až 2000 kusů a na eBay se prodával jeden v dubnu 2013 za cenu přesahující 17.000 EUR, tedy cca půlmilion Kč.)
- 65C816 – hybridní procesor, který rozšiřuje 65C02 o šestnáctibitové instrukce.

Tyto procesory jsou „rozšířenou 6502" a chovají se jinak, zejména u „nedokumentovaných" instrukcí (některé z nich mají jiný význam, jiné se chovají jako NOP a nezaseknou celý procesor jako u 6502).

V následujícím popisu se budu věnovat té první skupině, tedy „originál 6502". Odlišné verze probereme až na úplném konci.

8.3 Popis vývodů

Na rozdíl od 8080 nebo 8085 má procesor 6502 vyvedené kompletní sběrnice: datovou, adresní i řídicí.

Datová sběrnice sestává ze signálů D0 – D7, adresní ze signálů A0 – A15.

Systémové hodiny se připojují na vývody Φ1 a Φ0 (jde o řecké písmeno Fí, kterým se někdy značí fáze – nejde o symbol „průměr"). Původní verze NMOS vyžadovala připojení externího oscilátoru, verze 65C02 má mezi vývody Φ0 (vstup) a Φ1 (výstup) připojený invertor, takže stačí, podobně jako u 8085, připojit krystal a kondenzátory. Ovšem pozor u procesorů W65C02 od WDC – tato firma doporučuje v datasheetu použít externí oscilátor a o invertoru se nezmiňuje, ačkoli to vypadá, že je i v nových čipech zapojený. Ale jedná se o nedokumentovanou vlastnost.

Životně důležitý je výstup Φ2 (Fí 2, někdy označovaný i jako PHI2). Tento signál slouží jako systémové hodiny. Každý cyklus procesoru začíná *sestupnou hranou* tohoto signálu (1->0). Během první poloviny cyklu (Φ2 = 0) procesor provádí interní operace a s okolím nekomunikuje. Náběžná hrana Φ2 (0->1) znamená, že procesor je připravený komunikovat, na adresové sběrnici je připravená požadovaná adresa a na datové sběrnici jsou data (pokud procesor hodlá zapisovat). Pokud procesor čte, je datová sběrnice přepnuta na vstup a data se vzorkují se sestupnou hranou Φ2.

Signál R/W říká, jestli procesor bude zapisovat nebo číst. Pokud je 1, procesor čte, pokud 0, procesor zapisuje.

Další důležitý vstup je /RST – jak název napovídá, jde o RESET, aktivní v nule. Můžete použít podobné zapojení, jako jsme použili u 8085 – rezistor a kondenzátor… Ovšem pozor: /RST je aktivní v nule a v procesoru není vstup vybaven Schmittovým klopným obvodem, takže bude potřeba použít hradlo se Schmittovým klopným obvodem (74ALS14). A mimochodem, některé verze starého NMOS procesoru se začaly přehřívat, pokud signál RESET trval příliš dlouho…

Co ty další vstupy? Vss a Vdd je samozřejmě napájecí napětí, Vss odpovídá zemi (GND). Procesor 6502 v NMOS verzi má dva vstupy na zem a jeden na napájecí napětí. CMOS verze od WDC jeden z nich nahrazuje signálem VPB (Vector Pull), který oznamuje, že procesor načítá adresu obslužného vektoru přerušení.

Výstup SYNC říká, že procesor právě načítá instrukci. Pomocí tohoto signálu se snadno zařídí mechanismus „Single step“, tedy že procesor vykoná jednu instrukci a čeká na stisknutí tlačítka.

Vývody NC znamenají „No connect“, tedy „nezapojovat“. Neznamená to nutně, že jsou „not connected“, nepřipojené. Některé z nich mohou sloužit k testování obvodů v závěrečné fázi výroby a uvnitř pouzdra jsou k něčemu připojeny. Ale vy je nezapojujte, prosím.

Vstup RDY slouží k odpojení procesorů od sběrnice, pokud je potřeba provést DMA, nebo pokud musí procesor čekat na pomalou paměť. Některé verze 65C02 má na tomto vstupu slabý pull-up rezistor, takže ho můžete nechat nezapojený; WDC verze tento rezistor nemá. Pokud nemáte s tímto vývodem žádné záměry, prostě jej připojte na 5 V přes rezistor 3k3.

Nejdivnější vstup celého procesoru je vstup /SO. Připojte ho, prosím, na napájecí napětí přes vhodný rezistor a zapomeňte, že existuje. Ne, vážně, udělejte to, protože ho asi nevyužijete. Jeho funkce je taková, že sestupná hrana na tomto vstupu nastaví příznak V v příznakovém registru. Za normálních okolností nechcete, aby vám něco zvenčí měnilo stav příznaků uprostřed práce.

Samozřejmostí jsou dva přerušovací vstupy, /IRQ a /NMI. *Úroveň 0 na vstupu /IRQ vyvolá maskovatelné přerušení, sestupná hrana* na vstupu /NMI vyvolá přerušení nemaskovatelné. Maskovatelné lze zakázat a povolit speciálními instrukcemi.

Verze od WDC W65C02 přidává další dva vývody – vstup BE (Bus Enable) pro odpojení sběrnice a /ML pro uzamčení přístupu do paměti ve víceprocesorových systémech.

Shrňme si to:

- Adresní sběrnice je vyvedená celá
- Datová také

- Procesor používá jednoduché jednofázové hodiny
- Odpadá rozlišení na paměť a periferie
- Procesor dokáže přistupovat k paměti v každém cyklu
- Procesor má zabudovaný jednoduchý přerušovací systém
- Synchronizace všech obvodů signálem Φ2

Co nám brání začít navrhovat vlastní konstrukci? Správně, nic. Pusťme se do toho…

8.4 Základní procesorová jednotka: Hodiny a RESET

Začněme, jako u počítače Alpha, tím, že si zapojíme procesor, hodiny a základní *bižuterii* okolo.

Hodinový obvod můžeme vyřešit stejně jako u 8085, pomocí krystalu a dvou kondenzátorů, ale bude to fungovat pouze u některých procesorů. Použijeme sice CMOS verzi, která by invertor měla mít zabudovaný, ale pokud použijete W65C02 od WDC, nemusí být… Je to nutné ověřit.

Druhá možnost je postavit vlastní oscilátor z invertorů a krystalu.

Třetí možnost je použít oscilátor v pouzdře DIP.

Použijeme stejnou frekvenci jako u Alphy, tedy 3,6864 MHz. Ovšem na rozdíl od Alphy, kde si procesor 8085 podělil frekvenci dvěma, poběží 65C02 na plný výkon. Proto dejte prosím pozor při nákupu, zda kupujete verzi, která je pro takovou frekvenci stavěná.

Procesory od Rockwellu používají označení R65C02Px, kde C značí CMOS verzi, P značí plastové pouzdro se 40 vývody (DIL) a číslice za P udává maximální frekvenci v MHz. Shánějte proto verzi R65C02P4.

U WDC půjde pravděpodobně o procesor W65C02S6P-14. Typ je 65C02S, 6 označuje výrobní technologii, P je pouzdro DIL 40 a -14 je označení rychlosti. Ano, procesor od WDC dokáže pracovat až na frekvenci 14 MHz. Ale můžete narazit i na verzi -6.

Pokud seženete pomalejší procesor (R65C02P2 například) nebo nějaký ještě exotičtější, popřípadě historický kousek, musíte použít odpovídající nižší frekvenci. Tedy třeba 1.8432 nebo 0,9216 MHz. Případně můžete použít krystal s plnou rychlostí, ale za něj zapojit jeden nebo dva klopné obvody D (74HC74) jako děličku frekvence.

Stále vycházím z toho, že by bylo fajn, aby pracovní frekvence šla snadno dělit na nějakou standardní komunikační frekvenci pro sériový port. Ale lze to vyřešit i tak, že pro sériový komunikační port použijete vlastní časovač.

Pokud se rozhodnete *přitopit* a zkusit, co z procesoru vymáčknete, nezapomeňte, že těch 14 MHz znamená, že procesor bude požadovat data z paměti každých cca 70 nanosekund a paměť bude mít na vybavení požadavku čas poloviční, takže musíte vybrat odpovídající rychlé paměti. EEPROM 28C256 od Atmelu mívají přístupové doby okolo 150 ns (popřípadě ještě pomalejší), ty nevyhovují. Ani 28HC256 (High speed) s přístupovou dobou 70 ns pro takové frekvence nevyhoví (mohou teoreticky zvládnout 8 MHz). Paměti FLASH (SST39LF010 například) mají přístupovou dobu 55 ns… Stále málo. Můžete to ale zkusit – a doufat…

Totéž platí i pro paměti RAM. Ty, které používáme (62256 apod.) jsou relativně pomalé. Obvykle mívají přístupovou dobu 70 ns, levnější kousky mají 85 ns, ale vyrábí se i v rychlejších variantách okolo 45 ns. Někteří výrobci nabízí i 10ns verze (např. MOSEL MS62256H-10). Jistější je ale sáhnout například po AS7C256B-15 od Alliance Memory – to je také paměť RAM 32x8, ovšem s rychlostí 15 nanosekund. Ta už vyhoví.

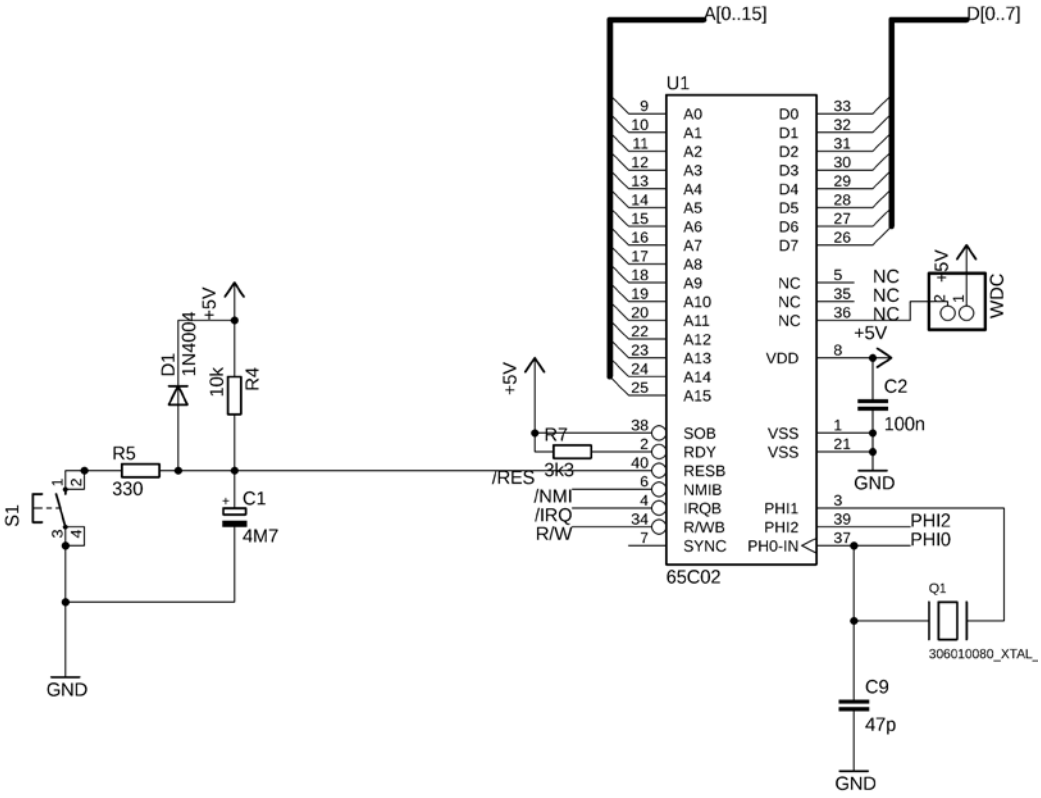
Jak ale vyřešit problém s pomalými paměťmi ROM (EEPROM, FLASH)? Podobně jako se řešily tehdy: pomocí rychlé cache z pamětí RAM. Procesor připravíte tak, aby bylo možné přepínat jeho rychlost, třeba mezi 14 MHz (plný výkon) a 3,5 MHz (čtvrtinová frekvence, pomalý běh). Po startu systému poběží procesor na 3,5 MHz a první úkol po startu bude zkopírovat obsah ROM do rychlé statické RAM. Jakmile bude tohle hotovo, procesor odpojí ROM, zakáže zápis do RAM na místa, kde je kopie ROM, a přepne hodiny na plný výkon.

Druhá možnost je při přístupu na adresy, kde je ROM, procesor brzdit – jakmile logika zjistí, že procesor hodlá přistupovat k ROM, použije signál RDY a drží ho aktivovaný například 100 nanosekund. Procesor pak nechá hodinový signál ve stavu 0, dokud nebude paměť připravená, a teprve po ukončení signálu RDY pokračuje ve čtení.

Mimochodem, slovo „přitopit" jsem použil záměrně. Vyšší frekvence vyžaduje větší proud (WDC udává 1,5 mA na každý 1 MHz) a vyšší napětí. Pokud chcete napájecí napětí snížit, musíte snížit i frekvenci. A samozřejmě platí, že vyzářené teplo je úměrné výkonu (tedy napětí x proud), takže procesor bude na vyšší frekvenci také pěkně hřát a bude možná potřebovat chladič.

U hodin dovolte ještě jednu osobní poznámku. Když jsem si navrhoval pro Bravo plošný spoj, použil jsem zapojení, co jsem si ověřil na nepájivém kontaktním poli. Jenže: nekmitalo to! Změnil jsem tedy celé zapojení, vynechal jsem oscilátor a nechal jsem vše na procesoru, a bylo to OK. Pak jsem v další verzi posunul krystal kousek stranou a pozměnil jsem zalitou zemní plochu. Opět to nekmitalo, až když jsem připojil další kondenzátor a paralelní rezistor. Přitom na nepájivém kontaktním poli identické zapojení fungovalo, dokonce se stejným kusem R65C02. Píšu to sem proto, abych vás na to připravil: je možné, že se vám stane, že stejné zapojení nebude fungovat – většinou u vysokofrekvenčních signálů. Může se to stát. Většinou pomůže přidat kondenzátor, ubrat kondenzátor…

Obvod pro RESET můžeme vlastně zrecyklovat z Alphy, ale s drobným rozdílem: U 65C02 není zaručené, že na vstupu RESET bude Schmittův klopný obvod – datasheety se o něm nezmiňují, ale někteří konstruktéři tvrdí, že tam je – a proto je lepší postavit obvod s nějakým oddělovacím prvkem, který Schmittův KO obsahuje. Další možnost je použít třeba specializovaný obvod (DS1813 od Maxim) nebo časovač 555. Zůstaneme u stejného obvodu, jako měla Alpha, a kdyby byly problémy, zapojte si mezi něj a procesor dva invertory se Schmittovým KO za sebou. (Proč dva? Protože jeden invertor by obrátil signál tak, že by byl stále v 0. Pokud chcete použít jen jeden invertor, musíte předělat *logiku* celého resetovacího obvodu tak, aby byla v klidu v 0 a po invertování v 1.)



Obrázek 36: Bravo, CPU

V zapojení není nic, co by bylo překvapivé. Pozor na vývod 36. U R65C02 a podobných je „not connect“ a nechte jej nezapojený, u W65C02 od WDC je to vstup Bus Enable a je potřeba jej připojit na +5 V.

A ještě jedna terminologická poznámka: ve světě procesorů 65xx se negované signály často označují písmenem „B“ na konci. Proto například u schematických značek najdete vývody RESB nebo NMIB. Znamená to „negovaný RES“, „negovaný NMI“. Já se přidržím zažitější konvence a budu je označovat jako /RES nebo /NMI.

8.5 Další tipy

- http://wilsonminesco.com/6502primer/
- http://wilsonminesco.com/6502primer/ClkGen.html
- http://wilsonminesco.com/6502primer/addr_decoding.html
- https://dave.cheney.net/2014/12/26/make-your-own-apple-1-replica
- http://hackaday.com/2014/06/01/propeddle-the-software-defined-6502/
- http://www.grappendorf.net/projects/6502-home-computer/#table-of-contents
- http://sbc.rictor.org/home.html

8.6 Paměť

Pár poznámek k zapojení paměti

6502 nerozlišuje, jak jsem už psal, mezi pamětí a periferií. Použijete instrukci STA 0C000h a procesor na tuto adresu pošle obsah registru A, aniž by se staral, jestli tam je paměť nebo třeba obvod ACIA.

Ale zas úplná anarchie to taky není. Nesmíme zapomínat, že procesor 6502 má některé věci takříkajíc „napevno“. Zopakujme si je:

- Systémový zásobník, kam se ukládají návratové adresy a další data, má osmibitový ukazatel S. To znamená, že může mít pouhých 256 bajtů. Hodnota v registru S vytvoří dolní polovinu adresy, horní polovina bude vždy 01h. Zásobník tedy sídlí na adresách 0100h až 01FFh a je nezbytně nutné, aby aspoň část tohoto prostoru byla paměť RAM.
- Procesor používá speciální režim se zkrácenou adresou, při kterém se využije jen jeden bajt na adresu a horní polovina je pak rovna 00h. Říká se tomu Zero Page (nultá stránka), a je tedy fajn, když i na adresách 0000h – 00FFh je alespoň někde RAM.

Když procesor startuje nebo když přijde přerušení, skočí procesor na nějakou adresu. U procesoru 8080 / 8085 to byla adresa 0 pro RESET a několik dalších adres pro přerušení. U 6502 to může být libovolná adresa. Ovšem ta adresa musí být uložena v paměti na specifickém místě.

Pro RESET to je FFFCh-FFFDh, pro přerušení IRQ to jsou adresy FFFEh-FFFFh, pro nemaskovatelné přerušení NMI to jsou adresy FFFAh-FFFBh. Ne snad že by procesor při RESETu *skočil na adresu FFFC*, to ne. Procesor si nejprve přečte dvoubajtovou hodnotu z adres FFFCh a FFFDh, a teprve tato hodnota je adresou místa, kam se skáče…

Je tedy dobré, aby na konci adresního prostoru byla paměť ROM – alespoň při startu.

V praxi to dopadá tak, že paměť RAM zabírá dolní část paměťového prostoru, paměť ROM horní část, a periferie „někde mezi tím". Připadá mi nejjednodušší zapojení takové, kde bude mít počítač paměť RAM 32kB od adresy 0000h do 7FFFh, paměť ROM 16 kB (ano, použijeme jen půl kapacity EEPROM) od adresy C000h do adresy FFFFh, a prostor mezi 8000h a BFFFh vyhradíme periferiím.

Dekodér

U Alphy s procesorem 8085 jsme ani žádný pořádný dekodér neměli, ovšem u Brava bude zapotřebí nějak vyřešit to, že v jednom adresním prostoru máme dvě různé paměti, a ještě k tomu periferie.

Už jsem psal, že prostor od 0000h do 7FFFh vyhradíme RAM, prostor od 8000h do BFFFh budou nějaké periferie a C000h až FFFFh bude EEPROM. K dekódování budeme potřebovat tedy dva nejvyšší bity adresy. Výsledkem budiž tři signály: /RAMCS, /ROMCS a /IOCS, které budou vybírat RAM, ROM, respektive porty.

A15	A14	Adresace	/RAMCS	/ROMCS	/IOCS
0	0	RAM	0	1	1
0	1	RAM	0	1	1
1	0	Porty	1	1	0
1	1	ROM	1	0	1

/RAMCS je vlastně ekvivalentní signálu A15, /ROMCS je výsledek A15 NAND A14 (=0, když jsou oba vstupy 1), a konečně /IOCS můžeme zkonstruovat jako /RAMCS NAND / ROMCS. Ale /IOCS nebude potřeba generovat samostatně – pomůže nám v tom dekodér pro periferie.

Signály /RD a /WR

Už jsme si říkali, že 6502 synchronizuje zbytek systému pomocí signálu Φ2. Když je 0, procesor dělá interní operace a externí obvody by neměly komunikovat, když je 1, je čas na komunikaci. Její směr udává signál R/W. Pojďme si tyto dva signály spojit do signálů /RD a /WR:

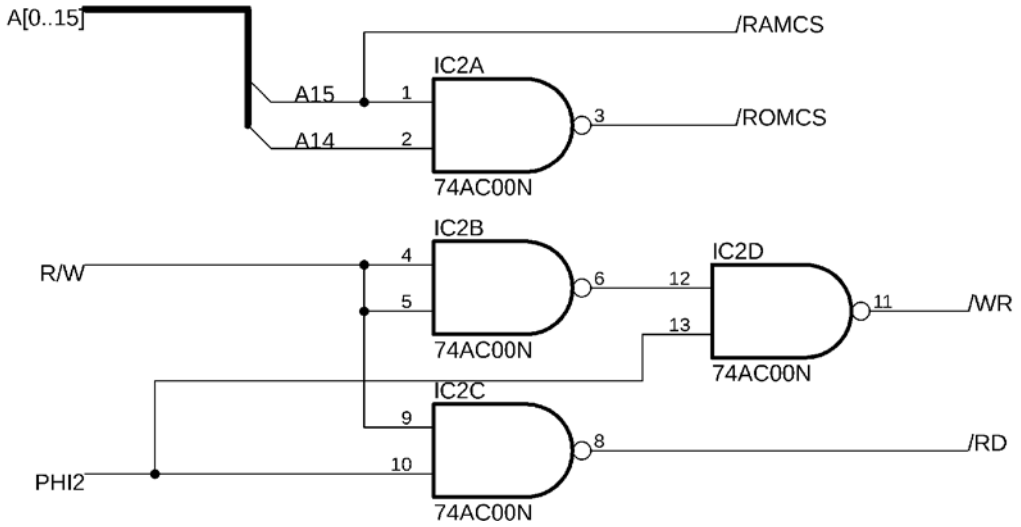
Φ2 (PHI2)	R/W	/RD	/WR
0	0	1	1
0	1	1	1
1	0	1	0
1	1	0	1

Platí tedy, že signál /RD je PHI2 NAND R/W – aktivní v 0, pokud jsou oba vstupy 1. Signál /WR je PHI2 NAND NOT R/W.

Jestli dobře počítám, tak pro paměti potřebujeme:

- 1 hradlo NAND pro signál /ROMCS
- 1 hradlo NAND pro signál /RD
- 1 hradlo NAND pro signál /WR
- 1 invertor pro negaci signálu R/W – a ten uděláme zase pomocí hradla NAND.

Potřebujeme tedy přesně jeden obvod 74x00. *Použijte prosím rychlé obvody HC, HCT, AC, ACT…*



Obrázek 37: Bravo, dekodér

Paměti

Použijeme stejné obvody jako u Alphy: Statickou RAM 32 kB typu 62256 a EEPROM 32 kB typu AT28C256. Stejné zapojení jako minule: /CS se připojí na /ROMCS nebo /RAMCS, /OE jde na signál /RD, /WE na signál /WR.

Z EEPROM využijeme jen půlku, protože paměť má kapacitu 32 kB a my pro ni máme v systému jen 16 kB. Pokud necháme vstup A14 připojený napevno k 0, budeme používat dolní polovinu adres (0000h – 3FFFh), pokud ji připojíme k 1, budeme používat horní polovinu (4000h – 7FFFh). Ideální řešení je přidat přepínač a mít k dispozici dvě různé ROM, mezi nimiž si můžeme před zapnutím počítače vybrat.

Druhá možnost je použít menší obvod EEPROM, třeba AT28C64, který má kapacitu 8 kB. Tento obvod nemá vstupy A14 ani A13, ale jinak má rozložení vývodů stejné. Jediný rozdíl je ten, že pro EEPROM máme vyhrazený prostor o velikosti 16 kB, takže se její obsah objeví v daném rozsahu dvakrát („zrcadlí se"). Zkrátka obsah paměti EEPROM, který je fyzicky v oblasti 0000h – 1FFFh (tedy 8 kB), se v adresním prostoru procesoru objeví od adresy C000h, a stejný obsah i od adresy E000h.

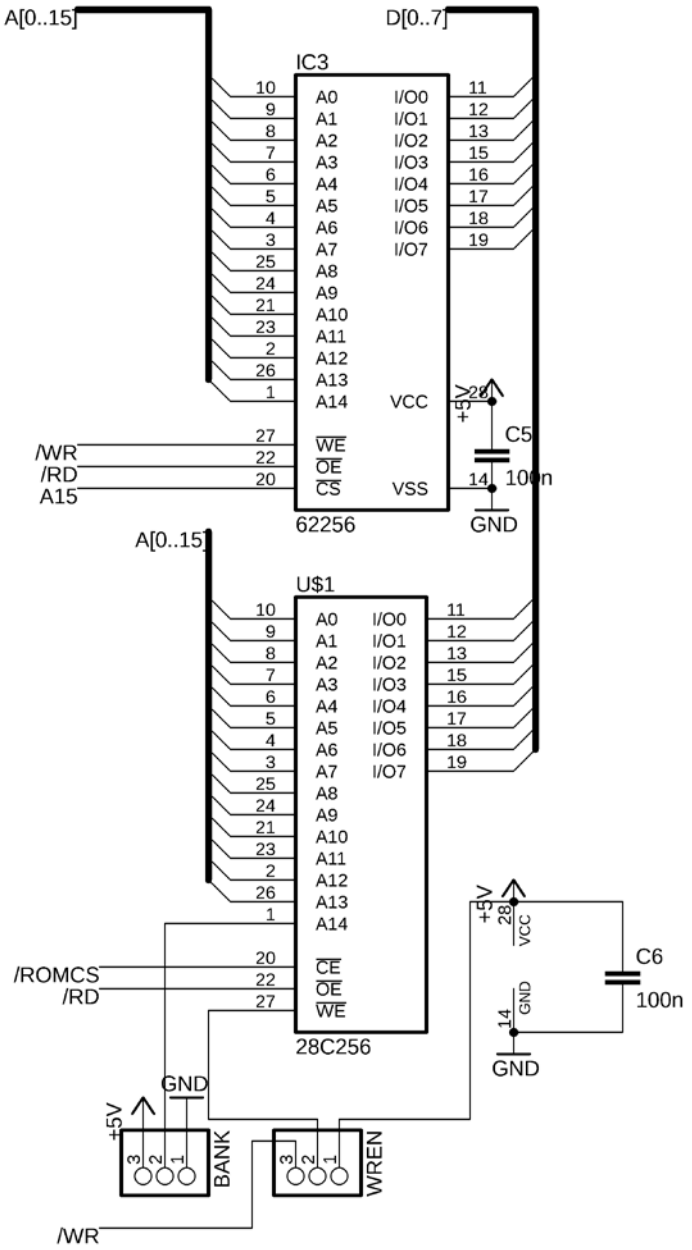
Co s tím? Nejjednodušší je nedělat s tím nic. Tak je EEPROM v paměti dvakrát, a co má být? Ovšem pořádkumilovné jedince to může štvát. Druhá možnost je udělat podrobnější dekodér adres a /ROMCS povolovat pouze pro adresy E000h – FFFFh. Znamená to tedy, že /ROMCS = A13 NAND A14 NAND A15. Budeme potřebovat třívstupové hradlo NAND, anebo zvolíme jednodušší variantu a použijeme obvod 74138, na jehož vstupy A, B a C zapojíme A13 až A15. Osm vývodů pak bude aktivních pro jednotlivé 8kB bloky. Pro /ROMCS pak využijeme /Y7, pro /IOCS třeba /Y6 a pro /RAMCS můžeme využít všechny ostatní (a nabídnout až 48 kB RAM).

Pokud bych slučoval vícero signálů /Yx do jednoho pomocí AND, tak by bylo lepší použít obvod 74156 v konfiguraci „dekodér 1-z-8", který má výstupy s otevřenými kolektory, a prostě je spojit…

Na schématu tedy opět není naprosto nic překvapivého:

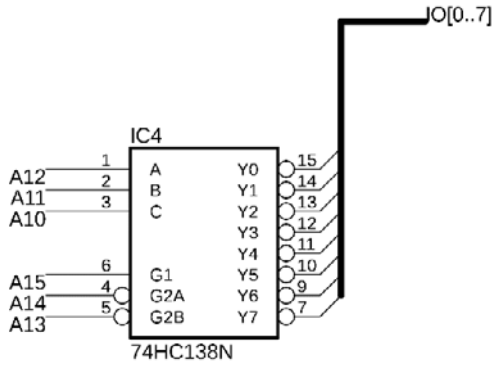
8.7 Periferie

Dekodér pro jednotlivé periferie postavíme opět na obvodu 74138. Na jeho povolovací vstupy G přivedeme signály A13, A14 a A15 tak, že obvod bude vybrán, pokud A15=1, A14=0. Díky tomu ušetříme jedno hradlo NAND pro tvorbu signálu /IOCS. A protože zbývá třetí negující vstup (/G2B), ke kterému je připojen signál A13, znamená to, že periferní obvody budou na adresách, které začínají (binárně) 100x, tedy 8000h – 9FFFh.



Obrázek 38: Bravo, paměti

Na vstupy A, B a C připojíme další adresní signály A12, A11 a A10. Prostor si tak rozdělíme na osm jednokilobytových oblastí (8000h – 83FFh, 8400h – 87FFh atd.)



Obrázek 39: Bravo, dekodér pro periferie

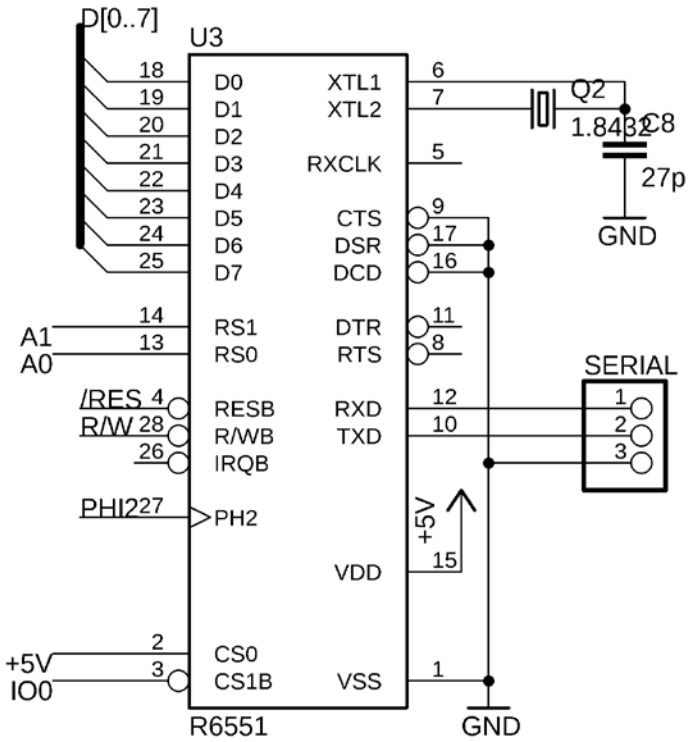
8.8 Sériový port 6551

Na rozdíl od Alphy, kde jsme použili obvod 6850 z rodiny periferních obvodů Motorola 68xx, použijeme v Bravu obvod z řady 65xx, tedy přímo určený pro procesory 6502.

Tento obvod má zabudovaný generátor hodinové frekvence. Pokud k němu připojíte krystal 1.8432MHz, můžete komunikovat až rychlostí 19200 Bd. Obvod má kromě datové sběrnice (D0-D7) a standardních signálů pro sériové rozhraní (TxD, RxD, CTS, DSR, DCD, DTR a RTS) i obvyklou sadu řídicích signálů: chip select (CS0 a /CS1 – použijeme invertující vstup /CS1 a spojíme ho s vývodem dekodéru /IO0), R/W, /RES a PHI2. Tyto signály připojíme k odpovídajícím vývodům procesoru 6502. Vývod /IRQ necháme nezapojený, ale mohli bychom ho spojit se vstupem /IRQ procesoru a využít přerušení pro řízení sériového přenosu.

Poslední dva řídicí vstupy nesou označení „Register select“ – RS0 a RS1. připojíme je na adresní sběrnici, bity A0 a A1.

Obvod ACIA 6551 se tak objeví na adresách 8000h – 83FFh a jeho čtyři vnitřní registry budou na adresách, končících na 0 (4, 8, C), 1 (5, 9, D), 2 (6, A, E) a 3 (7, B, F). Doporučuju opět použít ty adresy, které mají v „nedůležitých“ pozicích 1, tedy 83FCh až 83FFh.



Obrázek 40: Bravo, sériové rozhraní s ACIA 6551

Rozložení registrů pak bude následující:

Adresa	RS1	RS0	Zápis	Čtení
83FCh	0	0	Data k vysílání	Přijatá data
83FDh	0	1	Reset obvodu	Stavový registr
83FEh	1	0	Příkazový (Command) registr	
83FFh	1	1	Řídicí (Control) registr	

Význam dat v jednotlivých registrech potřebuje podrobnější vysvětlení.

Datový registr (83FCh) slouží pro zápis dat k vysílání, popřípadě přečtení dat, načtených ze sériového rozhraní.

Stavový registr (83FDh) je pouze pro čtení. Pokud se pokusíte na tuto adresu zapsat nějaká data, vyvoláte tím reset celého obvodu.

Bit	Název	Význam
0	Parity error	1 = byla detekována chyba parity
1	Framing Error	1 = detekována chyba rámce (start/stop bity)
2	Overrun	1 = chyba přeběhu. Znamená, že přišla nová data po sériovém portu dřív, než procesor načetl předchozí byte
3	Receiver data register full	1 = v datovém registru jsou načtená data a je možno je číst
4	Transmitter data register empty	1 = vysílací registr je prázdný, takže je možné poslat nová data
5	DCD	Stav signálu /DCD
6	DSR	Stav signálu /DSR
7	IRQ	1 = obvod vyvolal přerušení

Příkazový registr (83FEh) řídí některé parametry přenosu, jako je kontrola parity, přerušení nebo stav výstupu DTR.

7 6 5	**4**	**3 2**	**1**	**0**
Parita. 000 = nevysílá se, netestuje…	1 = Echo	Přerušení při vysílání a /RTS (00 = bez přerušení)	Přerušení při příjmu (1 = zakázáno)	/DTR

Do podrobnějšího popisu nechci zabředávat, stačí vědět, že do tohoto registru máme poslat hodnotu 02h (žádné přerušení, žádná parita, žádné echo…)

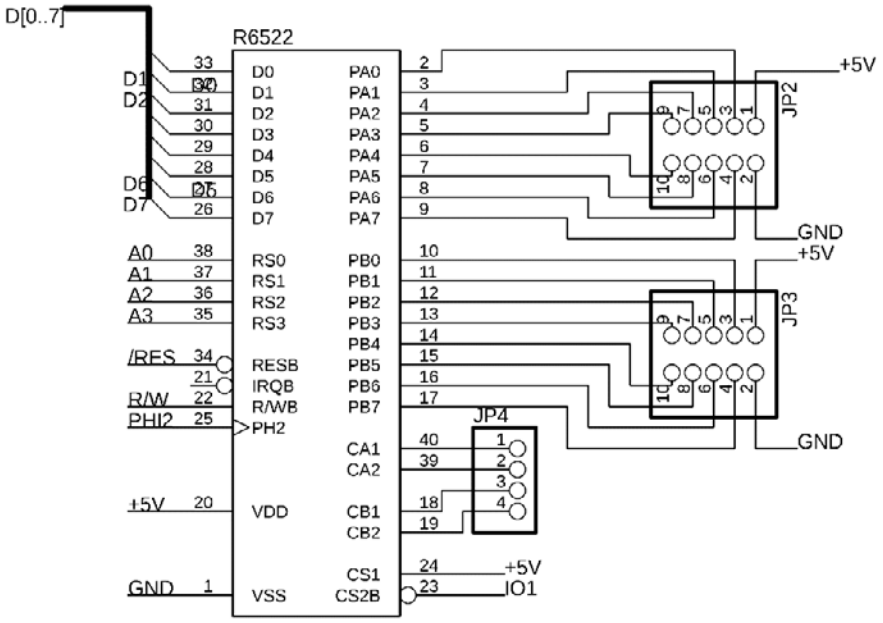
Pomocí řídicího registru (83FFh) nastavujeme další parametry, jako počet stop bitů, délku slov a časování. Bit 7 tohoto registru určuje počet stop bitů (0 = 1 stop bit, 1 = 2 stop bity, popř. 1 pro vysílání s paritou). Bity 6 a 5 udávají délku dat (00 = 8 bitů), bit 4 říká, kde se bere zdroj hodin pro přijímač (0 = externí, 1 = interní) a bity 3 až 0 udávají dělicí koeficienty pro generování přenosové frekvence.

Možné kombinace jsou 0000 (= používá se externí signál na RxC), 0001 (dělitel 1/36864) až 1111 (dělitel 96). S krystalem 1.8432MHz pak vychází přenosová rychlost 50 Bd (0001) až 19200 (1111). Opět doporučuju neexperimentovat a nastavit tento registr na 1Fh (hodiny interní, 19200 Bd, 8 bitů, 1 stop bit).

Úvodní nastavení tedy představuje sérii dvou zápisů: 1Fh na 83FFh a 02h na 83FEh. Tím je nastaveno a můžeme vysílat a přijímat data (83FCh). Nezapomeňte si zkontrolovat, zda můžete vysílat (bit 4 na adrese 83FDh) a jestli je připravený načtený znak (bit 3 na adrese 83FDh).

8.9 VIA 6522

Paralelní port je druhá standardní periferie, kterou k Bravu připojíme. Nebudeme používat stejný obvod Intel 8255, ale sáhneme po další periferii z rodiny 65xx, totiž po obvodu VIA 6522 (VIA: Versatile Interface Adapter). Sice nabízí jen dva osmibitové porty, ale na rozdíl od 8255 je možné každý bit každé brány nastavit nezávisle jako vstupní či výstupní. Zároveň je v čipu posuvný registr, který můžeme použít třeba pro hardwarovou podporu rozhraní SPI. Kromě toho obsahuje i čítač/ časovač, který lze využít jak pro sériové rozhraní, tak pro další činnosti.



Obrázek 41: Bravo, paralelní rozhraní s VIA 6522

Výrazně bohatší výbava oproti obvodu 8255 vyžaduje také komplexnější rozhraní. Obvod VIA proto nabízí rovných 16 registrů. Registry jsou vybírané pomocí signálů RS0 – RS3.

Číslo	Register select				Název	Popis	
	RS3	RS2	RS1	RS0		Zápis	Čtení
0	0	0	0	0	ORB/IRB	Výstup port B	Vstup port B
1	0	0	0	1	ORA/IRA	Výstup port A	Vstup port A
2	0	0	1	0	DDRB	Směr dat na portu B	
3	0	0	1	1	DDRA	Směr dat na portu A	
4	0	1	0	0	T1C-L	T1 latch nižší byte	Čítač T1, nižší byte
5	0	1	0	1	T1C-H	Čítač T1, vyšší byte	
6	0	1	1	0	T1L-L	Latch T1, nižší byte	
7	0	1	1	1	T1L-H	Latch T1, vyšší byte	
8	1	0	0	0	T2C-L	T2 latch nižší byte	Čítač T2, nižší byte
9	1	0	0	1	T2C-H	Čítač T2, vyšší byte	
10	1	0	1	0	SR	Posuvný registr	
11	1	0	1	1	ACR	Pomocný řídicí registr	
12	1	1	0	0	PCR	Registr řízení periferií	
13	1	1	0	1	IFR	Registr příznaků přerušení	
14	1	1	1	0	IER	Registr povolování přerušení	
15	1	1	1	1	ORA/IRA		

Paralelní port

Základní částí obvodu VIA jsou dva osmibitové paralelní porty. K jejich řízení slouží registry ORx/IRx a DDRx – tedy Output Register / Input Register / Data Direction Register. Datové registry slouží k čtení a zápisu hodnot, podobně jako u 8255. Novinkou je DDR. Tento osmibitový registr určuje u každého bitu příslušného portu, jestli je vstupní (=0), nebo výstupní (=1). Pokud tedy do registru DDRA zapíšeme hodnotu (binárně) 00001101, nastavíme tím piny portu A tak, že PA0, PA2 a PA3 budou výstupy, ostatní budou vstupy.

Nabízí se logická otázka: Jakou hodnotu přečteme z datového registru pro bity, které jsou nastavené jako výstupní? Tady se oba porty liší. U portu B to je hodnota, kterou jsme předtím do registru na dané pozici zapsali. U portu A to je hodnota, která je ve skutečnosti na daném vývodu.

Na chování paralelního portu má vliv i nastavení registru PCR. Pomocí tohoto registru se nastavuje chování „handshake“ vývodů Cx1, Cx2 (CA1, CA2, CB1, CB2). Zájemce odkážu na datasheet.

Posuvný registr

Osmibitový posuvný registr používá signály CB1 a CB2. Po CB2 vysílá a přijímá data, po CB1 hodinové pulsy. Hodinové pulsy mohou být buď generovány pomocí časovače T2, popřípadě ze signálu PHI2, nebo externě (pak se CB1 chová jako vstup). Chování posuvného registru řídí bity 2, 3 a 4 registru ACR.

Čítače / časovače

V obvodu VIA jsou dva šestnáctibitové čítače / časovače, nazývané Timer1 (T1) a Timer 2 (T2). Jejich chování řídí pomocný řídicí registr ACR. Obecně čítače fungují tak, že po zápisu 16bitové hodnoty začnou čítat směrem k nule s každou sestupnou hranou hodin PHI1. Aktuální hodnotu čítače lze číst v registrech T1C-L, T1C-H (pro T1) a T2C-L, T2C-H.

U čítače T1 jsou oddělené Jakmile dosáhne nuly, něco se stane – nejčastěji je vyvoláno přerušení.

Latch T1L slouží k zápisu hodnoty, která se použije k opětovné inicializaci čítače. Samotná hodnota je do čítače zapsána po zápisu do vyššího byte.

Čítač T1 může být nastaven tak, že na výstup PB7 posílá signál, který se nastaví do stavu 0 ve chvíli, kdy je v čítači nastavena hodnota, a jakmile dopočítá do 0, tak se PB změní na 1 (takzvaný „one shot“). V módu „free run“ je po dosažení nuly načtena do čítače znovu hodnota z latche a čítá se znovu. Zde lze nastavit, že se hodnota na pinu PB7 při každém dopočítání do nuly změní.

Čítač T2 je o něco jednodušší. Buď funguje jako T1 v módu „one shot“, tzn. počítá hodinové pulsy, odpočítává do 0 a pak vyvolá přerušení. V druhém módu počítá sestupné hrany na vstupu PB6.

Přerušení

V obvodu VIA existuje sedm různých zdrojů přerušení – od pinů CA1, CA2, CB1, CB2, od posuvného registru, když dokončí vysílání / příjem dat, od čítače T1 a od čítače T2. Tato přerušené mohou být zamaskována (registr IER). V registru IFR jsou informace o tom, jaká událost vedla k přerušení. Čtením tohoto registru lze také zjistit stav jednotlivých součástí, tj. jestli čítače dočítaly či posuvný registr dokončil operaci.

Obvod VIA je velmi komplexní. V této knize není dostatek prostoru na podrobný popis jeho programování, zájemce proto odkazuju na datasheet.

9 Základy programování v assembleru 6502

9 Základy programování v assembleru 6502

9.1 Na úvod

Mohl bych zopakovat všechny ty obecně-assemblerové věci, které jsem psal v části o programování v assembleru 8080. Ale pak by kniha zbytečně narostla, a s tím i její cena. Proto budu tiše předpokládat, že jste si přečetli i příslušnou kapitolu o assembleru 8080.

9.2 Adresní módy 6502

Procesor 6502 je, jak jsme si už řekli, velmi intenzivně závislý na práci s pamětí. Jeho instrukční sada není, jako u procesoru 8080, tvořena spoustou instrukcí, které se od sebe liší podle toho, jestli (například) čtou z paměti přímo adresované (LDA), nepřímo adresované dvojicí HL (MOV A,M) nebo nepřímo adresované dvojicemi BC / DE (LDAX). U procesoru 6502 je na toto všechno jediná instrukce, která se jmenuje LDA, a její funkce je: „Do registru A (akumulátoru) zkopíruj hodnotu operandu".

Co je ale hodnotou operandu? To právě určuje onen adresní mód. Podle toho, jaký mód použijeme, tak se vyhodnotí operand. Adresní módy jsou následující:

Immediate (imm)

Tento mód říká, že operandem je ta hodnota, která je na následující pozici za operačním kódem. Instrukce, které používají tento mód, zabírají dva byty (první je operační kód, druhý hodnota)

V assembleru se použití tohoto módu označuje zapsáním znaku # před operand: LDA #23, LDA #0, LDA #23h, LDA #$5A

Implied (imp)

Též „implicitní mód". Instrukce samotná pracuje vždy se stejným operandem – registrem, hodnotou na zásobníku apod. Proto není potřeba žádný operand dodávat.

Absolute (abs)

Operandem je hodnota v paměti, která leží na adrese MMNNh, kde NN je byte, uvedený za operačním kódem na druhé pozici, MM na třetí pozici (opět zápis adresy stylem Little Endian, jako u 8080). Instrukce s absolutním adresováním tedy zabírají tři bajty. V assembleru se zapisuje tento mód plnou adresou: LDA 1234h, LDA $1234

Zero page (zp)

Operandem je hodnota v paměti, která leží na adrese 00NNh, kde NN je byte, uvedený za operačním kódem. Instrukce, které adresují zero page, mají dva bajty. Zapisují se stejně jako předchozí, tedy adresou: LDA 12h, LDA $12

Nojo, ale! Jak překladač pozná, jestli chci použít mód abs nebo zp, když napíšu „LDA 12h“? Co když mám na mysli 0012h a chci plnou adresu? A jak to pozná, když je adresa zadaná symbolicky, tedy „LDA promenna“? Adresa by měla být absolutní, ale když je „promenna“ v nulté stránce, může použít zp… a jak to zjistí, když třeba v tu chvíli hodnota ještě není známá? Řešení jsou různá. Assembler se může řídit podle toho, jak je hodnota zapsaná (0012h je abs, 12h je zp). Dál může použít speciální instrukci pro definici proměnných v zero page (EQU pro normální, EZP pro zp). Může použít speciální zápis, kterým natvrdo řekne, že vyžaduje adresaci ZP (např. pomocí hvězdičky: LDA *promenna). Může využít další průběh, ve kterém optimalizuje instrukce, které používají absolutní mód a jejichž operand je menší než 0100h…

Já jsem v ASM80 použil následující postup: Pokud je hodnota zapsaná přímo, nebo pokud je už dřív v kódu jednoznačně definovaná, tak se pro adresy mezi 0000h a 00FFh použije zero page. Pokud je adresa v tu chvíli ještě neznámá, používá se absolutní mód, ale lze vynutit zero page pomocí zápisu s hvězdičkou.

Absolute indexed (abx, aby)

Operandem je hodnota v paměti. Jeho adresa je získána tak, že se k adrese, zapsané v dvou následujících bajtech (jako u absolutního adresování) přičte hodnota registru X nebo Y. Výsledek je použit jako adresa do paměti. V assembleru se zapíše jako číslo, následované čárkou a znakem X (nebo Y): LDA 1234h,X

Zero page indexed (zpx, zpy)

Operandem je hodnota v nulté stránce paměti. Její adresa je získána tak, že se k bajtu, následujícímu za operačním kódem, přičte hodnota registru X nebo Y. Výsledek se ořízne na osm bitů (pokud tedy vyjde např. 0106h, bude to 06h) a je použit jako adresa v zero page. V assembleru se zapíše jako číslo, následované čárkou a znakem X (nebo Y): LDA 12h,X

Relative (rel)

Operandem je adresa, která je spočítána tak, že se k aktuální hodnotě registru PC přičte hodnota bajtu, který je zapsaný za operačním kódem. Jeho hodnota je brána jako číslo se znaménkem (00 = 0, 07Fh = +127, 080h = -128, 0FFh = -1). Toto adresování se používá u podmíněných skoků.

Indirect (ind)

Nepřímé adresování (indirect) spočívá v tom, že adresa toho místa, o které jde, je uložená v paměti na nějaké úplně jiné adrese, a ta je zadána. Instrukce, které využívají nepřímé adresování, zabírají tři bajty – první je operační kód, druhý je nižší a třetí vyšší bajt nepřímé adresy. Procesor vezme tuto nepřímou adresu (NA) a přečte si obsah dvou buněk (NA a NA+1) z paměti. Tento obsah dá dohromady cílovou (efektivní) adresu.

Nepřímé adresování může použít pouze instrukce JMP. Zapíše se pak např. jako JMP (1234h). V takovém případě vezme procesor obsah na adrese 1234h (řekněme, že to je 0DAh) a na adrese o 1 vyšší, tedy 1235h (řekněme, že tam je 0DEh). Tyto dvě hodnoty dají dohromady adresu 0DEDAh, a na tu se skočí.

Aby nebyly věci tak jednoduché, tak procesor 6502 obsahuje chybu, která způsobuje, že nepřímá adresa, končící na FFh (např. 12FFh), nevezme vyšší část adresy z 1300h, ale z 1200h. Ve skutečnosti totiž pouze přičte ke spodnímu bajtu 1, ale případný přenos do vyšší části adresy ignoruje.

Indexed indirect (izx)

Tento mód se v assembleru zapisuje jako (nn,X) – tedy podobně jako indexovaný mód, ale v závorkách. LDA (12h,X) vezme parametr (12h), k němu přičte hodnotu registru X a získá nepřímou adresu v zero page (podobně jako u zero page indexed). Na rozdíl od „zpx" tím ale nic nekončí – z paměti na nepřímé adrese (nezapomeňte – v zero page!) jsou načteny dva bajty a z nich je složena cílová adresa.

Zůstaňme u instrukce LDA (12h,X) a řekněme, že v registru X je hodnota 3. Co se stane? Procesor vezme parametr 12h a k němu přičte obsah registru X (12h+03h = 15h), aby získal nepřímou adresu do nulté stránky (0015h). Z adresy 0015h načte nižší bajt výsledné adresy, z adresy 0016h vyšší bajt. Nakonec do registru A uloží obsah paměti na této výsledné adrese.

Indirect indexed (izy)

Tento mód je podobný předchozímu, ale liší se v pořadí operací „sečtu" a „přečtu nepřímou adresu". Zapisuje se (nn),Y. Vše osvětlí příklad, řekněme instrukce LDA (12h),Y. Instrukce vezme parametr (12h) a považuje ho za adresu v zero page. Z té načte dva bajty (resp. z adres 0012h a 0013h) a získá tak nepřímou adresu. K ní přičte hodnotu registru Y a výsledek je cílová adresa, ze které se přečte požadovaný bajt do registru A.

Uff.

Já vím. První setkání s adresními módy je pro nezvyklého člověka ukrutná nálož. První půlka ještě jde, ale nepřímé adresování s indexováním je už docela *makačka na představivost*. Nakonec se ale do toho vpravíte, nebojte – i když to není přímočaré jako u 8080.

Bohužel problém je, že tyto adresní módy nejsou úplně ortogonální. Instrukci LDA („ulož do registru A hodnotu") můžete použít s módy IMM (bezprostředně zadaná hodnota), ABS (dvoubajtová adresa), ABX a ABY (dvoubajtová adresa s indexem v X, Y), ZP (jednobajtová adresa do zero page), ZPX (jednobajtová indexovaná adresa do ZP), ZPY… *moment, ZPY použít nelze!* Proč? No, prostě nelze. Zato můžete využít indexovaně-nepřímé IZX a IZY.

Navíc si všimněte, že je (nn,X), ale není (nn,Y) – na druhou stranu je (nn),Y, ale není (nn),X. Indexové registry nejsou tedy plně ortogonální.

Proto si budeme u každé instrukce ukazovat, jaké adresní módy s nimi lze využít.

9.3 Přesuny dat

Instrukční soubor 6502 začneme probírat od instrukcí, které přesouvají data.

Už jsem zmiňoval, že na rozdíl od 8080, kde se liší mnemotechnický zápis instrukcí pro přesuny mezi registry od instrukcí pro přesun z/do paměti (a tam se navíc rozlišuje, zda je adresa uložená v registrech, nebo zapsaná přímo), vystačí si 6502 s několika málo instrukcemi, které přesunou data z/do registru, a to, odkud (nebo kam) se přesouvá, je určeno adresním módem. Základních instrukcí je šest: LDA, LDX, LDY, STA, STX, STY. Trojice LD* (LOAD) přesouvá do registrů (A, X, resp. Y), instrukce ST* (STORE) naopak data z registrů ukládá.

LDA

LDA přesune operand do registru A. Na této instrukci jsme si minule ukazovali adresní módy, její funkce je tedy jasná. Můžeme ji použít s následujícími módy:

- imm: LDA #nn – přesune do registru A přímo hodnotu nn (jednobajtové číslo)
- zp: LDA nn (nebo LDA *nn) – přesune do registru A hodnotu na adrese 00nn
- zpx: LDA nn,X (nebo LDA *nn,X) – dtto jako předchozí, ale k nn je přičten obsah registru X
- abs: LDA nnnn – přesune do registru A hodnotu na adrese nnnn (adresa je 16bitové číslo)
- abx: LDA nnnn,X – dtto jako předchozí, ale k adrese se přičítá obsah registru X
- aby: LDA nnnn,Y – dtto jako předchozí, ale k adrese se přičítá obsah registru Y
- izx: LDA (nn,X) – nepřímé adresování operandu, viz adresní módy
- izy: LDA (nn),Y – nepřímé adresování operandu, viz adresní módy

Všimněte si, že nelze použít mód zpy (tedy zero page s indexací přes Y). Zde je malá ukázka chování těchto instrukcí. Můžete si je vyzkoušet třeba v assembleru ASM80.

https://www.asm80.com

Pokud jste se ještě s online assemblerem ASM80 nesetkali, tak vězte, že kód kliknutím na COMPILE přeložíte, kliknutím na EMULATOR se otevře emulační okno, kde je možné kód procházet a sledovat, jak se mění obsah paměti či hodnoty v registrech.

```
0000 A5 02           LDA *2
0002 A5 01           LDA $0001
0004 A9 65           LDA #$65
0006 A2 02           LDX #2
0008 B5 03           LDA *3,x
```

(V kódu jsem použil instrukci LDX #2, která – analogicky – naplní registr X hodnotou 2)

Jeden zajímavý detail, který může člověka, co přechází ze světa procesorů 8080, zarazit, splést a škaredě překvapit: **instrukce LDA (i LDX, LDY a další) mění příznaky Z a N**, tedy pokud je přenášená hodnota nulová, nastaví Z, pokud je záporná (=nejvyšší bit je 1), nastaví příznak N. O příznacích budeme teprve mluvit, ale tato zvláštnost, tj. že příznaky ovlivní i některé instrukce pro přenos dat, je natolik důležitá, že ji zmiňuju už teď.

LDX

Analogicky k LDA pracuje tato instrukce s registrem X. Oproti LDA můžete použít jen následující adresní módy:

- imm: LDX #nn – přesune do registru X přímo hodnotu nn (jednobajtové číslo)
- zp: LDX nn (nebo LDX *nn) – přesune do registru X hodnotu na adrese 00nn
- zpy: LDX nn,Y (nebo LDX *nn,Y) – dtto jako předchozí, ale k nn je přičten obsah registru Y. *POZOR – nelze použít mód zpx!*
- abs: LDX nnnn – přesune do registru X hodnotu na adrese nnnn (adresa je 16bitové číslo)
- aby: LDX nnnn,Y – dtto jako předchozí, ale k adrese se přičítá obsah registru Y. *POZOR – nelze použít mód abx!*

LDY

Instrukce je obdobná předchozí, ale u indexovaného přístupu zase nedokáže použít obsah registru Y, tj. naopak umožňuje pouze zpx a abx.

- imm: LDY #nn – přesune do registru X přímo hodnotu nn (jednobajtové číslo)
- zp: LDY nn (nebo LDY *nn) – přesune do registru X hodnotu na adrese 00nn
- zpx: LDY nn,X (nebo LDY *nn,X) – dtto jako předchozí, ale k nn je přičten obsah registru X. *POZOR – nelze použít mód zpy!*
- abs: LDY nnnn – přesune do registru Y hodnotu na adrese nnnn (adresa je 16bitové číslo)
- abx: LDY nnnn,X – dtto jako předchozí, ale k adrese se přičítá obsah registru X. *POZOR – nelze použít mód aby!*

STA, STX, STY

Ukládací instrukce, které přenášejí data v opačném směru než jejich LD* obdoby. Dovolují stejné adresní módy jako odpovídající LD instrukce, s jednou výjimkou: nelze použít mód imm. Dává to smysl, protože nelze uložit hodnotu z registru do konstanty. Instrukce STX a STY navíc neumožňují použít mód „absolutní indexovaný“ (abx, aby). Instrukce ST* navíc nemění stav příznaků. Pro úplnost jen doplním seznam možných adresovacích módů:

- STA: zp, zpx, abs, abx, aby, izx, izy
- STX: zp, zpy, abs
- STY: zp, zpx, abs

9.4 Přesuny

Tím jsme si prošli šest základních instrukcí pro přesun mezi registry a pamětí. Ve světě 8080 by jim zhruba odpovídaly instrukce MVI, MOV r,M, MOV M,r. Pro přesun mezi jednotlivými registry je k dispozici série instrukcí T** (Transfer)

TAX, TXA, TAY, TYA

Čtveřice instrukcí, která kopíruje hodnotu mezi akumulátorem a registry X,Y. Adresní mód je implicitní, tj. instrukce sama ví, odkud se má kam co přesouvat a nepotřebuje žádné další informace. TAX přesouvá hodnotu z registru A do X, TAY analogicky do registru Y, TXA přesouvá z registru X do registru A, TYA z registru Y do registru A. Na přímé přesuny mezi registry X a Y instrukce nejsou. Všechny čtyři navíc, podobně jako LD*, ovlivňují příznaky N a Z (o příznacích si povíme za chvíli).

TSX, TXS

TSX vezme hodnotu registru SP (ukazatel zásobníku) a zkopíruje ji do registru X. Přitom nastaví příznaky N a Z podle přenášené hodnoty. TXS naopak přesune hodnotu z registru X do registru S a příznaky nemění.

9.5 Zásobník

Už jsem zmiňoval, že procesor 6502 má ukazatel zásobníku (SP) pouze osmibitový. Adresa v paměti je napevno v první stránce, tedy na adresách 0100h – 01FFh. Zásobník stejně jako u 8080 roste směrem k nižším adresám. Pokud je ukazatel roven 0 a vy uložíte další hodnotu, zapíše se na adresu 0100h a SP se sníží o 1, tedy na FFh. Což znamená, že další ukládání přepíše hodnotu na adrese 01FFh!

PHA, PLA

PUSH A, resp. POP A – PHA uloží hodnotu z registru A do zásobníku, tj. na adresu (0100h+SP) a sníží hodnotu SP o 1. PLA funguje analogicky v opačném směru, tj. zvýší hodnotu SP o 1 a do registru A uloží obsah z adresy (0100h+SP). Navíc nastaví příznaky N a Z.

PHP, PLP

Obdoba předchozích dvou instrukcí, ale nepracuje se s hodnotou registru A, ale s registrem P (příznakový registr). PHP uloží na zásobník obsah příznakového registru, PLP naopak ze zásobníku takovou hodnotu přečte (a, logicky, změní hodnoty všech příznaků).

9.6 Ještě pár slov k přesunům

Probrali jsme instrukce, které u procesoru 6502 přenášejí data. Na jednu stranu mají poměrně bohaté možnosti, na druhou stranu „ne všechno lze použít se vším“ (LDX například dokáže použít

absolutní adresu s indexem Y, STX ne), adresní mód zpy funguje jen u instrukcí LDX, STX (ano, jen u těchto dvou, u žádných jiných se s tímto módem už nesetkáme)… Navíc je potřeba mít na paměti, že instrukce, které přenášejí hodnotu do registru A, X, Y, taky nastavují příznaky N a Z. No a v neposlední řadě dostává ortogonalita na frak u instrukcí přesunů – hodnotu z registru X do registru Y nepřesunete napřímo, do zásobníku můžete uložit jen registr A (a příznak), pokud chcete uložit X, Y, musíte přes registr A, a pokud chcete nastavit hodnotu ukazatele zásobníku, musíte k tomu zase využít registr X, nemůžete použít A (tady bych si tipnul historický vliv předchůdce 6800, kde SP a X byly oba šestnáctibitové).

9.7 Příznaky a instrukce pro práci s nimi

Podobnou roli, jakou má v procesoru 8080 registr F, zastává u 6502 registr P.

BIT	7	6	5	4	3	2	1	0
PŘÍZNAK	N	V	1	B	D	I	Z	C

- N (negative) informuje o znaménku výsledku (nebo přenesených dat). Je-li kladný, je to 0, je-li záporný, je to 1
- V (overflow) značí přetečení čísel se znaménkem (viz dál).
- B (break) je nastaven na 1, pokud bylo přerušení vyvoláno instrukcí BRK
- D (decimal) lze nastavit na 1, pak procesor zpracovává hodnoty v kódu BCD.
- I (interrupt) můžeme nastavit na 1, pokud chceme zakázat přerušení
- Z (zero) je 1, pokud byl výsledek nebo načtený bajt nulový.
- C (carry) se nastavuje na 1, jestliže došlo k přetečení ze 7. bitu

Pojem „přenosu“ jsme si vysvětlovali v kapitole o příznacích 8080 (doporučuju přečíst, i když jde o jiný procesor). U 6502 je potřeba věnovat pozornost příznaku V, který nemusí být zcela jasný.

Některé popisy se omezují na málo říkající a nepřesný „přenos ze 6. bitu“. Jiné popisy vysvětlují, že se jedná o XOR mezi přenosem ze 6. bitu a ze 7. bitu, což je technicky možná OK, ale neříká, co to vlastně znamená. Pojďme si to vysvětlit názorněji.

Příznak V říká, jestli došlo k přetečení čísla se znaménkem. Představme si, že sečteme dvě čísla: 127 a 1 (hexadecimálně 7Fh a 01h). Výsledek je 128 (tedy 80h). Pokud bychom ale používali aritmetiku se znaménkem, tak zjistíme, že 127 + 1 = -128, a to je špatně! Příznak V nás upozorňuje, že došlo k něčemu takovému, tj. že výsledek je mimo rozsah <-128;127>.

Při sčítání FFh a 01h sice dojde k normálnímu přetečení (C), ale z hlediska čísel se znaménkem se vlastně sčítalo „-1 + 1“ a výsledek je 0, bez přetečení. V tedy bude 0.

Při sčítání 80h a FFh bude výsledek 7Fh. U čísel bez znaménka došlo k přetečení (128 + 255), u čísel se znaménkem (-128 + -1) taky. Budou tedy nastaveny příznaky C i V.

Demonstrační kód si můžete vyzkoušet opět v emulátoru. Instrukce CLC slouží k nulování příznaku C a ADC sčítá dvě čísla (6502 má pouze instrukci sčítání s příznakem C, proto je ho potřeba nejprve nulovat, ale k tomu se ještě dostaneme). Můžete si vyzkoušet chování; sledujte hlavně stav bitů V a C.

```
0200                .ORG 200h
0200 18             CLC
0201 A9 01          LDA #$01
0203 69 01          ADC #$01
0205 18             CLC
0206 A9 01          LDA #$01
0208 69 FF          ADC #$FF
020A 18             CLC
020B A9 7F          LDA #$7F
020D 69 01          ADC #$01
020F 18             CLC
0210 A9 80          LDA #$80
0212 69 FF          ADC #$FF
```

Další bit v příznakovém registru, který zaslouží vysvětlení, je bit I. Pokud je tento bit roven 1, je zakázáno („zamaskováno“) přerušení a procesor nereaguje na signál, přivedený na přerušovací vstup IRQ (6502 má dva druhy přerušení, maskovatelné IRQ a nemaskovatelné NMI, ale k nim se ještě dostaneme). Příznak může na hodnotu 1 nastavit programátor instrukcí SEI, popřípadě procesor poté, co přišel požadavek na přerušení – tím se zabrání, aby bylo vyvoláno přerušení dřív, než skončila obsluha předchozího.

Příznak B označuje, že obsluha přerušení byla vyvolána instrukcí BRK, nikoli vnějším signálem IRQ. Instrukce návratu z obsluhy přerušení jej opět nuluje.

Příznak D může programátor nastavit na 1 a tím vynutit, aby procesor pracoval v režimu BCD – tedy jako by po každé operaci sčítání a odčítání prováděl dekadickou korekci.

9.8 Instrukce pro práci s příznakovým registrem

Příznakové bity nastavují různé instrukce v rámci své normální činnosti (většinou aritmetické, logické nebo instrukce přenosu dat), ale existuje i sada instrukcí pro nastavení či nulování konkrétních bitů.

CLC, SEC
Instrukce nuluje (CLC – CLear Carry) nebo nastavuje (SEC – SEt Carry) příznak C

CLD, SED
Instrukce nuluje (CLD) nebo nastavuje (SED) příznak D

CLI, SEI
Instrukce nuluje (CLI) nebo nastavuje (SEI) příznak I

CLV
Instrukce nuluje příznak V. (Vidíte správně, žádná instrukce SEV není.)

Tyto instrukce nemají žádný parametr (je tedy použit „implicitní mód“).

9.9 Přerušovací systém

Nejen o přerušení, ale také o tom, co se děje, když zapnete napájení.

Napětí je připojeno, hodinový takt běží, procesor 6502 začíná pracovat. Co udělá ze všeho nejdřív? *Správnou odpověď do vzkazů, pokud neuhodnete, musíte si dát tuto hádanku na svůj Facebook!* Promiňte, samozřejmě to není hádanka a nic si na Facebook dávat nemusíte. Řekneme si to hned teď. Ale začneme přerušením.

Procesor 6502 má, stejně jako jiné procesory, k dispozici přerušovací systém. Princip přerušení jsme si už popisovali u procesoru 8080: v situaci, kdy je potřeba zareagovat (např. přišel kompletní načtený znak z terminálu) si vyžádají okolní obvody pozornost procesoru přerušovacím signálem. Procesor k takovému účelu má extra přerušovací vstup (někdy víc), a zareaguje tak, že uloží svůj stav na zásobník a provede určitou instrukci, která většinou způsobí skok do podprogramu.

Přerušení u 8080 je maskovatelné, to znamená, že pomocí instrukce může programátor zakázat, aby procesor na přerušení reagoval (obvykle v časově kritických místech).

U 6502 jsou dva přerušovací vstupy. Jeden z nich je maskovatelný (IRQ) – signál na tomto vstupu vyvolá přerušení pouze v případě, že není nastaven příznak I. Druhý přerušovací vstup je nemaskovatelný (NMI, Non-Maskable Interrupt). Signál na tomto vstupu vyvolá přerušení vždy.

Co se stane, když systém vyvolá přerušení? Procesor v tu chvíli uloží na zásobník hodnotu registru PC (nejprve vyšší, potom nižší bajt), pak uloží rovněž na zásobník hodnotu příznakového registru P a pak skočí na adresu obsluhy přerušení.

A tu zjistí kde přesně? Správná otázka. Pokud šlo o maskovatelné přerušení IRQ, tak si ji přečte na adresách FFFEh a FFFFh, tedy na posledních dvou adresách adresního prostoru. Pokud vás trápí otázka „a jak se tam ta adresa dostane?“, odpověď zní: To záleží na návrháři systému. Pokud je tam pevná paměť (ROM/PROM/EPROM/atd.), je v ní adresa obsluhy přerušení („přerušovací vektor“) uložená napevno. Když je tam RAM, zapsal si ji tam programátor.

Pokud šlo o nemaskovatelné přerušení NMI, přečte si adresu obslužné rutiny na adresách FFFAh a FFFBh.

Zajímá vás, co je mezi tím? Na FFFAh a FFFBh je adresa obsluhy NMI, adresa obsluhy IRQ je na FFFEh a FFFFh, zbývá volný prostor FFFCh a FFFDh… Tam je adresa RESETu.

Ano, čtete dobře. Po zapnutí napájení nebo po přivedení signálu RESET se procesor nenastavuje do nějakého definovaného stavu, on prostě jen skočí na adresu, která je zapsaná v buňkách FFFCh a FFFDh. Jediný rozdíl proti přerušení (viz výše) je v tom, že RESET neukládá PC a P na zásobník.

Vzhledem k tomu je potřeba, aby na těchto adresách byla při startu systému smysluplná adresa. Nemůžeme spoléhat na to, že ji tam zapíše programátor, takže je to potřeba buď vyřešit tím, že na konci paměťového rozsahu je paměť ROM, nebo nějakým obvodovým hackem, který po startu „podvrhne“ procesoru tu správnou adresu.

Instrukce RTI

Instrukce RTI – Return from Interrupt doplňuje přerušení, jak jsme si popsali výše. Provádí přesně opačné kroky, tj. ze zásobníku načte obsah registru P, pak nižší a vyšší bajt registru PC. Postará se tedy o správný návrat z rutiny přerušení.

Jednoduchý příklad v assembleru: program uloží do registru A hodnotu 3 a tu pak zapíše do nulté stránky na adresu 0. Zápis pak probíhá stále dokola – můžete si ověřit pomocí krokování. Pokud kliknete na IRQ, procesor si odskočí do obsluhy přerušení, která změní obsah registru A. Po návratu do nekonečné zapisovací smyčky se už tedy bude zapisovat jiná hodnota.

```
0200                    .ORG $200
0200                    .ENT $
0200            START:
0200 A9 03              LDA #3
0202 85 00      LOOP:    STA 0
0204 4C 02 02           JMP loop
0207            INT:
0207 A9 05              LDA #5
0209 40                 RTI
```

```
FFFC                .ORG $fffc
FFFC 00 02          DW start
FFFE 07 02          DW int
```

Všimněte si, že tentokrát program začíná na adrese 0200h a od adresy FFFCh jsou zadány hodnoty startovací adresy a přerušovací rutiny.

Při krokování si všimněte, že při provádění obsluhy přerušení je nastaven příznak I.

Instrukce BRK

Co se stane, když procesor provede instrukci BRK? Uloží na zásobník hodnotu registru PC (nejprve vyšší, potom nižší bajt), pak uloží na zásobník hodnotu příznakového registru P a pak skočí na adresu obsluhy přerušení IRQ, kterou si přečte na adresách FFFEh a FFFFh.

Možná vám to připadá povědomé… Ano, přesně totéž se děje při maskovatelném přerušení IRQ! Není to náhoda.

Ve skutečnosti je procesor 6502 zapojen tak, že přerušovací požadavek po kontrole příznaku I uloží do registru, kam si načítá kód další instrukce, operační kód instrukce BRK (který je, čistě pro zajímavost, roven 00h). Takže se opravdu provádí to samé – s jedinou výjimkou: samotná instrukce BRK před skokem na obsluhu ještě nastaví příznakový bit B, při obsluze přerušení se tato fáze nastavování bitu B přeskočí. Podle stavu bitu B lze poznat, jestli obsluhu přerušení vyvolal vnější systém (B=0) nebo instrukce BRK (B=1).

Pro zájemce jen dodám, že stejně funguje i obsluha NMI, která rovněž podvrhne BRK, ale změní i adresy vektoru, a v zásadě i signál RESET, který ale místo „ukládání do paměti“ při práci se zásobníkem aktivuje signál „čtení z paměti“ – více o těchto vnitřních zajímavostech naleznete v článku Internals of BRK/IRQ/NMI/RESET on a MOS 6502. Z tohoto článku ocituju i souhrnnou tabulku, co se děje v procesoru 6502 při přerušení a instrukci BRK:

PŘÍČINA	VEKTOR	UKLÁDÁ PC A P NA ZÁSOBNÍK?	NASTAVUJE PŘÍZNAK B?
signál NMI	$FFFA/$FFFB	ano	ne
signál RESET	$FFFC/$FFFD	ne	ne
signál IRQ	$FFFE/$FFFF	ano	ne
instrukce BRK	$FFFE/$FFFF	ano	ano

9.10 Skoky a podprogramy 6502

V minulé kapitole jsem použil v kódu instrukci skoku a popisoval jsem návrat z přerušení. Pojďme si tedy doplnit sérii a probrat zbývající instrukce skoků.

Nepodmíněný skok – JMP

Instrukce nepodmíněného skoku dělá přesně to, co u jiných procesorů – tedy to, co se označuje známým „GOTO“. Skočí se na jinou adresu a pokračuje se odtamtud.

Instrukce JMP používá dva adresní módy – buď absolutní adresování, nebo nepřímé. Absolutní znamená, že se skáče přímo na zadanou adresu:

```
0200                    .ORG 200h
0200                    .ENT $
0200 A9 00              LDA #0
0202 69 01      LOOP:    ADC #1
0204 4C 02 02           JMP loop
FFFC                    .ORG 0fffch
FFFC 00 02              DW 200h
```

Nepřímé adresování (viz kapitola o adresních módech) pracuje tak, že ze zadané adresy (a z adresy o 1 vyšší) se načtou dva bajty, které dohromady dají dvoubajtovou *efektivní adresu* a skáče se na ni.

```
0200                    .ORG 200h
0200                    .ENT $
0200 A9 00              LDA #0
0202 69 01      LOOP:   ADC #1
0204 6C 07 02           JMP (ind)
0207 02 02      IND:    DW loop
FFFC                    .ORG 0fffch
FFFC 00 02              DW 200h
```

Všimněte si, že v tomto druhém případě neskáče JMP přímo na adresu LOOP, ale na (ind). ind je návěští, na kterém jsou uložené dva bajty (viz výpis přeloženého programu).

Podmíněné skoky – Bxx

Podmíněných skoků je osm pro osm různých podmínek – podle čtyř příznakových bitů, vždy 0 nebo 1. Zde jsou v přehledné tabulce:

PŘÍZNAK	=0	=1
N	BPL	BMI
V	BVC	BVS
C	BCC	BCS
Z	BNE	BEQ

Mnemotechnika těchto názvů je prostá. Instrukce jsou skoky (Branch), z toho je písmeno B. Další dvě písmena jsou název podmínky – u příznaku N (Negative), který říká, jestli je číslo kladné nebo záporné, je to BPL (Branch if PLus) a BMI (Branch if MInus). U příznaku Z, který udává, jestli je číslo nula, to není Zero-Nonzero, ale využívá se toho, že tento skok bývá často prováděn po testu na rovnost (který interně probíhá jako odčítání). Při nerovnosti, nenulovém výsledku (Z=0) se skáče instrukcí BNE (Branch if Not Equal), analogicky při rovnosti, a tedy Z=1, se skáče instrukcí BEQ (Branch if EQual).

U příznaků C a V, které nemají takhle jednoznačné „vysvětlení“, se používá mnemotechnika „Branch if V is Clear“ (BVC), „Branch if C is Set“ (BCS) apod.

Adresní mód těchto instrukcí je vždy relativní. To znamená, že se skáče v rozmezí -128..+127. Využívá se toho, že podobné podmíněné skoky jsou většinou součástí krátkých smyček. Pokud tomu tak není, musíte použít „náhražkovou“ konstrukci:

```
BCC NekamDaleko ; nelze, pokud je adresa vzdálená
                ; víc než 128 pozic
                ; Místo toho je třeba použít:
BCS Skip        ; Obrácená podmínka, která přeskočí
                ; následující instrukci
JMP NekamDaleko ; tady se provede samotný skok
Skip: ….      ; a tady se pokračuje
```

Překladač naštěstí za vás spočítá správnou hodnotu odskoku z aktuální adresy návěští a cílové adresy. Hodnota 0 znamená následující adresu (tj. žádný efekt), hodnota 0FEh skáče o dvě místa zpátky, tj. na tu samou adresu, kde je instrukce (je to hodnota -2, a Bxx jsou dvoubajtové).

BRK, RTI

Tyto instrukce jsme probrali v minulé kapitole – slouží pro vyvolání přerušení a pro návrat z obslužné rutiny přerušení.

Podprogramy – JSR, RTS

To, k čemu u 8080 sloužily instrukce CALL a RET, zajišťují u 6502 instrukce JSR a RTS (Jump to Subroutine / Return from Subroutine). JSR používá pouze absolutní adresní mód, tj. za instrukcí jsou nižší a vyšší bajty cílové adresy. JSR uloží na zásobník vyšší a nižší bajt návratové adresy, a pak do PC nahraje přečtenou adresu. Čímž se vlastně provede skok na nějakou adresu (jako u JMP), s tím rozdílem, že na zásobníku je adresa, kam se má program vrátit (ve skutečnosti je o 1 menší, s čímž počítá instrukce RTS).

K návratu slouží instrukce RTS. Ta přečte ze zásobníku dva bajty, z nich složí adresu, přičte 1 (viz výše) a na ni skočí. Pokud podprogram zanechal zásobník v takovém stavu, v jakém ho našel, tak se skočí na instrukci, následující za příslušnou instrukcí JSR.

Rozdíl mezi RTS a RTI je v tom, že RTI načítá ze zásobníku i uloženou hodnotu příznakového registru P a k návratové adrese nepřičítá 1 (instrukce BRK i přerušení ukládají pravou návratovou adresu).

Úhrnem lze o instrukční sadě procesoru 6502 v souvislosti se skoky říct, že je hodně omezená. Podmíněné skoky pouze relativní, skoky do podprogramu a návraty pouze nepodmíněné a s absolutní adresou, jen nepodmíněný skok lze adresovat i nepřímou adresou (ale nelze relativně).

9.11 Aritmetika 6502

V této kapitole konečně donutíme procesor 6502 něco spočítat.

Když jsem v minulých kapitolách naznačoval, že to s ortogonalitou instrukční sady procesoru 6502 není, ani přes velké množství adresních módů, moc slavné, tak věřte, že jsem si to nejhorší šetřil až na závěr.

INC, DEC

Nejjednodušší aritmetické instrukce jsou inkrement a dekrement, tedy přičtení jedničky a odečtení jedničky. 6502 má k tomu účelu instrukce INC a DEC. Tyto instrukce zvýší (INC) nebo sníží (DEC) obsah paměťové buňky o 1. Můžete je použít s následujícími adresními módy:

- abs: INC 1234h – zvýší obsah buňky na adrese 1234h o 1
- zp: INC 12h – zvýší obsah buňky na adrese 0012h o 1
- abx: INC 1234h,X – zvýší obsah buňky na adrese (1234h + X) o 1
- zpx: INC 12h,X – zvýší obsah buňky v nulté stránce paměti na adrese (12h+X) o 1 (nezapomeňte, že nultá stránka má vždycky horní byte adresy rovný 0, pokud tedy bude v X hodnota FFh, nebude se pracovat s adresou 0111h, ale 0011h!)

Totéž pro instrukci DEC.

INX, INY, DEX, DEY

Obdoba instrukcí INC, DEC, ale místo obsahu paměti se pracuje s registry X (INX, DEX) a Y (INY, DEY).

Instrukce inkrementu a dekrementu nastavují podle výsledku operace příznaky N a Z.

Možná jste si všimli, že jsem v seznamu neuvedl instrukce, které inkrementují/dekrementují obsah akumulátoru A. Neuvedl jsem je, protože je procesor 6502 nemá.

ADC, SBC

Sčítání a odčítání 6502 samozřejmě obsahuje. ADC (Addition with Carry) přičte parametr a hodnotu příznaku C k registru A a výsledek ponechá v registru A. SBC (Subtraction with Carry) odečte od obsahu registru A parametr a negaci příznaku C a výsledek uloží do registru A.

Co z toho vyplývá? Zaprvé: 6502 vždycky uvažuje stav příznaku C (přenos). Neexistuje instrukce pro sčítání nebo odčítání, která by jeho stav ignorovala. Pokud chceme „jen" sčítat dvě čísla, je potřeba předtím nastavit C na nulu instrukcí CLC, jinak může být výsledek o 1 vyšší. Pokud výsledek sčítání přeteče 255, bude C=1, jinak zůstane nulový.

U instrukce pro odčítání platí přesný opak – pokud chceme zanedbat přenos, musíme příznak nastavit na 1 (instrukcí SEC). Pokud výsledek při odčítání podteče nulu (výsledkem je záporné číslo), bude příznak C roven 0, jinak 1.

A tak se může stát, pokud neošetříte příznaky správně, že dvojice instrukcí ADC #1, SBC #1 ve skutečnosti dělají věci nečekané. Viz následující kód – sledujte instrukce a výsledek v registru A:

```
0000 A9 08           LDA #8
0002 E9 01           SBC #1
0004 69 01           ADC #1
0006 38              SEC
0007 E9 01           SBC #1
0009 69 01           ADC #1
000B A9 00           LDA #0
000D E9 01           SBC #1
000F 69 01           ADC #1
```

Adresní módy těchto instrukcí jsou stejné jako např. u instrukce LDA, tedy:

- imm – přímý operand: ADC #1 přičte 1
- abs, zp – přímo zadaná adresa, buď plná, nebo v zero page
- abx, aby – absolutní adresa, zvýšená o obsah registru X či Y

- zpx – adresa v zero page, indexovaná přes registr X
- izx, izy – nepřímo adresovaný operand

Porovnání - CMP

Instrukce CMP porovná hodnotu v registru A s operandem. Vnitřně funguje tak, že od hodnoty v registru A odečte hodnotu operandu, podle výsledku nastaví příznaky N, Z a C a výsledek zahodí.

Mohou nastat tři situace, které si ukážeme v následující tabulce:

SITUACE	N	Z	C
A = operand	0	1	1
A > operand	0	0	1
A < operand	1	0	0

(Hodnoty uvažujeme jako čísla bez znaménka)

Instrukce CMP nabízí stejné adresační možnosti jako instrukce ADC či SBC.

```
0000 A9 08           LDA #8
0002 C9 08           CMP #8
0004 C9 07           CMP #7
0006 C9 09           CMP #9
```

CPX, CPY

Podobně jako existují obdoby instrukcí INC a DEC pro práci s registry X a Y, tak i CMP má obdoby CPX a CPY. Liší se od CMP tím, že neporovnávají operand s hodnotou registru A, ale s registrem X, resp. Y. CPX a CPY mají jen tři adresní módy: přímý operand (imm), absolutní adresa (abs) nebo adresa v nulté stránce (zp).

9.12 Logické a bitové operace 6502

Blížíme se ke konci, zbývá doprobrat už jen pár instrukcí, konkrétně logické operace a manipulace s bity.

AND, ORA, EOR

Trojice instrukcí pro základní bitové operace – and, or, xor (exclusive or) se u procesoru 6502 jmenují AND, ORA a EOR. Provedou danou logickou operaci s obsahem registru A, výsledek uloží do A a nastaví příznaky N a Z.

Všechny tři instrukce mají poměrně bohaté možnosti adresování:

- imm – přímý operand: AND #1 provede operaci A = A & 01
- abs, zp – přímo zadaná adresa, buď plná, nebo v zero page
- abx, aby – absolutní adresa, zvýšená o obsah registru X či Y
- zpx – adresa v zero page, indexovaná přes registr X
- izx, izy – nepřímo adresovaný operand

```
0000 A9 55            LDA #55h
0002 49 FF            EOR #0ffh
0004 A2 05            LDX #5
0006 35 FB            AND -5,x
0008 15 FD            ORA -3,x
```

Rotace – ROL, ROR

Instrukce ROL a ROR rotují bitově obsah registru A nebo paměti (ROL doleva, ROR doprava). Při rotaci se rotuje přes příznak C.

ROL posune bity o 1 doleva. To znamená, že bit 0 se přesune na pozici 1, bit 1 na pozici 2, bit 2 na pozici 3 a tak dál, a bit 7, který nám vypadne zleva ven, je zapsán do příznaku C, a původní hodnota z C je přesunuta do pozice 0 v registru A. ROR funguje stejně, jen obráceným směrem. Graficky to vypadá nějak takto:

Operace	Pozice v registru A								Příznak
	7	6	5	4	3	2	1	0	CY
Výchozí stav	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	CY
ROL	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	CY	Bit 7
ROR	CY	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0

Instrukce ROL a ROR nabízejí opět několik adresních módů. Bez operandu pracuje s obsahem registru A. Pokud chcete pracovat s obsahem paměti, můžete buňku adresovat buď absolutně (abs, 2 bajty adresa), v nulové stránce (zp, 1 bajt), nebo indexovaně (abx nebo zpx).

Posuny – ASL a LSR

Posuny se od rotací liší v tom, že „vypadnuvší" bit je přesunut do příznaku C, ale původní hodnota tohoto příznaku je zahozena a místo ní vstoupí zpět hodnota 0. ASL posouvá doleva, zprava dopl-

ní 0, LSR posouvá doprava, zleva doplní 0. Adresní módy jsou stejné jako u rotací – bez operandů se pracuje s registrem A, pokud chcete pracovat s pamětí, můžete použít abs, zp, abx nebo zpx.

Matematicky odpovídá posun doleva vynásobení hodnoty dvojkou, posun doprava pak celočíselnému dělení 2 (zbytek je v příznaku C). Grafické znázornění zde:

Operace	**Pozice v registru A**								**Příznak**
	7	**6**	**5**	**4**	**3**	**2**	**1**	**0**	**CY**
Výchozí stav	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	CY
ASL	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	0	Bit 7
LSR	0	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0

```
0000 A9 55          LDA #55h
0002 26 01          ROL 1
0004 26 01          ROL 1
0006 26 01          ROL 1
0008 26 01          ROL 1
000A 66 01          ROR 1
000C 66 01          ROR 1
000E 66 01          ROR 1
0010 66 01          ROR 1
0012 0A             ASL
0013 0A             ASL
0014 0A             ASL
0015 0A             ASL
0016 0A             ASL
0017 4A             LSR
0018 4A             LSR


0019 4A             LSR
001A 4A             LSR
001B 4A             LSR
001C 4A             LSR
001D 4A             LSR
001E 4A             LSR
```

Instrukce BIT

Instrukce BIT provede logický součin (AND) obsahu registru A a operandu, který je adresován buď absolutní adresou (abs), nebo nulovou stránkou (zp). Výsledek je zahozen, ale předtím je podle něj nastaven stav příznaku Z. Je-li tedy výsledek 0, je Z=1.

Instrukce BIT ještě nastaví příznaky N a V – zkopíruje do nich šestý a sedmý bit operandu (bit 7 do příznaku N, bit 6 do příznaku V).

9.13 Algoritmy sčítání a násobení pro 6502

16bitový součet

U procesoru 8080 není sčítání 16bitových čísel problém – procesor má dostatek registrů a má k tomu i speciální instrukci DAD. U 6502 nemáme ani instrukci, ani registry. Proto se musí sčítání dvoubajtových čísel řešit algoritmem. Naštěstí není příliš složitý, využívá jen instrukci ADC.

```
0000 18              CLC
0001 A5 60           LDA *P1
0003 65 62           ADC *P2
0005 85 64           STA *R
0007 A5 61           LDA *P1+1
0009 65 63           ADC *P2+1
000B 85 65           STA *R+1
0060                 .ORG 60h
0060 34 12     P1:   DW 1234h
0062 11 11     P2:   DW 1111h
0064           R:    DS 2
```

Všimněte si, že obě čísla (P1 a P2) jsou uloženy v zero page (na adresách 60h a 62h) a do zero page se ukládá i součet (R, 64h). Nejprve se nuluje příznak přenosu, pak se sečtou dva nižší bajty a poté dva vyšší. Případný přenos z nižšího do vyššího bajtu zařídí příznak C.

Odčítání je pak naprosto analogické, jen na začátku je potřeba příznak C nastavit na 1, nikoli nulovat.

Násobení

Vzpomínáte, jak jsme u procesoru 8080 násobili dvě osmibitová čísla? Můžeme podobný postup použít i u 6502? Tak v zásadě ano, ale se stejnými výhradami jako u sčítání: Nejsou registry.

Algoritmus rovněž využívá rotace do vyššího bajtu, jako u 8080, ale protože 6502 jaksi nemá nic jako „vyšší bajt“, jedná se zas o místo v paměti. A protože k paměti se nepřistupuje žádnou 16bitovou

operací, není nezbytně nutné, aby „vyšší bajt“ byl hned za „nižším bajtem“. A protože není 16bitové sčítání, viz výše, je nahrazeno rovněž algoritmem.

```
0200                    .ORG 200h
0200 A9 00              LDA #00h
0202 A8                 TAY
0203 84 62              STY *P1hi
0205 F0 0D              BEQ enterLoop
0207 18         DOADD:   CLC
0208 65 60              ADC *P1
020A AA                 TAX
020B 98                 TYA
020C 65 62              ADC *P1hi
020E A8                 TAY
020F 8A                 TXA
0210            LOOP:
0210 06 60              ASL *P1
0212 26 62              ROL *P1hi
0214            ENTERLOOP:
0214 46 61              LSR *P2
0216 B0 EF              BCS doAdd
0218 D0 F6              BNE loop
021A 4C 1A 02   INF:    JMP inf
0060                    .ORG 60h
0060 0D         P1:     DB 13
0061 09         P2:     DB 9
0062            P1HI:    DS 1
```

Výsledek násobení je v registrech X (vyšší bajt) a A (nižší bajt).

Všimněte si, že násobení dvojkou je zařízeno jako „šestnáctibitový shift“ – pomocí ASL a ROL.

9.14 Ahoj, světe, tady 6502

Nastala chvíle, kdy opět křemík ožije a provede, co po něm chceme. Začneme zase výpisem obligátního HELLO WORLD. K tomu ale budeme muset nějak ovládat ten komunikační obvod, který je v OMEN Bravo… Naštěstí to není složité, a tak si aspoň ukážeme, jak v takových chvílích postupovat.

Je dobré si nejprve nadefinovat základní konstanty. Obvod ACIA 6551 má, na rozdíl od 6850, o něco složitější ovládání, především proto, že máme hned čtyři registry. Příkazový, řídicí, stavový a datový.

```
ACIA_BASE   EQU     83FCh

SDR         EQU     ACIA_BASE
SSR         EQU     ACIA_BASE+1
SCMD        EQU     ACIA_BASE+2
SCTL        EQU     ACIA_BASE+3

SCTL_V      EQU     00011111b
SCMD_V      EQU     00001011b
TX_RDY      EQU     00010000b
RX_RDY      EQU     00001000b
```

V Bravu sídlí ACIA na adresách 8000h – 83ffh, a je dobrý zvyk používat co nejvyšší adresy, takové, v nichž jsou nedůležité bity rovné 1. Proto jsem zvolil adresy 83FCh – 83FFh.

Nejprve je třeba obvod inicializovat. Tentokrát musíme zapsat hned dvě řídicí hodnoty – do řídicího registru 1Fh, do příkazového 0Bh:

```
            LDA     #SCTL_V
            STA     SCTL
            LDA     #SCMD_V
            STA     SCMD
```

Samotná rutina SEROUT zas není tak moc odlišná, jen je třeba vzít na vědomí, že příznakové bity pro „znak připraven“ nebo „možno vysílat“ jsou na jiných pozicích:

```
SEROUT:     STA     SDR
WRS1:       LDA     SSR
            AND     #TX_RDY
            BEQ     WRS1
            RTS
```

Tentokrát nečekáme, až bude volno, ale volíme opačný přístup: nejprve vyšleme, a pak počkáme, až bude vysláno.

Pro zajímavost – takto vypadá čtení ze sériového portu:

```
SERIN:      LDA     SSR
            AND     #RX_RDY
            BEQ     SERIN
            LDA     SDR
            RTS
```

!etěvs ,johA

Všechno jde udělat i jinak, hlavně v osmibitovém assembleru. Takže ani pro Hello, world není jen jediný předepsaný postup.

U počítače Alpha jsem použil postup, kterému se říká ASCIIZ – tedy ASCII řetězec, ukončený bajtem s hodnotou 0 (ASCII + Zero). Tento způsob zápisu řetězců používají třeba překladače jazyka C – kdo z vás zná Céčko, tak ví.

Jazyk Pascal používal jiný přístup – první bajt udával délku řetězce ve znacích, a pak následovaly kýžené znaky. Někdy může být tento přístup výhodnější – hlavně tehdy, když nám délku spočítá překladač a natvrdo uloží do kódu.

Tak, základ zůstane stejný – definice konstant, inicializace i rutina SEROUT, jen ten vnitřek se změní. Registr X použijeme jako počítadlo a index znaku v řetězci. Když dosáhne hodnoty rovné délce řetězce, přestaneme. Ta zásadní pasáž kódu bude vypadat nějak takhle:

```
; Začínáme vypisovat znaky
TOUT:     LDX     #0 ; pozice
TLOOP:    LDA     TEXT,X
          JSR     SEROUT
          INX     ; X++

          CPX     #TSIZE ; už jsme na konci?
          BNE     TLOOP ; pokud ne, tak pokračujeme
ENDLOOP:  JMP     ENDLOOP ; věčná smyčka
TEXT:     DB      0Ch,"My hovercraft is full of eels!",0Dh,0Ah
TSIZE     EQU     $-TEXT
```

Steve Wozniak v Monitoru pro Apple I použil trik, kterým ušetřil několik bajtů (a tedy taktů procesoru). Ačkoli mi jeho trik připadá v tomto případě jako klasický příklad „overengineeringu“ (tedy optimalizace až přehnaná), tak si ji ukážeme, protože ilustruje schopnost podívat se na problém z naprosto neobvyklého úhlu. Což se zase při programování v assembleru často hodí.

Úvodní úvaha je jednoduchá: na začátku se nuluje registr X, v průběhu se pak zvyšuje o 1 a kontroluje se, jestli už má hodnotu N. Kdybychom na začátku do registru X uložili hodnotu N a šli v opačném pořadí, tedy směrem k nule, tak by odpadla instrukce porovnání a konec by nastal ve chvíli, kdy instrukce DEX (X=X-1) dojde k nule. Pak nastaví příznak Z (viz popis instrukce).

Nese to s sebou jeden drobný problém: čtení znaků by fungovalo od konce. Hm, a co? Tak je tam zapíšeme pozpátku!

```
; Začínáme vypisovat znaky
TOUT:     LDX     #tsize
; X je počet znaků, co zbývá vypsat
TLOOP:    LDA     text-1,x
; A protože X jde od hodnoty TSIZE k nule,
; tak se znaky berou od konce
          JSR     SEROUT
          DEX     ; X--
          BNE     tloop
; Dokud není 0, tak pokračujeme
ENDLOOP: JMP      endloop
TEXT:     DB      0ah,0dh,"slee fo lluf si tfarcrevoh yM",0ch
TSIZE     EQU     $-TEXT
```

Ušetřili jsme dva bajty a nějaký ten takt, přišli jsme o snadnou čitelnost. Ano, při programování osmibitů jsou situace, kdy dva bajty či pár taktů znamená hodně.

9.15 Ještě nějaký trik, prosím!

„*Ale no jistě,*“ odpovědělo sluchátko. Pojďte se podívat na následující kód:

```
0000                    .ORG 0
0000 A2 FF              LDX #0FFh
0002 9A                 TXS
0003 20 09 00           JSR label
0006 4C 00 00           JMP 0
0009 60         LABEL:   RTS
```

Přeložte si ho, spusťte emulátor a pojďme krokovat:

První instrukce (adresa 0000, 2 bajty) nastaví X na hodnotu FFh, druhá (adresa 0002, 1 bajt) tuto hodnotu zkopíruje do ukazatele zásobníku S. Třetí instrukce (adresa 0003, 3 bajty) je instrukce volání podprogramu. Volá se podprogram na adrese 0009…

V tuto chvíli se zastavíme a podíváme se na vrchol zásobníku, co se tam uložilo. Víme, že instrukce JSR ukládá dva bajty návratové adresy. Protože byl ukazatel nastaven na FFh, budou tyto dva bajty na adresách 01FEh a 01FFh. Co tam najdeme?

Na zásobníku je uložena hodnota 0005. Vidíme, že to není adresa instrukce za voláním JSR (ta je 0006), ale o 1 nižší. Instrukce RTS vezme hodnotu ze zásobníku, k ní přičte 1 a na tu adresu skočí.

K čemu nám bylo tohle mentální cvičení? Ukážeme si totiž jeden princip, který se u osmibitových procesorů používá docela často, a nejčastěji právě u výpisu různých textů. Pokud se text vyskytuje v programu jen jednou a je konstantní (tj. typicky nějaké hlášení), tak je pro programátora pohodlné napsat ho přímo do kódu, tam, kde potřebuje. Ušetří tím sekvenci „zadej někam adresu hlášení, co chceš vypsat – zavolej rutinu, která vypíše řetězec ze zadané adresy". Místo toho jen zavolá podprogram, jehož funkce se dá popsat slovy „vypiš znaky, co se nacházejí za instrukcí JSR, a až narazíš na ukončovací znak 00, tak se vrať za tu nulu, tam pokračuje program."

Nějak takhle (stále upravujeme kód z předchozího příkladu):

```
       JSR     PRIMM
       DB      0Ch,"My hovercraft is full of eels!",0Dh,0Ah,00h
DONE:  JMP     DONE ; TO JE KONEC!!! :(
```

Vidíte, že je tam instrukce volání podprogramu PRIMM (PRint IMMediately), za ní jsou přímo znaky požadované hlášky, ukončené nulou, a za tím zase pokračuje program.

Co musí udělat podprogram PRIMM? Představte si, že je vyvolán. V tu chvíli je na zásobníku „adresa instrukce za JSR – 1". Ukazatel SP je jeden bajt POD touto hodnotou, návratová adresa je tedy na adresách SP+1 a SP+2.

Nejdřív si uložíme pracovní registry A, X a Y – tím se SP sníží o 3 a situace na zásobníku bude vypadat takto:

SP+5	Vyšší bajt návratové adresy
SP+4	Nižší bajt návratové adresy
SP+3	Obsah registru A
SP+2	Obsah registru X
SP+1	Obsah registru Y
SP	První volná pozice na zásobníku

Takže na adrese SP + 0100h + 4 je nižší bajt návratové adresy, na adrese SP + 0100h + 5 je vyšší. Tuto hodnotu si můžeme někam zkopírovat – ideálně do zero page do dvou buněk vedle sebe. Příhodně pak využijeme adresní mód IZY. Připomeňme si*: tento mód vezme adresu ze dvou vedle sebe ležících paměťových míst, k té adrese přičte obsah registru Y a výsledek udává adresu, kam se má sahat pro data.*

Jakmile narazíme na konec řetězce (anebo nám přeteče registr Y), tak končíme s vypisováním. Teď je potřeba vzít tu původní adresu, k ní přičíst počet vypsaných znaků (tedy registr Y), tu pak zase zapsat na zásobník – a pak už jen standardně vrátit obsah registrů a provést RTS.

A protože vlastní studium zdrojového kódu řekne víc než sáhodlouhé popisy, tak bez dalšího vysvětlování – podprogram PRIMM:

```
PRIMM:
          PHA     ; Uložím A
          TXA
          PHA     ; Uložím X
          TYA
          PHA     ; Uložím Y
          TSX     ; Ukazatel na zásobník si načtu do X
          LDA     0104h,X ; Nižší byte návratové adresy

; (0100h je základní adresa zásobníku, X je tu aktuální
; ukazatel zásobníku, +4 proto, že ukazatel SP ukazuje na
; první volné místo, SP+1 je uložený registr X,
; SP+2 je uložený registr Y, SP+3 je uložený registr A
; (na začátku podprogramu jsme si je ukládali)
; SP+4 a SP+5 jsou nižší a vyšší bajt návratové adresy,
; tedy poslední bajt instrukce JSR

          STA     00h ; Uložíme do ZP (třeba na adresu 00)
          LDA     0105h,X
; Analogicky vyšší byte návratové adresy…
; … ukládáme do ZP na adresu 01

          STA     01h
          LDY     #01h
; Nastavíme Y na počáteční hodnotu. Měla by to být
; nula, ale protože víme, že návratová adresa je ve
; skutečnosti o 1 nižší, než adresa prvního bajtu za JSR,
; tak začneme od jedničky.
PRIM2:
          LDA     (00h),Y
; Načteme bajt. Adresa je „obsah buněk 00 a 01“ + Y
          BEQ     PRIM3 ; Načetli jsme nulu? Tak končíme!

          JSR     SEROUT ; Nenulový znak ale vypíšeme
          INY     ; posuneme se na další adresu
; a pokud jsme ještě nepřetočili počítadlo, tak
; pokračujeme v tisknutí znaků.
; Když už je Y nulové, tak je načase skončit.
          BNE     PRIM2
```

```
PRIM3:
          TYA     ; V Y je „počet znaků + 1“ – přesuneme do A
          CLC     ; budeme sčítat, je potřeba vynulovat C
; K A si přičteme nižší bajt původní návratové adresy
          ADC     00h
          STA     0104h,X ; a „podvrhneme“ ji do zásobníku
          LDA     #00h ; Vynulujeme A
; a přičteme hodnotu vyššího byte návratové adresy.
          ADC     01h
; Pokud při předchozím sčítání došlo k přenosu, tak se

; vyšší bajt zvedne o 1, jinak zůstane stejný
; A opět vyšší bajt návratové hodnoty analogicky uložíme
; na zásobník a budeme se tvářit, že to tak už bylo
          STA     0105h,X
          PLA     ; Přečteme uloženou hodnotu
          TAY     ; co patří do registru Y
          PLA     ; a úplně stejně tu, co
          TAX     ; patří do registru X
          PLA     ; ještě původní hodnotu A
          RTS     ; a návrat!
```

Můžete si zkusit složit celý zdrojový kód a vyzkoušet, jak hezky funguje.

(Rutina PRIMM pochází z operačního systému Commodore C64 a je mírně upravena.)

Pro pozorné: v kódu jsou dvě chyby, které se projeví při určité konstelaci – zkuste na ně přijít.

Mimochodem – existuje ještě jeden používaný způsob označování konce řetězců, hlavně u anglických textů. Kromě zadaného počtu znaků + řetězce nebo řetězce ukončeného bajtem 00h (u CP/M služba pro vypisování řetězců používá ukončování znakem $) můžeme použít i trik, který počítá s tím, že anglická abeceda si v ASCII vystačí se znaky z rozsahu 00h-7Fh. Pak stačí poslednímu znaku nastavit nejvyšší bit na 1 (tj. posunout jej do rozsahu 80h-FFh). Ze znaku „!“ (kód 21h) se tak stane znak s kódem A1h. No a postup je prostý – před vypsáním znaku použijeme AND s hodnotou 7Fh (abychom nastavili nejvyšší bit na 0), znak vypíšeme a pak zkontrolujeme, jestli není nejvyšší bit roven 1 (u 6502 třeba tak, že ho načteme do registru A – tím se nejvyšší bit zkopíruje do příznaku N). Pokud ano, byl to poslední znak a my se můžeme vrátit. Napsání rutiny, která bude takto pracovat, nechám už na vás, máte to za domácí úkol…

Řešení úlohy

První případ chyby je, že je počet znaků větší než 255. Registr Y u posledního znaku „přeteče“ do nuly, rutina tím končí, ale bohužel se tím návratová adresa ocitne někde uprostřed textu a výsledek bude katastrofální. Řešení existuje – nechat přetočit Y, ale přitom si uloženou adresu zvýšit o 0100h.

Druhý problém nastane v případě, že ukazatel zásobníku SP je nízko. V takovém případě přeteče přes nulu, dostane se opět do vysokých hodnot FDh-FFh, a v takovém případě jednoduchý přepočet pomocí vzorce „SP + 104h“ selže, protože se ocitneme mimo zásobník, na adresách 0200h a vyšších, a nejen že načteme nesmysly, ale taky nesmysly uložíme. I tuto situaci by bylo možné ošetřit, ale v tomto případě to není asi úplně potřeba, pokud inicializujeme zásobník standardně, tj. na hodnotu FFh. Pokud se totiž za takové situace stane, že se zásobník protočí přes nulu, tak máme zásadnější problém někde jinde…

9.16 Organizace kódu

Protože vy, čtenáři, snad všichni znáte nějaké vyšší jazyky, tak nemusím moc složitě představovat koncepty modulů a lokálních proměnných. Ano, i tyhle věci v assembleru máme, ale není to tak úplně prosté…

Moduly

No, říkejme tomu tak. Ve skutečnosti se jedná jen o jednoduchý INCLUDE „jméno“, který na to místo načte obsah externího souboru.

Některé staré assemblery vůbec žádný include neměly. Ono by to třeba u ZX Spectra 48 s páskou nebylo moc pohodlné. Čímž neříkám, že takové kompilery nebyly, třeba HiSoft C měl #include, jak se na céčko sluší a patří, a při překladu jste spustili magnetofon, kde byly soubory ke slinkování, překladač si je prošel, načetl, přeložil… (Ano, bylo to tak děsivé, jak to zní.)

Většina modernějších assemblerů include samozřejmě má, jen se liší jeho syntax. Některé assemblery používají .INC, některé INCLUDE, některé .INCLUDE, takže nezbývá než si přečíst manuál k tomu kterému kousku. Já ve svém překladači používám tvar *.INCLUDE název souboru*.

Řekněme, že mi připadá jako dobrý nápad (a on to dobrý nápad je) přesunout rutiny pro výpis znaku a načtení znaku někam stranou, do nějaké společné (common) knihovny (library), kterou si důvtipně nazvu „comlib.a65“. V hlavním programu tak budu moci vesele tyhle rutiny používat, aniž by mi překážely ve zdrojáku, stačí jen, když je vhodně includuju.

Tím se nám zdroják rozštípnul na dva soubory: comlib.a65 (knihovna) a vlastní zdrojový kód.

```
; COMLIB.A65 – základní komunikační knihovny

; Nastavení adres pro komunikační obvod ACIA 6551
ACIA_BASE   EQU     83FCh

SDR         EQU     ACIA_BASE
SSR         EQU     ACIA_BASE+1
SCMD        EQU     ACIA_BASE+2
SCTL        EQU     ACIA_BASE+3

SCTL_V      EQU     00011111b
SCMD_V      EQU     00001011b
TX_RDY      EQU     00010000b
RX_RDY      EQU     00001000b

ACIAINIT:   LDA     #SCTL_V
            STA     SCTL
            LDA     #SCMD_V
            STA     SCMD
            RTS

SEROUT:     STA     SDR
WRS1:       LDA     SSR
            AND     #TX_RDY
            BEQ     WRS1
            RTS
SERIN:      LDA     SSR
            AND     #RX_RDY
            BEQ     SERIN
            LDA     SDR
            RTS
```

Tenhle soubor pak elegantně načteme v hlavním programu:

```
; program začíná na adrese E000h, tedy tam, kde začíná ROM
          .ORG    0E000h
; Emulátor má začít odsud
          .ENT    $
; K testu použij emulátor počítače Bravo
; (pouze pro IDE ASM80.com)
```

```
          .ENGINE bravo
; Vstupní adresa
RESET:
; Nastavíme si ukazatel zásobníku
          LDX     #$ff
          TXS

          JSR ACIAINIT

LOOP:     JSR SERIN
          JSR SEROUT

          JMP     LOOP ; Stále dokola…

; ještě někam musíme vložit tu knihovnu…

; třeba sem, sem se hlavní program nedostane
.include comlib.a65

          .ORG    0FFFCh
          DW      reset
          DW      reset
```

Do comlib.a65 si klidně můžeme přihodit i rutinu PRIMM z minulé kapitoly. K tomu ale až na konci. Teď si musíme ukázat ještě jednu důležitou vlastnost assemblerů…

Assembler totiž sám o sobě nemá lokální jména. Jakmile jednou nadefinujete konstantu, návěští, něco, tak to je vidět v celém kódu. Což je docela problém, protože u složitějšího programu vám brzy dojde fantazie při pojmenovávání např. smyček. „LOOP1“, „LOOP2“, … to není moc elegantní.

Nemluvě o tom, že třeba použijete návěští „LOOP“ v nějaké knihovní funkci. V hlavním programu na to zapomenete (nebo o tom ani nevíte, protože tu knihovnu dělal někdo jiný), a překladač – logicky – zařve, že návěští bylo už použité. Co s tím?

Lokální návěstí

V téhle situaci se hodí lokální návěstí. Špatná zpráva je, že ne každý assembler je podporuje, a pokud ano, tak má svou konvenci, které bude pravděpodobně odlišná od všech ostatních konvencí všech ostatních assemblerů.

Některé mocnější assemblery zavádějí konstrukce „procedure“ a deklaraci „local“ apod., jiné se staví k problému z druhé strany a dovolují pro drobné smyčky a návěští používat jakási „pseudonávěstí“

a odkazovat se na ně zápisem „skoč na předchozí pseudonávěstí“, „skoč na následující pseudonávěstí“, „skoč o dvě pseudonávěstí zpátky“…

Já jsem v ASM80 zvolil cestu bloků. Blok začíná direktivou „.block“ a končí direktivou „.endblock“. Všechna návěstí, co jsou v něm definována, jsou lokální. To znamená že v bloku na ně můžete odkazovat, mimo něj nejsou vidět. Pokud chcete, aby bylo návěští vidět i mimo blok, dejte před jeho název znak @ – ten se nestane součástí jména, jen říká, že toto návěští bude globální. „@SEROUT:“ říká „Definuj globální návěští se jménem SEROUT“.

Díky tomu můžu jako první řádek knihovny comlib napsat .block, na poslední .endblock, a vím, že pokud takovou knihovnu includuju, tak mi z ní nic „nevyteče“ ven, pokud explicitně neřeknu, co má být vidět zvenčí. Takže nová, šetrná verze comlib vypadá takto:

```
; COMLIB.A65 – základní komunikační knihovny

.block
; díky deklaraci BLOCK nebudou následující návěští vidět
; ve zbytku kódu, kromě těch, před kterými je @

ACIA_BASE   EQU     83FCh

SDR         EQU     ACIA_BASE
SSR         EQU     ACIA_BASE+1
SCMD        EQU     ACIA_BASE+2
SCTL        EQU     ACIA_BASE+3

SCTL_V      EQU     00011111b
SCMD_V      EQU     00001011b
TX_RDY      EQU     00010000b
RX_RDY      EQU     00001000b

@ACIAINIT:   LDA     #SCTL_V
            STA     SCTL
            LDA     #SCMD_V
            STA     SCMD
            RTS

@SEROUT:     STA     SDR
WRS1:       LDA     SSR
            AND     #TX_RDY
            BEQ     WRS1
            RTS
@SERIN:      LDA     SSR
```

```
            AND     #RX_RDY
            BEQ     SERIN
            LDA     SDR
            RTS
.endblock
```

Díky uzavření, „zapouzdření" zdrojáku můžu v hlavním programu použít klidně návěští WRS1, a nedojde k chybě, protože to, které jsem použil v comlib.a65, bude vidět pouze v comlib.a65, nikde jinde. Stejně tak názvy jako ACIABase, SSR apod. Jediné, co bude vidět zvenčí, je SERIN, SEROUT a ACIAINIT.

https://8bt.cz/65src

9.17 65C02 – vylepšená verze s technologií CMOS

I dnes se procesor 6502 stále vyrábí, a vyrábí jej dokonce několik výrobců. Nejvýznamnější z nich je Western Design Company (WDC) – firma, založená jedním ze spoluautorů 6502, Billem Menschem. Kromě WDC vyrábí 65C02 i další výrobci – Rockwell, Synertek, GTE – ale jejich verze se mírně liší. Doporučuju proto dát pozor při hledání datasheetů a nepředpokládat, že W65C02 od WDC bude mít stejné vývody a instrukce jako 65C02 od Rockwellu.

Přidané instrukce

Instrukce	Operační kód (hexadecimálně)	Popis
PHX	DA	Push X (není potřeba přes registr A)
PLX	FA	Pull X
PHY	5A	Push Y
PLY	7A	Pull Y
STZ addr	9C	STore Zero – uloží nulu na zadané místo
STZ addr, X	9E	
STZ zp	64	
STZ zp,X	74	
TRB addr	1C	Test and Reset Bit
TRB zp	14	

Instrukce	Operační kód (hexadecimálně)	Popis
TSB addr	0C	Test and Set Bit
TSB zp	04	
BRA rel	80	Nepodmíněný relativní skok (-128 až +127)
BBR0 zp,rel – BBR7 zp,rel (jen WDC/Rockwell)	0F, 1F, 2F, 3F, 4F, 5F, 6F, 7F	Skok pokud je vybraný bit =0
BBS0 zp, rel – BBS7 zp,rel (jen WDC/Rockwell)	8F, 9F, AF, BF, CF, DF, EF, FF	Skok pokud je vybraný bit 1
RMB0 zp – RMB7 zp (jen WDC / Rockwell)	07, 17, 27, 37, 47, 57, 67, 77	Nuluje vybraný bit v bajtu na dané adrese
SMB0 zp – SMB7 zp (jen WDC / Rockwell)	87, 97, A7, B7, C7, D7, E7, F7	Nastaví vybraný bit v bajtu na dané adrese
STP (jen WDC)	DB	SToP – zastaví procesor a přepne ho do režimu sníženého odběru až do signálu RESET
WAI (jen WDC)	CB	WAIt – zastaví procesor jako STP, ale k probuzení kromě signálu RESET lze použít i libovolné přerušení

Instrukce TRB nejprve udělá AND mezi obsahem registru A a zadaného paměťového místa. Podle výsledku nastaví příznak Z. Pak změní obsah paměti tak, že vynuluje bity, které jsou v registru A nastavené. Obsah registru A se nezmění. Instrukce TSB funguje podobně, jen v posledním kroku změní obsah paměti tak, že bity, které jsou v registru A nastavené, nastaví na 1 (udělá operaci OR).

Instrukce RMB a SMB slouží k nastavování bitů v paměti (v zero page). Součástí operačního kódu je číslo bitu k nastavení / nulování.

Instrukce BBR a BBS naopak vybraný bit v zero page otestují, a podle jeho stavu udělají relativní skok.

Rozšíření adresace

Některé kombinace instrukcí a adresních módů v klasickém 6502 nejsou validní. 65C02 přidává následující možnosti:

Instrukce	Operační kód (hexadecimálně)	Význam
ADC (zp)	72	ADC s nepřímou adresou – podobné (zp,X) pro X=0
AND (zp)	32	AND s nepřímou adresou
CMP (zp)	D2	
EOR (zp)	52	
LDA (zp)	B2	
ORA (zp)	12	
SBC (zp)	F2	
STA (zp)	92	
JMP (addr,X)	7C	Nepřímý skok s indexem – k adrese se přičte obsah registru X a výsledek se použije pro získání dvou bajtů cílové adresy
DEA	3A	Též DEC A – snížení A o 1
INA	1A	Též INC A – zvýšení A o 1
BIT addr,X	3C	Instrukce BIT s absolutní indexovanou adresou
BIT zp,X	34	BIT s indexovanou adresou v ZP
BIT #	89	BIT s konstantou

https://8bt.cz/65c02

10 Intermezzo paměťové

10 Intermezzo paměťové

Když už jsem to v předchozí kapitole nakousl, pojďme to dožvýkat. Tématem tohoto intermezza budiž „jak zařídit, aby se po zapnutí počítače něco dělo, a pak aby se dělo něco jiného“?

Jeden příklad jsem už nadhodil při probírání operační frekvence procesoru 6502 – nějak zařídit, aby počítač po startu běžel na pomalejší frekvenci, zkopíroval si obsah z (pomalé) EEPROM do (rychlé) SRAM a pak sám sebe přepnul do TURBO módu.

Podobný problém řešili konstruktéři počítačů se systémem CP/M. Tento systém pro počítače s procesorem 8080 / Z80 měl jako jednu ze základních vlastností to, že se programy zaváděly do paměti od adresy 0100h. To znamená, že v těch místech musela být paměť RAM, ideálně co největší souvislý blok. Asi můžu dát od adresy 0 až do, cojávím, FBFFh samou RAM a ROM od adresy FC00h do FFFFh. Jenže uvažte následující:

- po startu jede procesor od adresy 0000h. Pokud tam je (neinicializovaná) RAM, plná náhodného obsahu, je pravděpodobnost, že to celé zhavaruje dřív, než to vůbec do ROM dospěje, téměř rovna jedné.
- když dáme ROM i do prostoru 0000 – 0100h, tak by se to dalo nějak vyřešit, ale je potřeba po skoku do ROM tuto paměť zase odpojit, protože systém potřebuje zrovna tady mít taky RAM.

Co s tím?

Podobný problém řešil třeba i autor PMD-85. Tento počítač měl RAM od adresy 0000 do 7FFFh, od 8000h do C000h byla EPROM, od C000h do FFFFh zase další RAM. A co při startu?

Řešení bylo prosté: použil se klopný obvod R-S. Signál RESET ho nastavil do jednoho stavu, nazvěme ho „stav 0“. V tomto stavu se vůbec negeneroval signál /RAMCS (toto berte s rezervou, v PMD se ty signály jmenovaly a generovaly úplně jinak) a místo toho byla vybrána paměť EEPROM. Procesor začal číst od adresy 0000h, kde byl v tu chvíli stejný obsah jako od adresy 8000h (totéž by se stalo třeba u Alphy, kdybychom nějakým dalším obvodem aktivovali signál /ROMCS).

První instrukce pak byla JMP 8003h – skok na následující instrukci. Vypadá to zbytečně, ale ve skutečnosti se tím do programového čítače dostala správná adresa EPROM. Následující instrukce inicializovaly obvod 8255 pro komunikaci s klávesnicí. Signál pro zápis do periferie překlopil klopný obvod do stavu 1, dekodér se tím přepnul do normálního režimu a paměti se začaly objevovat v paměťovém prostoru tak, jak mají.

U jedné konstrukce se systémem CP/M jsem viděl obdobný fígl. Opět klopný obvod, nulovaný signálem RESET, ovládal dekodér a připojoval ROM od adresy 0. Zde byl krátký program – bootstrap, který nahrál do paměti RAM jeden konkrétní sektor z diskety, ve kterém byl vlastní

loader pro zbytek systému. Program se nahrál kamsi do RAM, spustil, a opět jeho prvním úkolem bylo zase odpojit ROM.

U konstrukcí a la PMD by šlo využít ještě jednoho fíglu, totiž stavu signálu A15. Jakmile by se skočilo na adresu nad 8000h, logická 1 na vývodu A15 by přepnula klopný obvod.

Podobná konstrukce samozřejmě zesložití dekodér adres. Tam, kde jsme si vystačili s jedním obvodem 7400 a čtyřmi hradly NAND, budeme potřebovat mnohem složitější zapojení. Budeme potřebovat standardní dekodér, budeme potřebovat logiku, která jej podle stavu klopného obvodu „přehlasuje", budeme potřebovat samotný klopný obvod a pak něco, co ho nastaví a nuluje.

Mimochodem, s tou pracovní frekvencí je to docela jednoduché. Představte si opět stejný klopný obvod: RESET jej překlopí do 0, nějaký signál od CPU do 1. Generátor hodin není interní, ale externí, třeba s obvodem 7400 a krystalem, co kmitá třeba na frekvenci 14 MHz. Za tímto oscilátorem jsou dva klopné obvody D, nebo binární čítač, zkrátka nějaký dělič kmitočtu. Bude stačit kmitočet vydělit čtyřmi (=3,5 MHz) nebo osmi (1,75 MHz). Klopný obvod pak jednoduchým multiplexorem vybere buď nízký kmitočet (=0), nebo vysoký (=1). Po RESETu tak procesor poběží hezky pomaloučku, bude si moci zkopírovat vše potřebné do RAM, a pak (třeba) pomocí přístupu ke konkrétní adrese přepne klopný obvod… Zbytek bude fungovat podle výše uvedené *příručky pro lov jelena*. Tedy pardon, podle výše uvedeného popisu.

10.1 Moc paměti?

Nebývá to pravidlem. Většinou si stěžujeme, že máme paměti málo. Ale stane se, že konstruktér má paměti prostě moc. Nám se to stalo u Brava. Máte k dispozici prostor 16 kB a paměť s kapacitou 32 kB. Co teď, přátelé, co teď?

Jedna možnost je přebývající část prostě ignorovat, příslušný adresový bit natvrdo zapojit na 0 (nebo na 1) a používat jen půl paměti.

Druhá možnost je přidat propojku (jumper) nebo přepínač a při startu si zvolit, jestli chcete použít vyšší polovinu, nebo nižší polovinu paměti. Můžete tak mít dvě různé verze programového vybavení.

Třetí možnost je zase využít nějakého klopného obvodu či registru, který můžete programově ovládat. Můžete si přepínat programy „za běhu".

Pokud byl součástí konstrukce obvod pro paralelní rozhraní typu 8255 nebo 6821/6822, který sloužil třeba k obsluze klávesnice, bylo možné jej použít pro výběr konkrétní stránky paměti (kdo si vzpomínáte na Didaktik Gama a jeho přepínání „horní části paměti", tak vězte, že to bylo řešené přesně tímto způsobem).

Tady je na místě velmi výrazné doporučení, tak raději tučně: **Není dobré přepínat paměť v oblasti, v níž se právě provádí program!**

Velmi pravděpodobně se totiž stane, že hned po přepnutí se ocitnete někde, kde jste být nechtěli, něco si přepíšete, a nakonec všechno zhavaruje. Ne že by to nešlo vůbec – třeba u ZX Spectra se přepínaly externí ROM ve chvíli, kdy procesor prováděl program v interní ROM, ale platilo velmi přísné pravidlo: přepnout se může pouze na konkrétních adresách, na kterých bylo zaručeno, že ve stínové ROM bude následovat smysluplný obsah.

A co když chceme víc RAM? Na trhu jsou k dispozici čipy SRAM s kapacitou 128 či 512 kB (AS6C1008 / AS6C4008, popř. 628128 / 628512). Vypadají stejně jako ty, které používáme (32 kB), jen mají víc nožiček (128 kB mají i adresní vstupy A14 a A15, půlmegové přidávají i A16 a A17). Je lákavý nápad takovou paměť připojit a nějak naplno využít…

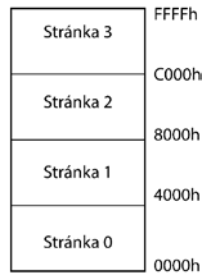
10.2 Stránkování

Osmibitové procesory, se kterými pracujeme, mají zkrátka šestnáct bitů adresové sběrnice a s tím nehneme. Navíc nemají žádnou jednotku správy paměti. Některé pokročilejší (Zilog Z180, WDC 65816) obojím oplývají, ale ve světě starých osmibitů si musíme poradit jinak.

Samozřejmě že odpověď „použij jiný procesor“ je zcela legitimní a na místě. Jenže smyslem a cílem téhle knihy není „použít jiný procesor“, ale ukázat, jak se takový problém v praxi řešil a řeší.

Pokud k osmibitovému procesoru chcete připojit víc paměti, než je 64 kB, je jasné, že potřebujete nějakou podporu v okolních obvodech, něco, co zařídí, aby se paměť připojovala tak, jak má.

Problém se řeší stránkováním a mapováním. Představme si adresní prostor nějakého osmibitu, tedy adresy v rozsahu 0000 až FFFFh, rozdělený na čtyři stránky po 16 kB. Nějak takto:



Obrázek 42: Stránkování paměti u ZX Spectra

V OMEN Alpha máme v horních dvou stránkách RAM, v dolních EEPROM. U Brava je RAM v dolních dvou stránkách, v horních je EEPROM a periferie.

Pro zajímavost: U ZX Spectra 48 byla ve stránce 0 systémová ROM, ve stránce 1 byla video RAM (o níž se procesor dělil s obvodem ULA, proto programy uložené v této paměti běžely pomaleji) a ve stránkách 2 a 3 bylo 32 kB RAM. Fyzicky tvořila stránku 1 osmice dynamických pamětí 4116 a stránky 2 a 3 dalších osm obvodů (vlastně „kazové" obvody 64 kB x 1 bit, u kterých se používala jen polovina čipu).

U ZX Spectra 128 zůstalo rozložení stejné. Jenže najednou přebývalo 80 kB RAM, k nimž by se programátor za normálních okolností nedostal. Proto vývojáři celou paměť RAM rozdělili na osm „paměťových bank" (označených Bank 0 až Bank 7) a udělali následující:

- Ve stránce 0 zůstala ROM (popřípadě nová ROM 128)
- Ve stránce 1 byla připojena napevno banka 5
- Ve stránce 2 byla připojena banka 2
- Ve stránce 3 (C000h – FFFFh) byla připojena libovolná banka dle programátorovy libosti.

K přepínání sloužilo několik bitů paralelního portu (který byl součástí použitého zvukového čipu). Bity 0, 1, 2 vybíraly, která banka bude připojena ve stránce 3, bit 3 vybíral, která banka bude sloužit jako zdroj videosignálu (za normálních okolností to byla banka 5, ale mohli jste přepnout i na banku 7).

Takové řešení je poměrně elegantní a jednoduché na používání. Programátor má k dispozici 32 kB RAM přímo, ta se nemění. Nejvyšších 16 kB si ale může přepínat podle libosti. Díky tomu je možné psát programy tak, aby bylo přepínání bezpečné, tzn. aby procesor pracoval v oblasti, která se nepřepíná, a aby tamtéž byl uložen zásobník.

Didaktik Gama nepoužíval stránkování po 16 kB, místo toho stránkoval po 32 kB a přepínal obsah ve stránkách 2 a 3 najednou.

Co nám brání, abychom totéž neudělali taky, třeba u Alphy? No, brání nám v tom fakt, že bychom si takto přepnuli celou RAM, takže přepínat by mohl pouze kód v EEPROM, zároveň bychom po přepnutí přišli o veškerá uložená data, včetně návratových adres na zásobníku, takže by se po provedení přepínací rutiny neměl program kam vrátit, no a při případném kopírování dat mezi stránkami bychom byli odkázáni jen na vnitřní registry.

U Brava je odpověď ještě rezolutnější: při práci s procesorem 6502 prostě není dobré za běhu přemapovat paměť v oblasti 0000h – 0200h. Šmitec.

Řešení se nabízí dvojí. Buď říct, že třeba stránka 2 zůstane namapovaná napevno a přepínat se bude jen stránka 3, nebo zastínit EEPROM a dát programátorovi do rukou možnost připojit libovolnou banku RAM třeba do stránky 1.

Uvažme teď jednotlivé případy. Jako model si zvolme RAM o kapacitě 128 kB.

Prosté přepínání

Sice jsme tuto možnost zavrhli, ale přesto si řekněme, jak bychom k ní přistupovali.

Čip 128 kB bychom zapojili stejně jako 32 kB, tedy bez změny signálů /RAMCS a A0 až A14. Adresní vstupy A15 a A16 bychom museli *nějak* vyřešit. Dejme tomu, že bychom pro ně vyhradili dva bity nějakého paralelního portu, třeba PA0 a PA1 u portu A. Zápisem do dvou nejnižších bitů tohoto portu by se vybrala 32kB banka… Ale takto to dělat nebudeme.

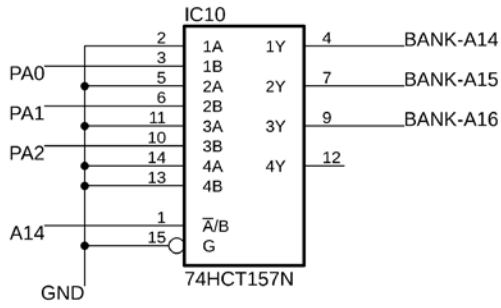
Mapování ve stránce 3

Při mapování ve stránce 3 to bude podstatně složitější. Budeme muset nějak zajistit „překlad" podle následujících pravidel:

A15	A14	Význam	BA16	BA15	BA14
0	X	Přístup do EEPROM	X	X	X
1	0	Napevno banka 0	0	0	0
1	1	Banka vybraná portem A	PA2	PA1	PA0

Paměť 128 kB jsme rozdělili na 8 bank po 16 kB. Výběr banky vyřešíme opět pomocí nejnižších bitů portu PA u obvodu 8255. A ve stránce 2 zůstane napevno banka 0.

Generování signálů /RAMCS a /ROMCS zůstává stejné. Signál A14 bude přiveden na multiplexor, který vybere buď vstupy se samými 0, nebo vstupy z portu PA. Poslouží například obvod 74HCT157:



Obrázek 43: zapojení multiplexoru pro rozšíření paměti

Vybavovací vstup G je připojen nastálo k zemi, ale může být stejně dobře připojen k signálu /RAMCS. Signál A14 slouží k přepínání mezi vstupy A a B. Všechny vstupy A jsou připojeny k zemi, takže pokud je A14=0 a jsou připojeny vstupy A, je na výstupech 0. Výstupy jsem označil, aby se to nepletlo, prefixem BANK a jsou připojeny rovnou na příslušné adresní vstupy paměti 128 kB.

Úprava je tedy hodně jednoduchá a teoreticky by nám vystačila i pro čipy 256 kB. Takové ale k dispozici nebývají, častěji máme 128 kB, nebo 512 kB. Pro plný čip 512 kB bychom museli použít pětikanálový multiplexor (v TTL řadě 74xx nejsou, takže dva čtyřbitové…) nebo nějaké jiné řešení, třeba z pěti hradel AND…

Stínování ROM

Podobné řešení používalo třeba Atari 130 XE – část jeho paměti RAM byla za normálních okolností překryta pamětí ROM, ale programátor mohl ROM „odstínit" a přistupovat přímo k RAM.

U Alphy či Brava můžeme použít něco podobného. EEPROM můžeme odstínit buď v plném rozsahu, nebo třeba jen z části. Pokud odstíníme EEPROM v plném rozsahu, bude moct paměť přepínat pouze uživatelský program, běžící v RAM. Zároveň budeme muset vyřešit, aby po resetu byla připojena paměť ROM. Pokud budeme přepínat jen část, třeba stránku 1, odpadne starost s resetem, přepínání bude moci řešit i obslužný program v ROM, ale zesložití se generování /RAMCS, /ROMCS a dalších.

Pro příklad: port PA opět vyhradíme pro výběr stránky (PA0 – PA2) a jeden bit (PA3) bude indikovat, jestli je připojena ROM, nebo RAM. Pravidlo bude následující:

A15	A14	/ROMCS	/RAMCS	BA16	BA15	BA14
0	**0**	**0**	**1**	X	X	X
0	**1**	**PA3**	**/PA3**	PA2	PA1	PA0
1	**0**	**1**	**0**	0	0	0
1	**1**	**1**	**0**	0	0	1

Když si rozepíšeme jednotlivé kombinace, vypadá to takto:

A15	A14	PA3	/ROMCS	/RAMCS
0	**0**	**0**	0	1
0	**0**	**1**	0	1
0	**1**	**0**	0	1
0	**1**	**1**	1	0
1	**x**	**x**	1	0

Vychází mi z toho, že /RAMCS je A15 NAND (A14 OR PA3) a /ROMCS negace – tedy A15 OR (A14 AND PA3). Multiplexor necháme přepínat signálem A15, a to mezi variantami PA-2-PA1-PA0 (A15=0) a 0 – 0 – A14 (A15=1).

K tomu samozřejmě ještě připočítejte signál IO/M, takže plná verze je:

- /RAMCS = IO/M OR (A15 NAND (A14 OR PA3))
- /ROMCS = IO/M OR (A15 OR (A14 AND PA3))

Ať počítám, jak počítám, s jedním hradlem NAND nevystačíme… Buď bude potřeba zkombinovat hradla NAND a OR, nebo využít třeba programovatelný obvod GAL, v němž takovéto složitější kombinační obvody lze poměrně snadno syntetizovat.

Ve skutečnosti se naštěstí dají oba složité výrazy poměrně snadno přepsat do podoby NANDů a negací:

- /RAMCS = (NOT IO/M) NAND (NOT A15 NAND (A14 NAND PA3))
- /ROMCS = /RAMCS NAND (NOT IO/M)

Kolik hradel tedy budeme potřebovat?

- H1: IO/M -> NOT IO/M
- H2: A15 -> NOT A15
- H3: A14 NAND PA3
- H4: H2 NAND H3
- H5: H1 NAND H4 (/RAMCS)
- H6: H5 NAND H1 (/ROMCS)

Šest hradel tedy bude stačit – použijeme dva obvody 7400 a ještě dvě hradla zbývají pro jiné použití.

Fyzicky oddělené paměti

Můžeme trochu vylepšit předchozí řešení, a to tak, že v bankách 2 a 3 necháme původní RAM 32 kB a celou paměť 128 kB připojíme jako druhý čip a budeme ji mapovat zase do stránky 1. Výsledná kapacita paměti bude tedy 160 kB. Co tímto řešením získáme a co obětujeme?

1. Na desce bude o jeden čip víc. Dobře, to může být problém.
2. /RAMCS zůstane jako byl dřív
3. Odpadne multiplexor – rozšířená paměť bude mít zapojené adresní vstupy A0 až A13, A14 až A16 budou přímo připojené na PA0, PA1, PA2.
4. Vznikne nový signál /ERAMCS (jako že EXTENDED), který bude přímo ovládat /CE u paměti 128 kB.
5. /ROMCS bude potřeba změnit.

Pokud ale oželíme 16 kB ROM, tak můžeme stránku 1 nechat pouze pro rozšiřující paměť. Potřebné výrazy pak budou:

- /RAMCS = (NOT IO/M) NAND A15
- /ROMCS = A15 OR A14 OR IO/M
- /ERAMCS = A15 OR (NOT A14) OR IO/M

S trochou úprav získáme:

- /ROMCS = (NOT A15) NAND (NOT A14) NAND (NOT IO/M)
- /ERAMCS = (NOT A15) NAND A14 NAND (NOT IO/M)

Stačí nám tedy dvě třívstupová hradla NAND (74HC10) a čtyři dvouvstupové NANDy (negace A15, negace A14, negace IO/M a vytvoření /RAMCS).

Výhoda takového řešení bez multiplexoru je, že stačí použít o dva bity portu A víc – a můžeme použít přímo paměť 512 kB.

10.3 Všechno najednou, a ještě něco navrch…

Samozřejmě lze výše uvedené postupy kombinovat – dovedu si představit situaci, kdy počítač obsahuje třeba 128 kB RAM, která je nějak mapovaná po stránkách, zároveň v systému je i paměť ROM, která je po RESETu připojená a následně se odpojí…

Mimochodem, pokud nebudete chtít použít 8255 nebo podobný velký obvod, můžete použít obvod 74HC273 – je podobný obvodu 573, který jsme použili u 8085 jako adresový latch, ale má nulovací vstup, který lze připojit k signálu /RESET a zajistit, že po startu bude v tomto registru vždy 0.

Když už tedy máme v systému víc paměti, můžeme uvažovat třeba nad něčím jako je jednoduchý kooperativní multitasking. Pokud budeme stránkovat ve stránce 3 například, můžeme říct, že naše „aplikace“ musí být vždy napsány tak, aby běžely od adresy C000h, a pak jich můžeme do paměti nahrát až 7. No a při vhodné příležitosti, řekněme volání nějakých systémových funkcí, čas od času prostě uložíme stav dané aplikace (registry apod.) a přepneme stránku paměti.

Samozřejmě takový „multitasking“ vyžaduje velmi dobře napsané aplikace, které neudělají nic nečekaného, budou se chovat slušně, nebudou si zabírat systém pro sebe, budou k prostředkům přistupovat správně…

Existuje možnost, jak i na osmibitovém procesoru zařídit nějakou ochranu paměti, respektive obecně vyřešit úrovně přístupu k systémovým prostředkům? Osmibitové procesory samotné většinou nic takového neposkytují, ale je možno vnějšími obvody podobné chování nasimulovat.

Pokud znáte 16- a vícebitové procesory, které podobnými technikami oplývají, tak víte, že ochrana paměti před přepsáním či čtením od jiné běžící aplikace je jen část problému. Je zapotřebí nějak zajistit, aby se aplikace nepokoušela udělat něco, co dělat nemá, typicky třeba přepínat paměť nebo přistupovat k periferiím.

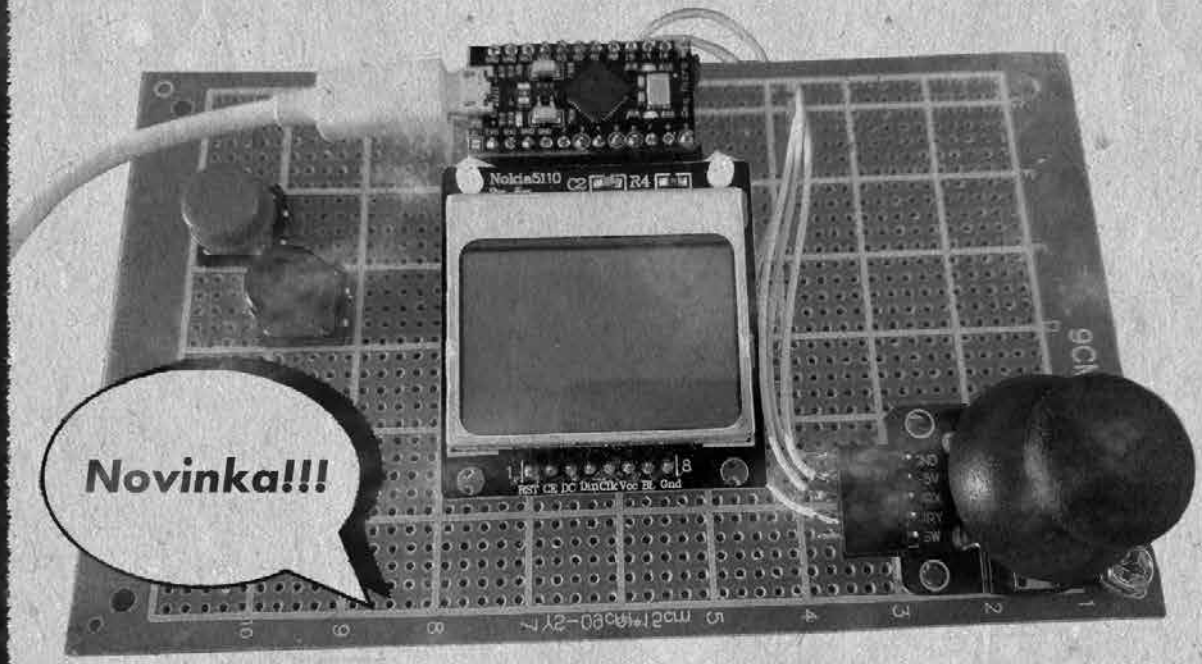
To znamená, že musíte určité „nebezpečné" instrukce aplikacím zakázat a povolit je pouze systému. Jenže osmibitový procesor nic takového neumí. Informaci o tom, jestli běží uživatelský program nebo systém musíte opět udržovat mimo procesor, ideálně tak, aby programátor neměl možnost sám od sebe tuto informaci změnit.

To znamená, že přechod do režimu „supervizor" (tedy „běží systém") musí být zajištěna přímo z návrhu – například tím, že procesor načte instrukci z konkrétní adresy v ROM. V tu chvíli se přepne někde klopný obvod a udržuje si informaci o tom, že systém běží v supervizorském módu. Zpátky do uživatelského se přepne například zápisem do nějaké k tomu určené periferie.

Jenže to není všechno. Systém musí správně odlišit, jestli se z ROM někdo pokouší číst, nebo zda si procesor žádá instrukci (naštěstí tuhle informaci procesory poskytují). Další věc k vyřešení je rozhodnout, co se má stát, když uživatelská aplikace udělá něco, na co nemá právo – například se pokusí přistupovat k periferii napřímo (to asi lze zakázat kombinační logikou) nebo třeba skočit přímo někam do ROM. To by asi dělat neměla, takže musíme detekovat pokus o čtení instrukce z této oblasti v neprivilegovaném režimu a asi vyvolat nemaskovatelné přerušení…

Zkrátka: není to úplně jednoduché, ale lze to nějak udělat. Otázka je, zda se to vyplatí.

11 OMEN Charlie



OMEN CHARLIE

Baví vás počítačové hry? Chcete si nějakou jednoduchou vlastní hru napsat? A chcete se přitom pocvičit v assembleru 8080 nebo 6502? Pak je pro vás ideální OMEN Charlie - jednoduchá herní konzole, připravená pro vaše vlastní experimenty. Vývoj her usnadní předpřipravená knihovna rutin. Kromě výběru procesoru je možné vybrat i displej a použít buď černobílý, nebo barevný.

11 OMEN Charlie

K tomu, abyste si na vlastní kůži vyzkoušeli, jak se s takovým osmibitovým procesorem pracuje, nemusíte nutně vlastnit počítač s tímto procesorem. Stačí vám bohatě výkonnější stroj a v něm *emulátor*.

Emulátor procesoru je vlastně jen jednoduchá rutina, napsaná v *nějakém* programovacím jazyce (C, assembler, JavaScript, Arduino, …), jejíž úkol je jednoduchý: přečíst operační kód z *paměti* a provést jej. To *provedení* záleží na konkrétním emulovaném procesoru a instrukci.

Určitě jste si už všimli, že instrukce mikroprocesoru nejčastěji buď přesouvají data odněkud někam, nebo s nimi vykonávají nějaké operace, popřípadě řídí stav programu (podmínky, skoky atd.) V emulátoru se vnitřní registry řeší pomocí lokálních proměnných, čtení a zápis do *paměti emulovaného procesoru* se nejlépe simulují obyčejným polem (nějaké *unsigned char memory[65536]*), no a řídicí instrukce buď mění stav registru PC (= interní proměnná v emulátoru), nebo nastavují nějaké příznaky (a ty jsou zase implementované jako proměnná).

V praxi se píšou emulátory o něco obecnější, takže zápis do paměti a čtení z paměti nebývá přímo uvnitř zadrátované, ale bývá řešeno pomocí *callback funkcí* memoryRead, memoryWrite apod. Díky tomu může nadřízený program simulovat např. zápis do obrazové paměti (a v realitě ho převést na ovládání bodů na displeji) nebo naopak nasimulovat, že do paměti ROM nelze zapsat…

Pokud emulujeme reálný počítač a jde nám o maximální věrnost, je potřeba, aby emulátor počítal strojové takty T. Emulátor většinou běží na mnohem výkonnějším stroji, takže obvyklý scénář vypadá tak, že se každou sekundu volá emulační rutina tak dlouho, dokud neproběhnou instrukce v délce např. dvou milionů T (pro procesor s pracovní frekvencí 2 MHz), a zbytek času je volno.

11.1 Emulátor osmibitového procesoru

Jeden takový emulátor procesoru 8080 jsem si napsal pro ASM80.com. Jeho zdrojové kódy jsou k dispozici na GitHubu:

https://8bt.cz/8080js

V kódu samotného emulátoru je vidět zmíněnou část, která se stará o emulaci vnitřních registrů procesoru:

```
var Cpu = function () {
   this.b = 0;
   this.c = 0;
   this.d = 0;
   this.e = 0;
```

```
    this.f = 0;
    this.h = 0;
    this.l = 0;
    this.a = 0;
    this.pc = 0;
    this.inte = 0;
    this.halted = 0;
    this.sp = 0xF000;
    this.cycles = 0;
    this.ram=[];
};
```

Za tímto kódem je několik pomocných funkcí, které se starají o práci s dvojicemi registrů, s různě adresovanou pamětí nebo s příznakovým registrem. Další funkce pak emulují základní operace na obecné úrovni (ADD, SUB apod.) – tyto funkce pracují s předanými hodnotami, nikoli s konkrétními registry, a starají se o společné efekty těchto operací (např. správné nastavení příznaků).

Srdcem emulátoru je funkce execute. Jejím parametrem je operační kód a hlavním blokem je obří switch s částí case pro každý platný operační kód.

```
  case 0x01:
     {       // LDX B,nn
       this.BC(this.nextWord());
       this.cycles += 10;
     }
     break;
   case 0x02:
     {       // STAX B
       this.writeByte(this.bc(), this.a);
       this.cycles += 7;
     }
     break;
   case 0x03:
     {       // INX B
       this.BC((this.bc() + 1) & 0xFFFF);
       this.cycles += 6;
     }
     break;      …
```

Díky pomocným funkcím (nextWord, writeByte) jsou jednotlivé obslužné rutiny velmi stručné.

Asi nejsložitější věc je správné nastavování příznaků u aritmetických a logických operací, ale s pomocí pár šikovných rutinek to je mnohem snazší. Příznaky S a Z jsou jednoduché: Z je 1, když je výsledek = 0, S je rovno nejvyššímu bitu výsledku. Příznak parity bývá trošku složitější, ale dobrý trik je nechat si paritu předpočítat a uložit do tabulky 256 hodnot. Největší problémy jsou s přenosem a přetečením, kde musíte porovnat výsledek a předchozí hodnotu a podle nich nastavovat. A zbytek, zbytek je už jen rutina.

11.2 Emulace procesoru v Arduinu / AVR

Mikrokontroléry AVR jsou velmi oblíbené stroje pro emulování starých procesorů. Jejich výkon je dostatečný pro emulaci 6502 nebo 8080 a s taktovací frekvencí okolo 20 MHz zvládají spoustu věcí, včetně generování videosignálu pro televizi. Existují tak třeba emulátory počítače Apple II nebo československého PMD-85.

Ostatně následující seznam odkazů hodně napoví:

- Emulace Sinclairu ZX81 a ZX Spectra v AVR, počítač se zabudovaným BASICem: http://www.jcwolfram.de/projekte/avr/main.php
- PMD-85 v Arduinu: http://pmd85.topindex.sk/
- Arduino 6502 – emulátor procesoru 6509: https://github.com/davecheney/arduino6502
- Apple][v Arduinu: http://damian.pecke.tt/turning-the-arduino-uno-into-an-apple
- Amatérský počítač Mouse6502 v Arduinu: https://github.com/mkeller0815/MOUSE2Go
- Počítač s CP/M v Arduinu Due (verze s procesorem ARM): https://github.com/ptcryan/CPM_Due
- Počítač s CP/M a procesorem Z80, emulovaný Arduinem Due: https://hackaday.io/project/19560-z80-cpm-computer-using-an-arduino
- Z80 emulovaný pomocí ARM s funkčním systémem CP/M: http://spritesmods.com/?art=avrcpm&page=4
- Další verze CP/M a emulace procesoru v Arduinu Due: https://weblambdazero.blogspot.cz/2016/07/cpm-on-stick_16.html
- 6502+BASIC v ESP8266 / STM32: http://forum.arduino.cc/index.php?topic=193216.msg2369600#msg2369600
- Emulátor 6502 s video výstupem v ATMega: https://sites.google.com/site/retroelec/hardware

Při emulaci nejste omezeni pouze na Arduina. Mnoho konstrukcí používá obvody ATMega vyšších řad, především kvůli větším pouzdrům, které usnadňují připojení externí paměti. Některé konstrukce používají třeba populární obvod ESP8266 se zabudovaným WiFi nebo velmi levné vývojové desky s procesorem ARM typu STM32F103 (zvané BluePill).

11.3 Praktické cvičení: Emulátor systému s procesorem 8080 v Arduinu

Je to jednoduché. Vezměte si libovolné Arduino Uno. Použijete kód emulátoru 8080, co je pod licencí GPLv2 dostupný v rámci emulátoru Altairu 8800.

https://github.com/companje/Altair8800

Stačí jen samotný emulátor.

V Arduino IDE si napište jednoduchý HAL – Hardware Abstraction Layer. V něm se definuje, jak se čte z paměti, zapisuje do paměti a jak se pracuje s periferiemi. Já si nadefinoval, že v prostoru 0x0000-0x0fff bude ROM (a k tomu je potřeba direktiva PROGMEM), v prostoru 0x1000-0x13ff bude 1 kB RAM (vlastně jen pole hodnot typu byte).

RAM by šla zvětšit na 1.5 kB, i když překladač píše varování. Zkuste si to, ale nezapomeňte změnit příslušné hodnoty ve zdrojáku BASICu.

Otestujete si, jak to běhá, jednoduchým příkladem ve strojáku do ROM.

Můžete přidat obsluhu portů. Port 1 bude sériový port, port 0xFE bude ovládat LEDku na Arduinu. Jen tak, pro radost. Krátkým kódem ve strojáku ověříte, že to funguje.

Můžete zkusit přeložit Tiny BASIC. Zdrojové kódy jsou k dispozici, tak stačí upravit jen tu část, kde se komunikuje se sériovým portem.

K překladu můžete použít zdrojové kódy z GitHubu a online assembler ASM80.com. Součástí tohoto assembleru jsou i nástroje, které dokáží z přeloženého kódu vytvořit „pole čísel". Když jim dodáte formát, jaký má definice pole v C, můžete výsledek pomocí #include přidat do výsledného *.ino*.

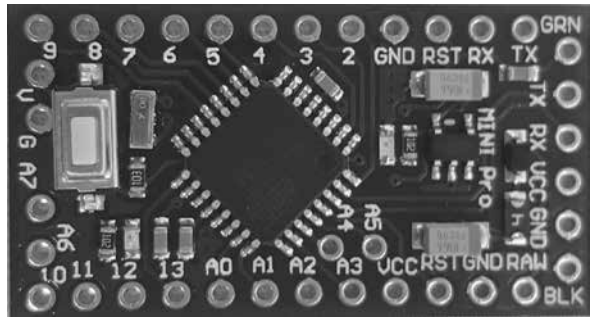
Můžete si stáhnout zdrojové kódy a zkusit je upravit. Můžete si nastavit stejné adresy jako v počítačích OMEN, můžete si zkusit nahradit emulaci procesoru 8080 emulací procesoru 6502, můžete si zkusit připojit periferie podle svých schopností a možností. Můžete experimentovat i s výkonnějšími stroji – nabízí se Arduino Due, popřípadě BluePill či ESP8266.

11.4 Konzole do ruky

Vezmeme si Arduino Micro, displej z mobilu Nokia 5110 a šest tlačítek (4 směry, 2 akční tlačítka), a z nich si sestavíme úplně jednoduchou herní konzolu. Displej má závratné rozlišení 84x48 bodů monochromaticky, takže žádná sláva, ale různé hříčky jako Had či Pac Man, Auta, Flappy a podobné jednoduché hry není problém udělat.

Berte prosím herní konzolu jako „malý počítač bez klávesnice" – ostatně společnost Atari přesně takto své první počítače udělala.

Pokud použijeme Arduino Micro Pro, budeme potřebovat 6 pinů pro tlačítka, 5 pinů pro řízení displeje a 1 pin pro zvuk.



Obrázek 44: Arduino Mini Pro

Určitě by bylo nejlepší naprogramovat nějaké hry v jazyce C a přeložit přímo pro Arduino. Ale když už máme ověřené emulátory procesorů 8080 a 6502, tak proč nepoužít je?

Arduino bude sloužit jako vstupně-výstupní zařízení. Dokáže emulovat sériové rozhraní, dokáže načítat hodnoty tlačítek a joysticku, a když na to přijde, může generovat i jednoduché tóny. Tyto funkce zpřístupníme emulovanému procesoru, třeba ve formě „virtuálních portů".

Co se týče obrazové paměti, můžeme použít stejný princip, jaký byl použitý například u ZX Spectra – obsah obrazovky byl uložený v paměti a každému bodu odpovídal jeden bit. Náš displej má rozlišení 84x48 bodů, což znamená při osmi bodech na bajt kapacitu 504 bajtů. Tedy téměř půl kilobyte.

Pro emulovaný procesor zůstává v RAM Arduina nejvýš tak 1,5 kB volného místa. Pokud si půl kilobyte „zaplácneme" video RAM, zůstává 1 kB na samotné výpočty, a to není mnoho. Na druhou stranu, u některých domácích počítačů měla celá paměť 1 kB, a dalo se s tím dosáhnout mnohého.

Druhá možnost, jak se postavit k obrazu, je taková, že řekneme, že obraz je „write only“, tedy že do něj může procesor jen zapisovat. Můžeme pak emulátor nastavit tak, že třeba od adresy C000h do adresy C1F8h bude obrazová paměť, a případný zápis do této paměti půjde ve skutečnosti na displej. Zvýší se nám tak dostupná RAM, ale na druhou stranu přijdeme o možnost měnit obsah displeje, bude muset být vygenerován pokaždé znovu.

Konzoly, říkáte?

Třetí možnost je inspirovat se reálnými starými konzolami, třeba jako Atari 2600. Tato herní konzola používala pro generování obrazu jeden specializovaný obvod TIA, který uměl vygenerovat právě jeden řádek na displeji (ve skutečnosti se generovaly dva stejné televizní řádky). Generoval ho na základě několika málo hodnot, které programátor uložil do jeho registrů.

Na každém řádku obrazu se objevovala herní plocha (playfield), rozdělená na levou a pravou polovinu. Každá polovina měla rozlišení 20 pixelů, přičemž pravá polovina mohla být buď opakováním, nebo zrcadlovým obrazem levé poloviny. Plocha používala tři osmibitové registry, v nichž byla bitová maska, která určovala, jestli se na daném místě zobrazí barva pozadí, nebo barva herní plochy.

Na řádku se mohly objevit následující objekty: Hráč 0, Hráč 1, Střela 0, Střela 1 a Míč (Player, Missile, Ball). Střely jsou prosté horizontální čárky a programátor zadává jen jejich pozici na řádku a šířku (1, 2, 4 nebo 8 bodů). Míč (ball) má vlastnosti podobné jako střela, ale může být posunutý o jeden TV řádek vertikálně.

Hráči byly komplexnější objekty: mohli jste zadat osmibitovou bodovou masku a mohli jste zapnout různé módy opakování (dvakrát vedle sebe, třikrát, dvojnásobná šířka bodů apod.)

Všechny tyto objekty měly přiřazenou barvu a svou pozici na řádku (Atari 2600 pracovalo s horizontálním rozlišením 160 bodů). Programátor tedy nastavil výše uvedené hodnoty do obvodu TIA, a obvod vygeneroval jeden řádek (tedy dva televizní). Programátor mezitím připravil nové hodnoty a jelo se znovu.

Vypadá to velmi komplikovaně, a při prvních experimentech to asi komplikované bude, ale dá se na to zvyknout. Obvod TIA navíc informoval programátora o tom, jestli na daném řádku došlo ke kolizi mezi jednotlivými objekty, třeba mezi hráčem a střelou, mezi hráči navzájem, mezi hráči a míčem, mezi míčem a střelou atd., předával informace o tom, jak jsou nastavené ovladače, a zároveň generoval zvuk.

Velká výhoda tohoto přístupu byla, že programátorovi stačilo nastavit pro každý řádek jen několik málo údajů. Nevýhoda byla, že musel řešit, který řádek se právě vykresluje a jaké údaje má tedy poslat, a to celé musel stihnout v poměrně malém čase. Ale pro Atari 2600 vznikly tisíce her, takže to zcela zjevně šlo. Navíc v základním nastavení měl programátor k dispozici pouhých 128 bajtů RAM…

Konkrétní implementaci nechám na vás. Vždyť – je to vaše konzola! Vy ji budete programovat! Můžete se inspirovat mým řešením, je na GitHubu, ale budu rád, když vymyslíte lepší.

Asi jsem to ještě nezmínil, ale všechny konstrukce v této knize jsou k dispozici pod svobodnými licencemi typu MIT (software), CC (texty) nebo CERN OHL (hardware). To znamená, že i hardware můžete libovolně používat, přepracovat, vylepšovat a dále šířit dle vlastního uvážení, pokud uvedete autora. Nikdo vás nenutí, abyste své úpravy zveřejňovali, ale pokud to uděláte, budu velmi rád.

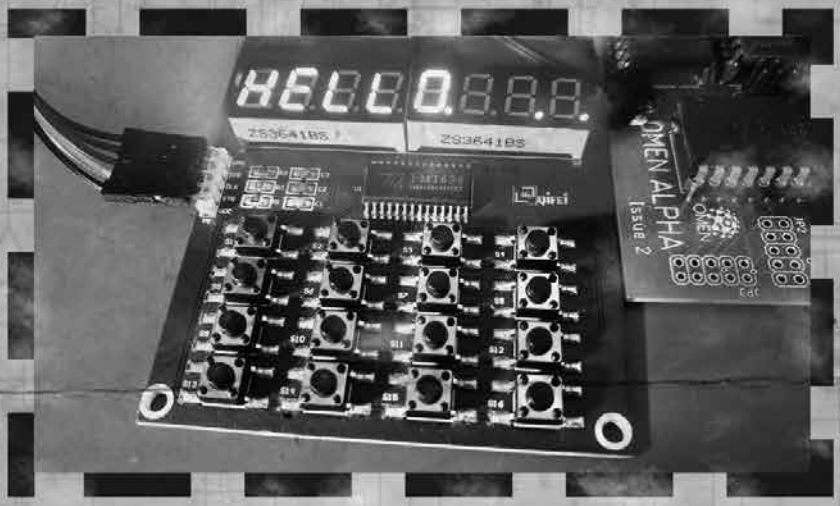
Přihodím proto jen pár námětů na vylepšení.

Můžete použít barevný displej – za příznivé ceny lze sehnat barevný displej 128x128 bodů, který se připojuje přes zjednodušené rozhraní SPI.

Budete pravděpodobně chtít i víc operační paměti. Můžete sáhnout třeba po sériových RAM, anebo po větším procesoru, jako je už zmíněný BluePill s procesorem STM32F103C8. Překlad knihoven i připojení periferií je bez problémů a získáte tím velmi rychlý (72 MHz) procesor s 20 kB RAM.

12 Omen Delta (koncepce)

OMEN DELTA



"Profesionální systém CP/M, praotce osmibitových operačních systémů, už nemusíte emulovat. OMEN Delta vám nabízí jedinečné spojení původní technologie s moderním přístupem. 128 kB RAM (část použitelná i jako RAMdisk), výkonný procesor Z80 pro naprosto perfektní kompatibilitu, moderní LCD displej a místo starých nespolehlivých disket paměťové karty - to je Delta, jednoduchá konstrukce profesionálního počítače z 80. let!"

Jen pár čipů!

KSTLER

12 Omen Delta (koncepce)

12.1 Spojení starého a nového

Na rovinu přiznávám, že u Delty jsem se inspiroval zapojením uživatele „Just4Fun“ či „SuperFabius“, které zveřejnil na serveru Hackaday:

https://8bt.cz/z80

Jeho zapojení tvoří čtyři integrované obvody: Procesor Zilog Z80, paměť RAM 128 kB (ale využívá se jen polovina), mikrokontrolér Atmel AVR ATMega32 a obvod 7400, jehož nejdůležitější funkcí je správně zastavit procesor, když se pokouší přistupovat k periferii.

Samozřejmě že by šlo postavit klon Alphy, jen s procesorem Z80, a vše by fungovalo. Na výše odkazovaném zapojení mě zaujalo právě to, že se na něm snadno demonstruje práce více procesorů v jednom systému.

12.2 Jak to funguje?

V systému jsou dva procesory a jedna paměť RAM, která zabírá celý prostor. Paměť ROM chybí, respektive není potřeba. Procesor Z80 je ten, který vykonává vlastní program, a ATMega je v roli koprocesoru, ovšem tento koprocesor má v systému větší „váhu“.

Může to vypadat zvláštně, ale zas tak překvapivé to není. Bývalo téměř pravidlem, že například videoprocesor měl při přístupu k paměti přednost a procesor čekal. Je to logické: videoprocesor nepočká, musí posílat obrazová data na výstup stále, a kdyby neměl přednost, obraz by se rozsypal. U ZX Spectra tak obvod ULA, který mj. generoval právě obraz, měl při přístupu k paměti prioritu, a pokud náhodou procesor v tu chvíli přistupoval ke stejné oblasti v paměti, ULA ho prostě pozastavila.

ATMega proto řídí skoro vše, co se s procesorem Z80 děje. Generuje pro něj hodinové pulsy, řídí jeho chod (tj. zastavuje a odpojuje od sběrnice), na začátku práce nahraje do paměti potřebné programy a v průběhu práce funguje jako univerzální vstupně-výstupní zařízení.

Tím se dostáváme k funkci obvodu 7400. Jeho jedna polovina (dvě hradla) budí LED, které signalizují stavy WAIT a DMA. Druhá polovina slouží jako klopný obvod RS, jehož funkce je následující:

Pokud procesor Z80 hodlá přistupovat k periferiím (aktivuje signál IORQ), klopný obvod se přepne a rovnou procesor uvede do stavu WAIT. To znamená, že procesor nechá na adresové sběrnici

vystavenou adresu, popřípadě data na datové sběrnici při zápisu. Zároveň se tím pošle signál do ATMegy, která má nyní čas zjistit, co vlastně procesor chce (datová i adresová sběrnice Z80 je přivedena na její vývody) a buď načíst data z datové sběrnice, nebo naopak požadované vystavit. Když je hotovo, ATMega pošle signál BUSRQ, vynuluje klopný obvod (čímž zruší stav WAIT), počká na načtení dat procesorem Z80, pokud šlo o operaci čtení, pak se od sběrnice opět odpojí, aby nerušila, a nakonec zruší stav DMA (BUSRQ).

Mezikrok s požadavkem na sběrnici BUSRQ využívá toho faktu, že zatímco signál WAIT drží procesor uprostřed instrukce tak, aby na sběrnici byla požadovaná data, tak do stavu DMA (Bus Request) se vstupuje až na konci celého instrukčního cyklu, to znamená poté, co jsou data přenesena. Tímto trikem se tedy docílí toho, že procesor Z80 přenese data a zastaví se. ATMega se pak bezpečně odpojí od sběrnice a nechá Z80 pokračovat.

Boot, tedy úvodní zavedení programu pro Z80, probíhá dvoufázově. V první fázi začíná ATMega, která odpojí Z80 od sběrnice a nahraje krátký zaváděcí program do paměti. Pak povolí Z80 práci a resetuje ho. Procesor Z80 začne provádět tento kód, který načítá vlastní programové vybavení z paměti FLASH v ATMega pomocí vstupně-výstupních operací, popsaných výše. Tím se nahraje vlastní firmware do paměti. Zároveň je to odpověď na otázku „jak to, že to funguje bez ROM?“ (Samozřejmě je riziko, že si programem přepíšete část firmware a celý systém zhavaruje.)

Pojďme se teď podrobněji podívat na procesor Z80, ale než tak učiním, dovolte jednu autorskou vsuvku.

12.3 Autorská vsuvka

Samozřejmě že bych vám nejradši prozradil úplně všechno, co mě kolem osmibitů napadá. Rád bych vám ukázal všechny možné procesory a vysvětlil vám, jak si takový vlastní procesor zkonstruujete, a to nejen teoreticky, ale i prakticky! Jenže pak by tato kniha měla skoro dvojnásobný rozsah, proto si něco budu muset nechat do pokračování. Vydavatel slíbil, že drobné pikantérie dokáže vydat v nějakém menším nákladu pro zájemce, tak mi to připadá jako vhodnější řešení než zbytečně prodražovat tuto knihu.

Bohužel to s sebou přineslo nutnost rozhodnout, které konstrukce zůstanou v této knize a které se přesunou do pokračování. Stejně tak rozhodnout, který procesor zůstane a který bude muset počkat.

Když jsem se o této knize zmiňoval lidem, všichni měli za nezpochybnitelné a *ložené*, že v knize bude velká část věnována procesoru Z80. Proč? No přeci – to je jasný! Byl ve Spectrech, v Amstradu, ve Sharpu i Sordu, v MSX i v počítačích s CP/M, spousta lidí ho tu zná, je kompatibilní s procesorem 8080, existuje pro něj spousta materiálů, třeba Bity do bytu nebo různé klubové materiály, v knížkách je zmiňován, takže vlastně je jasné, že tu být musí.

Když jsem nad tím přemýšlel hlouběji, musel jsem dát lidem zapravdu. Ano, v různých knihách o mikroprocesorech, co tu vyšly, je procesor Z80 popisován, měl seriál v Amatérském rádiu, klubové zpravodaje ho popisovaly zleva zprava, existuje velké množství konstrukcí… Ale není to spíš argument pro to, že tu být nutně nemusí?

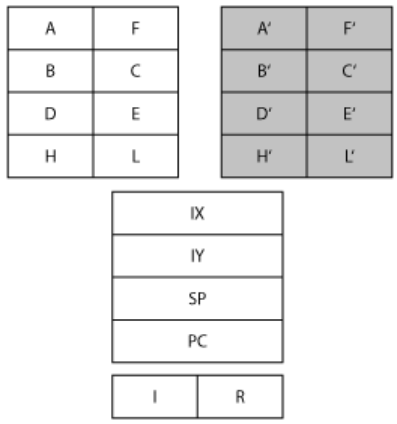
A tak, s plným vědomím, že jsem právě naštval spoustu čtenářů, říkám: Místo opakování toho, co napsali o procesoru Z80 jiní, jinde, mnohokrát a různými způsoby, jsem se rozhodl, že Z80 věnuju jen obecnou přehledovou kapitolu a zájemce o podrobnosti odkážu na výše zmíněné pokračování. Namísto toho si popíšeme jiný procesor, který u nás není tak moc rozšířený, používaný a dokumentovaný, ale je to určitě škoda.

Do dodatků se tím dostává i OMEN Delta…

12.4 Architektura procesoru Zilog Z80

Z80 je mikroprocesor, který výrazně vylepšoval koncept 8080, ale zároveň udržel kompatibilitu. Můžete vzít přeložený program pro 8080, a nejspíš vám bude fungovat i na Z80 (výjimkou jsou programy, které počítaly s tím, že instrukce trvají nějaký přesný počet cyklů – na Z80 jsou většinou rychlejší). Procesor Z80 nepotřebuje podpůrné obvody a na rozdíl od 8080 si vystačí s jedním napájecím napětím (procesor 8080 potřeboval +5, +12 a -5 voltů, navíc zapínané v určitém pořadí).

Z programátorského hlediska vycházel Z80 přímo z 8080, ale rozšiřuje jeho možnosti o mnoho instrukcí, včetně posunů či blokových přenosů dat, přidává druhou sadu registrů, možnost využít indexování pomocí speciálních registrů IX, IY, navíc integruje i obvody, které usnadňují připojení dynamických pamětí.

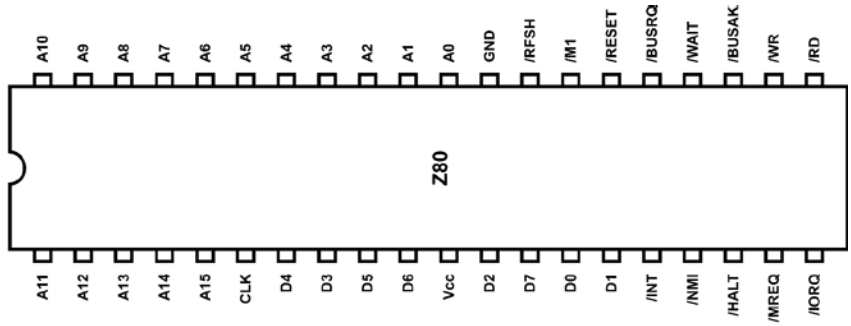


Obrázek 45: Registry Z80

Sada registrů je podobná 8080 či 8085. Druhá, stínová sada registrů (na obrázku šedé, označovaly se zápisem s apostrofem) nebyla přístupná přímo, ale pomocí určitých instrukcí šlo obě sady „přehodit“ a pracovat s alternativní.

Kromě indexových registrů IX a IY přibyl registr R, který slouží k výše zmíněnému obnovování dynamických pamětí, a registr I, v němž je uložena vyšší polovina adresy vektorů přerušení (pouze ve speciálním přerušovacím módu IM2).

Z80 dále vylepšuje například subsystém přerušení, vystačí si s jedním prostým hodinovým signálem, navíc může fungovat na vyšší frekvenci než 8080A – a vůbec s ním byla radost pracovat. Procesor Z80 se tak stal základem mnoha počítačů té doby, od známých Sinclairových počítačů ZX až po poloprůmyslové počítače se systémem CP/M. V Československu byl méně rozšířený než 8080, to proto, že klon Z80 se začal vyrábět až později v tehdejším NDR (pod označením U880D / UB880D), zatímco klon 8080 vyráběla přímo TESLA.



Všimněte si, že základ zůstává stejný: osmibitová datová sběrnice, šestnáctibitová adresová… Další vývody řídí sběrnici a tok dat.

Na co měl 8080 signály /MEMR, /MEMW, /IOR a /IOW, na to má Z80 signály /MREQ, /IORQ, /RD a /WR (Memory Request, IO Request, Read, Write) a vždy kombinace dvou signálů určuje, jaká operace probíhá. Rozdíly popisuje následující tabulka, která ukazuje, jaké signály jsou aktivní pro jakou činnost:

Činnost	**8080**	**Z80**
Čtení z paměti	/MEMR	/MREQ + /RD
Zápis do paměti	/MEMW	/MREQ + /WR
Čtení z periferie	/IOR	/IORQ + /RD
Zápis do periferie	/IOW	/IORQ + /WR

/WAIT

Obdoba signálu READY, ale u Z80 se jmenuje /WAIT. Pokud je v logické 0 poté, co se procesor pokouší přistoupit k paměti nebo periferii, zastaví se, udržuje informace na datové sběrnici a čeká, až periferie potvrdí, že stihla data zpracovat a vrátí signál /WAIT zpět do logické 1. Procesor nijak nepotvrzuje, že čeká.

DMA

U procesoru Z80 žádá okolí o odpojení od sběrnice pomocí signálu /BUSRQ (Bus Request), procesor jej potvrzuje signálem /BUSACK (Bus Acknowledge)

Halt

V procesoru Z80 existuje (stejně jako u 8080) instrukce HALT. Jakmile ji procesor vykoná, dostává se do čekacího stavu, v němž stále dokola vykonává prázdnou instrukci NOP (No OPeration). Z tohoto stavu jej vyvede až RESET nebo přerušení. K čemu to je dobré? Většinou se tato instrukce používala k čekání na nějaké vnější přerušení, až některá z periferií dokončí svou práci. To, že je Z80 ve stavu HALT, dává najevo výstupním signálem, který se, nepřekvapivě, jmenuje /HALT.

/M1

Tento výstup spolu se signálem /MREQ oznamuje, že procesor právě teď čte z paměti operační kód instrukce – probíhá cyklus FETCH. (Pokud je tento signál aktivní s /IORQ, znamená to, že očekává přerušovací vektor, viz dál.)

/RFSH

Poté, co procesor vyzvedne z paměti operační kód instrukce, začne ji dekódovat. V tu dobu nepotřebuje přistupovat k vnějším zařízením, tak návrháři Z80 přišli s nápadem, že během této doby pomohou občerstvovat dynamické paměti. Dynamické paměti, připomeňme si, potřebují poměrně často obnovovat informace, které jsou v nich uložené. Obnovení probíhá tak, že se „naprázdno“ přečte jeden řádek informace. Pokud není určitý řádek přečten po delší dobu, informace se poškodí – a „delší doba“ jsou třeba jednotky milisekund (u oblíbeného obvodu 4116 to byly dvě milisekundy). Proto procesor obsahuje speciální osmibitový registr R (Refresh), jehož hodnota se každým instrukčním cyklem zvyšuje o 1. Nejvyšší bit se nikdy nemění, takže efektivně čítá hodnoty 0-127 (resp. 128 – 255). Ale to nevadilo, protože většina dynamických pamětí v té době měla sedmibitové adresy řádků. Po cyklu M1 tedy procesor pošle na adresovou sběrnici sedm bitů z registru R a aktivuje signál /RFSH.

Přerušení

Procesor Z80 má dva různé typy přerušení – maskovatelné a nemaskovatelné. Maskovatelné přerušení odpovídá tomu, co známe z procesoru 8080, ovšem v procesoru Z80 máme k dispozici tři módy, které může programátor přepnout speciální instrukcí. V módu 0 je funkce totožná s procesorem 8080. V módu 1 je funkce zjednodušená: přerušení vždy vyvolá skok na adresu 0038h. V módu 2 se po přerušení čte z datové sběrnice takzvaný „přerušovací vektor“, což je obecně

osmibitová hodnota. Uvnitř procesoru se tato hodnota použije jako dolní polovina adresy. Horní polovina se vezme z registru I (ten předtím nastaví programátor). Tím se získá adresa v paměti, z níž se postupně přečtou dva bajty (z buňky na této adrese a z buňky na adrese o 1 vyšší). A tyto dva bajty udávají adresu, na které se nachází přerušovací rutina. Tento mód je tedy nejflexibilnější, ale také nejnáročnější.

Každopádně maskované přerušení může programátor zakázat – „zamaskovat" – pomocí speciální instrukce DI (Disable Interrupt). Tato instrukce nastaví vnitřní příznak v procesoru. Vstup /IRQ (Interrupt Request) je tímto příznakem „maskován" – pokud je přerušení zakázáno, signál se nedostane dál a požadavek je ignorován. „Odmaskovat" přerušení lze instrukcí EI (Enable Interrupt).

Kromě maskovaného přerušení /IRQ existuje i signál nemaskovaného přerušení /NMI (Non Maskable interrupt). Tento signál vždy vyvolá přerušení, a donutí procesor odskočit na adresu 0066h.

- http://www.ecstaticlyrics.com/electronics/Z80/system_design/
- https://hackaday.io/project/7354-zaviour-board-avrz80-hybrid

12.5 Assembler Z80 ve zkratce (a se zvláštním přihlédnutím k instrukcím 8080)

O programování v assembleru Z80 vyšlo v češtině velké množství publikací, proto jsem dospěl k názoru, že není potřeba zacházet do velkých podrobností a zájemce odkázat například na knihu Ladislava Zajíčka „Bity do bytu" nebo na publikaci „Assembler a ZX Spectrum" od Tomáše Vilíma (známého pod značkou Universum).

Už jsem psal, že Z80 je zpětně binárně kompatibilní s procesorem 8080. To znamená, že například instrukce s kódem CDh představuje skok do podprogramu a instrukce s kódem 7Eh přenese do akumulátoru (registru A) hodnotu z paměti. Rozdíl je ale v pojmenování většiny instrukcí.

Důvod je prý obchodní: Intel tvrdil, že mnemotechnické názvy instrukcí procesoru 8080 jsou jeho autorským dílem a nikdo jiný je nesmí použít, takže v Zilogu zvolili nejjednodušší způsob, totiž nechali instrukce stejné a dali jim jiná pojmenování.

První zásadní rozdíl je, že dvojice registrů se neoznačují jménem vyššího, jako u 8080 (třeba v instrukci PUSH B), ale oběma písmeny – BC, DE, HL (Z80: PUSH BC).

Druhý rozdíl je v tom, že se začaly používat závorky okolo operandů všude tam, kde se odkazuje na paměť. S tímto pravidlem také přišlo to, že místo lehce mystického registru M se v assembleru Z80 objevuje zápis (HL).

A konečně třetí, největší rozdíl: snad všechny instrukce, které odněkud někam přesouvají data, se jmenují „LD“ (ze slova „load“).

8080	Z80	Význam
MOV A,B	LD A,B	A <- B
MOV A,M	LD A,(HL)	A <- paměť (HL)
MVI A,55h	LD A, 55h	A <- 55h
LDA 1234h	LD A, (1234h)	A <- paměť (1234h)
STA 1234h	LD (1234h), A	paměť(1234h) <- A
LXI H,1234h	LD HL, 1234h	HL <- 1234
LDAX B	LD A, (BC)	A <- paměť (BC)

Na jednu stranu se tím syntax zjednodušila a zpřehlednila, na druhou stranu je potřeba si pamatovat, jaké kombinace operandů lze použít a jaké ne.

Podobně byly například sjednoceny aritmetické instrukce – sčítání je vždy ADD (nebo ADC), ať se sčítají registry (ADD A,B), přičítá konstanta (ADD A,23h) nebo sčítají dvojice registrů (ADD HL, DE).

Z80 používá 252 operačních kódů pro běžné instrukce a čtyři kódy jako takzvané prefixy. Kódy, které mají u 8080 definovanou funkci, zůstaly zachovány. Volná místa mezi nimi zaplnily například instrukce relativních skoků JR (které dokážou skočit v rozsahu -128 až +127 byte od aktuální adresy), instrukce DJNZ, která sníží hodnotu v registru B o 1 a pokud je nenulová, udělá relativní skok, podobně jako instrukce JR, instrukce EX AF,AF‘, která prohazuje zmíněnou dvojici registrů za její „stínovou“ variantu apod.

Kódy DDh a FDh slouží k adresování pomocí indexových registrů IX a IY. Většinou tak, že nahrazuje adresování pomocí registrů HL hodnotou z IX (IY), k níž je připočítán posun (osmibitové číslo se znaménkem). Například instrukce 7Eh je LD A,(HL) – u 8080 se jmenuje MOV A,M. Pokud ale zapíšeme před tuto instrukci prefix DDh, tj. DDh 7Eh, stane se z ní instrukce LD A,(IX+d). Hodnota „d“ (displacement) je výše zmíněný posun a je uložena ve třetím bajtu instrukce. Kompletní znění je tedy např. DDh 7Eh 12h, mnemotechnické vyjádření „LD A,(IX+12h)“ a význam je „vezmi hodnotu z registru IX, přičti 12h a výsledek ti dá adresu. Hodnotu z této adresy zkopíruj do registru A“.

Takto šlo „oprefixovat“ instrukce, které využívaly dvojregistr HL. Co se stane, když tento prefix dáte před instrukci, která pracuje jen s registrem H nebo L? Zilog o nich mlčí, v datasheetu jsou

na místě těchto instrukcí prázdná místa, ale poměrně rychle se ukázalo, že tyto instrukce dokáží pracovat s registry IX a IY po polovinách (IXH, IXL, IYH, IYL)

Další prefix, CBh, přinášel řadu rotací, posunů a bitových operací. Rotace byly už u 8080, u Z80 můžete rotovat libovolným osmibitovým registrem, včetně (HL), tedy obsahem paměti. Existují i duplikáty instrukcí pro registr A, které nastavují jinak příznaky.

Posuny u 8080 nebyly – jsou to instrukce, které se chovají podobně jako rotace, jen přetékající bit se nevrací zpátky. Prázdné místo je buď zaplněno nulou (SLA, SRL), nebo kopií nejvyššího bitu (SRA). Existuje i nedokumentovaná instrukce SLL (SLIA), která posouvá obsah registru doleva a doplňuje v nejnižším bitu jedničkou.

Bitové operace představuje sada instrukcí bit, set a res, například BIT 4,H nebo SET 6,(HL). Instrukce SET nastavuje konkrétní bit na 1, instrukce RES daný bit nuluje a instrukce BIT přenáší jeho hodnotu do příznaku Z.

Poslední prefix EDh přináší jen pár instrukcí, zato velmi zajímavých. Například šestnáctibitové přenosy, jako LD (1234h), BC. Zde jsou i speciální instrukce pro návrat z přerušení, nastavování přerušovacích módů, instrukce přístupu k periferiím – tedy jako IN a OUT u 8080, ale u Z80 můžete adresovat pomocí registru C a vysílat a číst libovolný osmibitový registr. Po pravdě to není tak úplně přesné, protože např. při instrukci OUT (C),H se na adresovou sběrnici neposílá jen obsah registru C, ale kompletní obsah registrů BC. Proto lze u Z80 použít pro periferie šestnáctibitový adresovatelný prostor.

Kromě těchto instrukcí jsou zde i instrukce blokového přesunu, blokového porovnání, blokového vstupu a blokového výstupu. Jedná se o poměrně komplexní instrukce, které používají většinu obecných registrů. Posuďte sami:

Instrukce LDI udělá následující:

1. Přečte jeden byte z paměti na adrese (HL)
2. Zapíše tento byte do paměti na adresu (DE)
3. Zvýší HL o 1
4. Zvýší DE o 1
5. Sníží BC o 1

Její varianta s opakováním, instrukce LDIR, opakuje tyto kroky, dokud není BC rovno nule. Varianty LDD, LDDR fungují stejně, jen s tím rozdílem, že se adresy v DE a HL nezvyšují, ale snižují o 1.

Další sada instrukcí funguje jako sofistikovanější varianta instrukcí OUT a IN. Například OUTI vezme obsah z (HL), pošle ho na port, jehož adresa je v registru C, zvýší HL o 1 a sníží B o 1. O krok dál je instrukce OTIR, která opakuje OUTI, dokud je B nenulové.

K OUTI existuje dekrementační varianta OUTD, která adresu v HL snižuje o 1. OTDR opakuje OUTD, dokud je B nenulové.

Analogicky k této čtveřici existují instrukce INI, INIR, IND a INDR.

Poslední sada blokových operací slouží k hledání hodnoty v bloku paměti. Instrukce CPI provede následující kroky:

1. CP (HL) – porovná hodnotu v registru A s hodnotou v (HL)
2. INC HL
3. DEC BC

V příznaku Z je 1, pokud se hodnota v A shodovala s obsahem paměti na adrese HL (před zvýšením).

Instrukce CPIR provádí CPI, dokud je BC nenulové, nebo pokud nenajde buňku, v níž je stejný obsah jako v registru A (Z=1).

CPD a CPDR jsou opět varianty, které postupují směrem k nižším adresám, tedy v každém kroku hodnotu v HL snižují.

Pokud jste přešli z procesoru 8080 k Z80, otevřel se před vámi naprosto úžasný svět! Těch možností, taková krása… Pamatuju se, že mě úplně uchvátily blokové operace. Až později jsem si uvědomil, že jsou vlastně velmi pomalé, a naučil se spoustu triků, jak přenášet obsah paměti mnohem rychleji. Ale i tak je Z80 velmi výkonný a komplexní mikroprocesor s pokročilými možnostmi.

13 Intermezzo obrazové

13 Intermezzo obrazové

Pokud jste v osmdesátých letech měli domácí počítač, měli jste ho pravděpodobně připojený k televizi, která sloužila jako monitor. U nás to pravděpodobně byla černobílá televize, u barevné hrálo roli ještě to, jestli pracovala jen v normě SECAM, nebo jestli uměla i „západní" normu PAL.

Počítač se staral o to, aby vytvořil obraz v takové podobě, kterou dokázala televize zpracovat. Což nebylo úplně jednoduché. Považte sami:

Počítač míval většinou vyhrazenou oblast paměti pro informace o obraze, takzvanou video RAM. Z obsahu paměti v této oblasti generoval černobílý nebo barevný signál, a to tak, že posílal obraz po jednotlivých mikrořádcích. Takto vzniklý surový videosignál jste mohli připojit k televizi s přímým video vstupem – ale takové moc nebývaly, takže většina počítačů té doby měla zabudovaný ještě takzvaný modulátor. Modulátor z videosignálu udělal namodulováním na vysokofrekvenční signál ekvivalent toho, co přijímač chytal z antény. Anténním kabelem se tento signál přenesl do anténního vstupu televize, která si z něj opět demodulovala původní videosignál, a ten zobrazila. Kvalita samozřejmě tím modulačním mezikrokem utrpěla, proto se šířily návody, jak modulátor v počítači obejít a jak v televizi najít vstup, kam lze připojit přímo videosignál.

13.1 Černobílý videosignál PAL/SECAM

Samotný signál je vlastně velmi jednoduchý (pokud je řeč o černobílém obrazu), ale chce to trošku pozornosti.

Obraz se rozkládá na jednotlivé body v mřížce. Počty řádků se liší podle normy, v normě PAL to je 312,5 řádku. Videosignál se přenáší postupně po jednotlivých řádcích zleva doprava, od nejvyššího řádku k nejnižšímu. Přenos každého řádku začíná zatemněním – signálem „černá barva" (front porch) a horizontálním synchronizačním pulsem, díky němuž se obvody připraví na vykreslování jednoho řádku. Po synchronizačním pulsu následuje chvíli signál „černá barva" (back porch) a pak následuje přenos samotných obrazových dat (velmi jednoduše kódováno: čím vyšší napětí, tím vyšší jas). Tím končí přenos jednoho řádku.

Přenos celého snímku vypadá podobně – nejprve je posláno zatemnění, pak vertikální synchronizační puls (v podobě dlouhých synchronizačních pulsů), pak přijde opět zatemnění a následují obrazová data.

U analogových televizí s vakuovými obrazovkami a vychylovacími cívkami zabralo přesunutí paprsku z jednoho konce obrazovky zpátky na začátek nějaký čas. Během tohoto přesunu muselo být takzvané *elektronové dělo* vypnuté, aby nezanechávalo na stínítku stopu, proto se posílala černá barva, respektive úroveň ještě o kousek menší, než má černá. Dnes u plochých obrazovek LCD,

LED a podobných už žádný paprsek ani vychylovací cívky nejsou, ale přesto je dobré na termín „návrat paprsku“ nezapomenout.

Ve skutečnosti se to celé zobrazí lehce posunuté – front porch je vlastně ještě „konec předchozího řádku“ (na obrazovce vpravo), synchronizační puls je na začátku řádku a back porch vidíte vlevo.

Tohle všechno proběhne padesátkrát za sekundu v Evropě, kde máme normu PAL; ve Spojených státech a dalších zemích, kde se používala norma NTSC, byla obnovovací frekvence 60 Hz (u černobílého vysílání, později u barevného byla frekvence změněna na 59,94 Hz). Aby to nebylo tak jednoduché, tak se při televizním vysílání používalo takzvané prokládání (interlacing) – skutečná frekvence obrázků byla 25 za sekundu s tím, že se poslal jeden půlsnímek, a druhý byl o půl řádku posunutý. Tím se zdvojnásobilo vertikální rozlišení na 625 řádků, ovšem přinášelo to problémy, lehký třas obrazu, rozmazání hran atd. Při sledování budovatelské veselohry ze života družstevníků to moc nevadilo, ale u připojení k počítači je to velký problém, protože roztřesená a rozpitá písmenka unavují oči.

U domácích počítačů se proto prokládání nepoužívalo a posílaly se půlsnímky bez posunutí, takže maximální rozlišení bylo na výšku 312 řádků. Ovšem nepoužívaly se všechny, některé jsou určené pro synchronizaci a na začátku a na konci se nechávalo několik řádků volných, aby nebyl obraz až úplně na kraji obrazovky. (U počítačů pro americký a japonský trh to bylo samozřejmě jinak, používalo se 525 řádků prokládaných s obrazovou obnovovací frekvencí 30 Hz.)

Samotný videosignál je jednoduchý analogový signál v rozmezí 0 V až 1,073 V. Synchronizační puls má 0 voltů, černé barvě odpovídá 0,339 V, bílá je 1 V.

Některá videorozhraní, například VGA, oddělují synchronizační pulsy do samostatného signálového vedení. Je to kvůli kvalitnějšímu obrazu

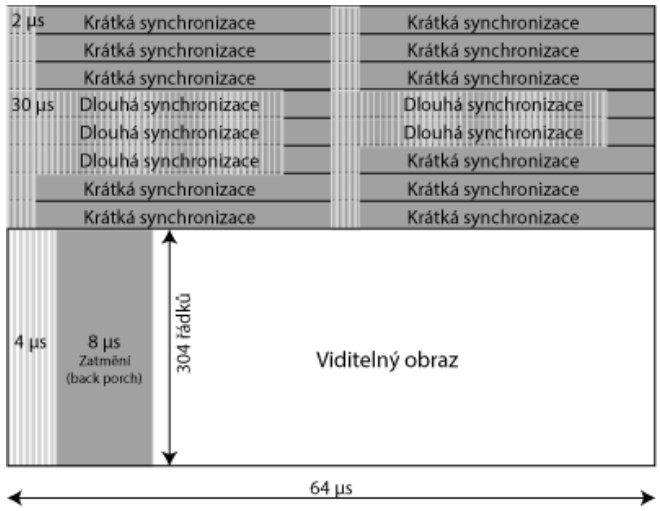
Nejdůležitější parametr je čas. Televize, zejména ty staré, jsou docela robustní zařízení, zvyklé poradit si s lecčíms, takže vše, co plus mínus vyhoví normě, nějak zobrazí. Paradoxně může být problém s novými televizemi – nadšenci do starých počítačů vědí, že některé staré počítače, které fungují se starými televizemi bez problémů, po připojení k novým LCD televizím nefungují a na obrazovce je vidět jen NO SIGNAL – to proto, že generují video mimo toleranci a moderní obvody si s tím už neporadí, respektive nechtějí.

V dalším textu budu primárně používat časování pro evropskou normu PAL. NTSC má princip obdobný, jen časy jsou jiné.

Každý televizní řádek trvá 64 μs. Úvodní zatmění (front porch) má mít 1,65 μs, synchronizační puls 4,7 μs, back porch 5,7 μs a na samotné aktivní video zbývá 51,95 μs. Norma předpokládá nějakou toleranci, televize obvykle jdou ještě dál a v praxi se ukazuje, že je dostatečné posílat 4 μs synchronizaci, 8 μs back porch a 52 μs video signál.

64 μs na řádek odpovídá řádkové frekvenci 15625 Hz. Když tuto frekvenci podělíme počtem řádků (312,5), získáme 50 Hz obrazové frekvence – takže to hezky vychází…

Vertikální synchronizace jsou vlastně jen řádky bez obrazové informace, pouze se střídá zatemnění a horizontální synchronizace. Nejprve jde 6 půlřádků (32 μs) s krátkou synchronizací – 2 μs synchronizační puls, 30 μs černá, 2 μs synchronizační puls, 30 μs černá. Pak jde pět půlřádků s dlouhou synchronizací – 30 μs synchronizační puls, 2 μs černá. Pak pět půlřádků s krátkou synchronizací. U dlouhých půlřádků by měl být paprsek „vlevo nahoře".



Obrázek 46: časování signálu PAL

Při generování videosignálu v počítači se zjednodušuje, nerozlišují se liché a sudé půlsnímky a generuje se 312 řádků.

Když budeme předpokládat displej s 240 řádky, musíme vše nastavit tak, aby šel vertikální synchronizační puls (5 dlouhých + 5 krátkých půlřádků), pak X černých řádků, pak 240 řádků videa, Y černých řádků a po nich 6 půlřádků synchronizace. Na synchronizaci tedy připadá úhrnem 8 řádků, zbývá 304 řádků pro užitečná data. 240 z toho jsou video, zbývá 64 černých řádků (respektive řádků okraje, border). Součet X a Y proto musí být roven 64, aby byl celkový počet 312. Obvykle

se nastavuje X=Y, tedy stejný počet před obrázkem a za obrázkem, a posunutím těchto konstant můžeme posunout obraz na stínítku výš či níž.

Počet bodů na řádku je dán pouze tím, jak rychle jsme schopni změnit signál během 52 μs. Tedy vlastně jakou maximální frekvenci dokážeme vygenerovat. Když dokážeme změnit signál každé 2 μs, získáme horizontální rozlišení 26 bodů. Což není mnoho. S frekvencí 1 MHz, tedy změnou každých 0,5 μs, získáme teoretické rozlišení 104 bodů. Když přitlačíme na 0,2 μs, což je 2,5 MHz, získáme 260 bodů horizontálně…

A teď počítejme: OMEN Alpha běží na frekvenci 1,8432 MHz. Nejkratší instrukce NOP trvá něco málo přes 2 μs. To nedáme ani náhodou! Jak to tedy dělaly ty počítače dřív?

No, neměly to lehké. Video musel zajišťovat samostatný obvod, který běžel na vyšší frekvenci a staral se o všechny ty synchronizace, o načítání dat z paměti a jejich posílání na video výstup. U těch lepších strojů na to byly samostatné procesory a koprocesory s vlastní pamětí, československé počítače té doby používaly divoké zapojení čítačů, multiplexorů a posuvných registrů, a některé stroje, namátkou ZX-80 a ZX-81, používaly ke generování videosignálu speciální zapojení, v němž hrál svou roli i procesor, který tak v době, kdy se zobrazoval obraz, nedělal nic, a veškeré výpočty a běh programů musel stihnout v době, kdy se nezobrazovalo nic a „vracel se paprsek". Tedy během těch osmi synchronizačních řádků. Což není moc, to je nějakých 2,5 % času.

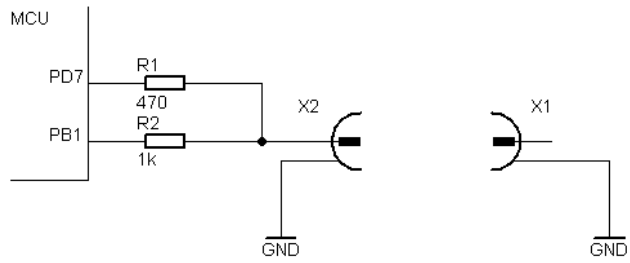
Ani takové ZX Spectrum na tom nebylo o moc lépe. Sice mělo vyspělejší obvod ULA, který dokázal generovat i barevný obraz, a mělo dvě sady pamětí RAM (16 kB + 32 kB), ale pokud běžel program někde na adresách 4000h – 7FFFh, přetahoval se procesor o přístup do pamětí s obvodem ULA, a ta měla prioritu, takže procesor byl na krátké okamžiky pozastavován a program tak běžel o něco pomaleji.

http://digitarworld.uw.hu/z80_video.png

13.2 Arduino a video

Proč Arduino? Protože je o něco rychlejší – pracuje na 16 MHz, ale procesory AVR zvládnou i 20 MHz. Navíc mají i zabudovanou periferii pro řízení sériové sběrnice SPI, z níž lze využít „serializátor" – tu část, která převádí paralelní data na sériovou posloupnost bitů.

K zapojení budeme potřebovat dva výstupní piny. Jeden pro synchronizaci, druhý pro barvu. Nějak takto:



Obrázek 47: nejjednodušší videovýstup

Na výstupu nebude „televizní signál“ jako z antény, ale videosignál. Zobrazí vám ho televizor nebo monitor, který umí pracovat s takzvaným „composite video“ signálem.

Hodnoty rezistorů jsou zvoleny tak, aby odpovídaly standardní řadě hodnot a výsledek se co nejvíc blížil potřebným hodnotám. Na videovstup v televizi můžeme pohlížet jako na další odpor 75 Ω a protože známe i výstupní napětí na pinech mikrokontroléru (5 V), můžeme spočítat úroveň napětí pomocí Ohmova zákona.

```
Isync= Uin / (Rsync+Rout)  (5 / 1075 = 4,6511 mA)
Ivideo= Uin / (Rvideo+Rout) (5 / 545 = 9,1745 mA)
Usync= Isync* Rout          ( = 0.34V)
Uvideo= Ivideo* Rout       ( = 0.688V)
Uvideo+sync= (Ivideo+Isync) * Rout  ( = 1.03V)
```

Vidíme tedy, že jsme získali sadu napětí, pomocí které můžeme pohodlně vytvořit černobílý obraz (či černo-šedo-bílý).

VIDEO	SYNC	V	Význam
0	0	0	Synchronizační puls
0	1	0.34	Černá
1	0	0.68	Šedá
1	1	1.03	Bílá

Po dobu generování HSYNC stačí stáhnout piny VIDEO a SYNC k log. 0. Pak nastavíme SYNC na log. 1 a získáme na výstupu „téměř černou“. Změnou hodnoty na výstupu VIDEO pak přepínáme černou a bílou.

Zapojení a knihovny naleznete na Arduino Playground pod názvem TVOut:

https://playground.arduino.cc/Main/TVout

Další odkazy ke studiu a vlastním experimentům jsou zde:

- NTSC displej z AVR: https://hackaday.com/2013/03/27/color-ntsc-video-directly-from-an-avr-chip/
- ATTiny s výstupem na VGA + zvuk: http://www.avrfreaks.net/forum/quark-85-demo-kube-184-x-240-vga-8-colors-and-sound-tiny85
- https://sites.google.com/site/retroelec/hardware

Pro vaše vlastní konstrukce si vám dovolím navrhnout zapojení, které publikoval Grant Searle na svém webu.

http://searle.hostei.com/grant/MonitorKeyboard/index.html

Využívá dvojice mikrokontrolérů ATmega 88 (168, 328). Jeden obsluhuje klávesnici standardu PS/2, druhý generuje monochromatický obraz včetně jednoduché semigrafiky. K vlastnímu počítači se připojují přes sériové rozhraní s rychlostí 115200 Bd. Výstup připojíte k televizoru, jako ve zlaté éře domácích počítačů.

13.3 OMEN Echo (koncepce)

Velmi zajímavou konstrukci s názvem DAN64 navrhl Juan J. Martínez:

https://www.usebox.net/jjm/dan64/

Jeho konstrukce využívá procesoru AVR a sériové paměti RAM typu 23LC512 (šlo by použít i větší 23LC1024). Tyto paměti vyrábí Microchip a nabízejí 64 kB, popř. 128 kB RAM s připojením SPI / DualSPI / QuadSPI.

Autor velmi důvtipně využil faktu, že instrukce čtení z paměti dokáže posílat obsah paměťových buněk po sobě, aniž by bylo potřeba opakovaně zadávat adresu, z níž se má číst. Na začátku televizního mikrořádku proto nastaví „čtení z paměti od zadané adresy“, pak přepojí sériový výstup z paměti (MISO) na televizní výstup a posílá pouze hodinové pulsy. Paměť automaticky posílá bit po bitu obsah paměti a po každé osmici bitů přejde na další buňku… Na konci mikrořádku zase procesor připojí výstup paměti ke svému rozhraní a může pokračovat v práci.

Uvnitř AVR běží časovač, který se stará o generování synchronizačních pulsů a výše uvedený mechanismus generování obrazu. Když má procesor „volno“, běží emulátor procesoru 6502 a může vykonávat program.

Já jsem konstrukci trochu upravil. Odstranil jsem nahrávání programů pomocí audiovstupu a použil jsem větší procesor AVR, takový, kde je možné využít volné piny i jako paralelní port.

Z počítače DAN64 jsem zachoval vstup z klávesnice PS/2. Výhoda je, že nemusíte stavět vlastní, ale použijete standardní klávesnici k PC – ještě stále se dají sehnat i nové.

Klávesnice PS/2
Klávesnice tohoto typu se připojují jednoduchým konektorem DIN, kde kromě napájení (+5 V a zem) jsou dva signálové vodiče: hodiny a data. Klávesnice komunikuje vlastně standardním synchronním sériovým protokolem (osm bitů + paritní bit), ale přináší jednu záludnost: hodinový takt pro přenos generuje sama klávesnice. Procesor je tedy v roli toho, kdo musí zareagovat na signál, nemůže si přenos řídit sám. Používá se proto takové zapojení, kde hodinový signál vyvolává přerušení a v obsluze tohoto přerušení se načítají jednotlivé bity.

Klávesnice PS/2 většinou vysílá informace o tom, že byla nějaká klávesa stisknuta a puštěna. Jsou ale situace, kdy je naopak ona v roli příjemce dat – typicky když počítač potřebuje změnit stav signalizačních LED nebo když je potřeba klávesnici resetovat. V takovém případě vyzve procesor klávesnici k příjmu tím, že stáhne hodinový vstup k logické nule alespoň na 0,1 ms, pak uvede datový vodič do stavu 0 a uvolní hodiny. Na tuto výzvu klávesnice zareaguje, začne generovat hodinové pulsy a procesor může posílat data.

13.4 Barevný videosignál

Zatímco černobílý videosignál snadno vytvoříte pomocí jednočipu, u barevného signálu je to velmi obtížné. Barevná informace se totiž přenáší pomocí vysokofrekvenčního signálu, který je přidán k výše popsanému jasovému signálu. V systému PAL je „barvonosná“ frekvence rovna 4,43361875 MHz a kóduje se nikoli jako RGB složky, ale jako rozdílové signály modré a červené složky vůči jasu (U, V).

Ke generování barevného videosignálu se proto používají specializované obvody, zvané RGB enkodéry, například AD724 nebo MC1377.

S dostatečně výkonným procesorem je možné samozřejmě generování barevného signálu zvládnout, ale je to velmi náročné na přesné časování. Rickard Gunée postavil videohru, která generuje barevný videosignál, s mikrokontrolérem SX taktovaným na 50 MHz.

http://www.rickard.gunee.com/projects/video/sx/howto.php

Některé televizory s konektorem SCART dokázaly zpracovat přímo RGB signály, ale podle mých testů je velká náhoda na takový televizor dnes narazit.

Mnohem jednodušší, než generovat barevný signál PAL / NTSC, je generování obrazu pro monitory VGA. Toto rozhraní používá dva oddělené synchronizační signály a tři analogové vstupy pro jednotlivé barevné složky R, G a B. Problém je, že VGA vyžaduje mnohem vyšší frekvence signálů než PAL, a i když je možné takový signál vygenerovat mikrokontrolérem, už nezbývá moc času na další činnosti.

Níže naleznete seznam zajímavých konstrukcí, které umožňují pracovat s VGA – některé z nich fungují jako sériové terminály, takže není problém je připojit k našim konstrukcím. Zajímavá konstrukce je „Octapentaveega", v níž autor použil tři jednočipy ATTiny85. Každý z nich generuje jednu barevnou složku a všechny dohromady fungují jako sériovým portem řízený barevný terminál.

- Displej k počítači z RasPi: https://hackaday.io/project/9567-5-graphics-card-for-homebrew-z80
- VGA terminál se STM32 (ChibiTerm): https://hw-by-design.blogspot.com/2018/07/low-cost-vga-terminal-module-project.html
- Arduino a VGA: https://github.com/smaffer/vgax
- Arduino VGAOut: https://code.google.com/archive/p/arduino-vgaout/downloads
- ATMega VGA/PAL: http://tinyvga.com/avr-vga
- AVR VGA demo: https://github.com/leonsodhi/avr-vga-demo
- VGA s třemi ATtiny: https://github.com/rakettitiede/octapentaveega
- http://www.instructables.com/id/Arduino-Basic-PC-With-TV-Output/

14 OMEN Kilo

OMEN Kilo vám přináší na pracovní stůl výkonný osmibitový mikroprocesor 6809 v zapojení, které je jednoduché, a přitom snadno rozšiřitelné.

Do sestavy můžete zapojit až pět přídavných desek a stvořit tak velmi výkonný poloprofesionální počítač.

OMEN Kilo

Poloprofesionální počítač

- Výkonný procesor Motorola 6809
- 128 kB RAM (rozšiřitelných)
- Až 64 kB EEPROM
- Možnost připojení přídavných modulů v konstrukci typu "backplane"

14 OMEN Kilo

14.1 Architektura procesoru Motorola 6809

Vraťme se zase na chvíli do historie. Motorola po svém prvním komerčně úspěšném procesoru 6800 samozřejmě neusnula na vavřínech a připravila několik variant téhož procesoru (s integrovanou pamětí, s omezenější instrukční sadou atd.) Zároveň se ale připravovala i na věci příští, a tak vyvíjela vhodného nástupce. Bylo jasné, že musí vyvinout šestnáctibitový procesor, ale zároveň nechtěla opustit osmibitový trh, protože pro osmibity bylo ještě mnoho let uplatnění. Proto začaly vznikat v laboratořích Motoroly dva procesory naráz. Jedním z nich byl 68000 – šestnáctibitový procesor, který byl ale navržen jako plně 32bitový s 16bitovou sběrnicí (model 68008 dokonce s osmibitovou – v té době znamenalo rozšíření datové sběrnice na dvojnásobek neúnosné zdražení celého systému). A druhým byl právě 6809.

V časopise Byte vyšel svého času článek od dvou návrhářů 6809, takže docela dobře víme, proč byl procesor navržen právě tak, jak byl navržen. Dnes je to cenný historický materiál, který popisuje tehdejší stav vývoje a mikroelektroniky. Pojďme se podívat na rozhodnutí, která návrháři udělali.

V první řadě bylo jasné, že nelze zahodit kompatibilitu s existujícím software pro 6800. Nakonec padlo rozhodnutí obětovat binární kompatibilitu (jako má třeba Z80 zpětně s 8080), ale zachovat kompatibilitu na úrovni zdrojového kódu. Tedy tak, aby programy v assembleru 6800 šly přeložit pro nový procesor ve funkčně ekvivalentní podobě. Instrukce se tedy jmenovaly stejně, i když se překládaly na jiné operační kódy.

Další rozhodnutí mělo za úkol snížení nákladů. Proto bylo rozhodnuto, že návrháři recyklují části, které se osvědčily v 6800 – například kombinační logiku pro dekódování instrukcí (6809 nepoužíval mikrokód).

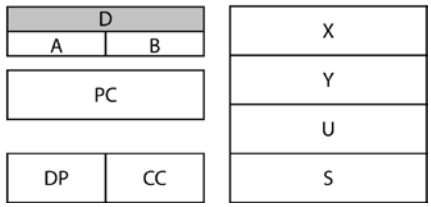
Vývojáři také podrobili existující zdrojové kódy pro procesor 6800 statické i dynamické analýze, aby zjistili, co vlastně programátoři používají a jak.

Pro zajímavost – statická analýza, která nezohledňuje např. průběhy smyčkou a pouze počítá výskyty instrukcí, ukázala, že instrukcí LOAD (tedy přesuny z paměti do registrů) je v kódech 23,4 %, STORE (opačným směrem) 15,3 %, volání podprogramů a návratů je 13 procent, 11 procent tvoří podmíněné skoky atd., až ke třem procentům instrukcí sčítání a odčítání. Bylo tedy zřejmé, co je třeba posílit, jaké instrukce jsou často používané (a mají tedy zůstat jednobajtové) a jaké je možné přesunout mezi instrukce s prefixem…

Další analýza se zaměřila na instrukce, které adresují pomocí indexu. Nejčastěji (53 % případů) byl index v rozmezí 1 – 31, ve 40 % případů byl index roven nule, 6 % případů připadalo na index v rozmezí 64 – 255, no a poslední procento tvořily indexy v rozsahu 32 – 63. Tohle zjištění vedlo například k výraznému rozšíření adresních módů.

Vzhledem k tomu, jak časté byly instrukce INC a DEC ve spojení s adresováním, padlo rozhodnutí přidat adresní mód, který dokáže automaticky zvyšovat a snižovat hodnotu v indexovém registru.

No a v neposlední řadě se rozrostl počet registrů: K šestnáctibitovému indexovému registru X přibyl druhý šestnáctibitový registr Y, k ukazateli zásobníku S přibyl ukazatel uživatelského zásobníku U, návrháři přidali osmibitový registr DP (direct page) a přidali možnost spojit registry A a B do jednoho šestnáctibitového registru D.



Obrázek 48: registry 6809

Registr DP souvisí s adresováním. Procesory Motorola mají totiž možnost odkazovat buď plnou adresou (2 bajty), nebo zkrácenou, kdy adresu tvořil jen jeden bajt, ten nižší, a vyšší je vždy 0. S tímto přístupem jsme se setkali i u procesoru 6502. U procesoru 6809 existuje taky zkrácené adresování, ale vyšší bajt není automaticky 0, ale je roven právě obsahu registru DP. Programátor tedy může „nultou stránku" posunout kamkoli v paměti (resp. nastavit vyšší bajt adresy). Místo „Zero page" se takové adresování nazývá „direct page".

Registr CC má stejnou funkci jako u Z80 registr F – je to registr příznaků (Condition Code). Jeho bity jsou (od nejvyššího):

Bit	Název	Funkce
7	E	Entire – při obsluze přerušení říká, zda je na zásobníku kompletní sada registrů, nebo jen PC a CC
6	F	FIRQ mask: když je 1, je zakázáno přerušení FIRQ
5	H	Half carry – přenos mezi horním a dolním půlbyte během sčítání
4	I	IRQ mask: když je 1, je zakázáno přerušení IRQ
3	N	Negative – záporný výsledek
2	Z	Zero – nulový výsledek
1	V	Overflow – přetečení
0	C	Carry – přenos

Adresní módy a indexy

Adresní módy zůstaly stejné jako u procesoru 6800 (a hodně podobné těm z 6502). Výrazně byly rozšířeny možnosti indexace. Instrukce, které používají indexovanou adresaci, obsahují (minimálně) o jeden bajt navíc. V tomto bajtu, zvaném „post byte", je uložena informace o tom, jak se indexuje a s jakými registry.

Obecně lze pro indexování použít indexové registry X a Y, ukazatele zásobníku S a U, a s určitými omezeními i ukazatel PC. Co máme tedy na výběr?

- **Indexový registr (X, Y, S, U) + konstantní offset.** Zde se rozlišuje několik možností podle toho, jak velký je offset. Pokud je 0, je to označeno v post byte a není zapotřebí další paměťová buňka. Druhá možnost je, že offset je v rozmezí -16 až +15. V takovém případě je offset zapsán rovněž v post byte. Zbývající dvě možnosti jsou „osmibitový index" a „šestnáctibitový index".
- **K indexovému registru (X, Y, S, U) je přičtena hodnota registru A, B nebo D** (D je šestnáctibitový virtuální registr, vzniklý spojením registrů A a B).
- Adresa je vzata přímo z indexového registru (X, Y, S, U) a poté je jeho hodnota zvýšena o 1 nebo 2. (**Postinkrement**)
- Hodnota v indexovém registru (X, Y, S, U) je snížena o 1 nebo 2 a pak je vzata k adresaci. (**Predekrement**)
- Adresa je dána **programovým čítačem PC, k němuž je přičten osmibitový nebo šestnáctibitový offset**.

Slušná sestava, že? A vynásobte si ji dvěma, protože všechny tyhle adresní módy (až na dvě výjimky) lze použít jako nepřímé (indirect), tj. adresa, kterou získáme indexací, ještě není ta cílová, ale ukazuje do paměti, ze které se přečtou dva bajty, a teprve ty udávají efektivní adresu, se kterou se bude pracovat.

Představme si, že od adresy 0800h máme uložené adresy nějakých dat ke zpracování a chceme je všechny projít. Není třeba žádných čachrů s mnoha registry, stačí třeba do X uložit adresu té tabulky adres (0800h) a pak načítat data přímo z cílových lokací pomocí LDA [X++] (hranaté závorky značí nepřímou adresaci, X++ je postinkrement o 2.)

Zmíněné dvě výjimky u nepřímé adresace jsou právě postinkrement o 1 a predekrement o 1. Pokud používáte nepřímou adresaci, nelze ji zkombinovat s posunem o 1, vždy o 2!

Relokace

Výrazné designérské rozhodnutí bylo posílení podpory relokace. Pokud jste pracovali s osmibitovými procesory jako třeba 8080, byla adresa, na jakou má být program nahrán, zadána při překladu. Pokud se pak kód ocitl na jiných adresách, vedly skoky nazdařbůh a havárie nastala v řádu milisekund. Z80 nabídl relativní skok, ale ten není všespásný: jen 128 bajtů dozadu a dopředu, navíc

volání podprogramu je opět s absolutní adresou. Řešení bylo dvojí – buď byly programy napevno od nějaké adresy (třeba u CP/M to bylo 0100h), nebo byla na začátku rutina, která všechny absolutní adresy v kódu vlastního programu změnila podle toho, kde byl program nahraný.

Návrháři 6809 posílili „relokovatelnost" programů významnou měrou: všechny relativní skoky mohou mít osmibitový i šestnáctibitový offset, včetně instrukce volání podprogramu (JSR). Mohou využívat i indexované adresace, jak jsem psal výše.

Dva zásobníky

Možná vás napadne jazyk FORTH, pro který je tento koncept jako stvořený. Ve skutečnosti návrháři měli na mysli jinou situaci: představte si, že podprogram potřebuje vrátit víc hodnot. Pokud je uloží někam do paměti na pevně danou adresu, nebude reentrantní (tzn. nelze daný podprogram vyvolat znovu v rámci provádění sebe sama). Pokud je předá na zásobníku, je potřeba nejprve vytáhnout návratovou adresu, tu si někam uložit (kam ale?), uložit data a pak vrátit návratovou adresu. Právě tenhle problém vývojáři obešli druhým zásobníkem U. Podprogram prostě PUSHuje data, která chce předat, do druhého zásobníku.

Instrukční sada 6809 obsahuje dvě instrukce pro ukládání na zásobník – PSHS a PSHU. První ukládá na standardní zásobník, druhá na uživatelský. Ale co ukládá? Samozřejmě obsah registrů. Zde návrháři použili další trik: za operačním kódem PSHx je další bajt, který říká, které registry se mají na zásobník uložit. Jednou instrukcí tak můžeme uložit kompletní sadu registrů. Což je poměrně elegantní řešení, protože se instrukce pro ukládání obsahu registrů většinou vyskytují v sériích, navíc nemuseli pro každou kombinaci registrů a zásobníků dělat vlastní operační kód.

Výtky

Ve výše zmíněném článku z Byte jsou sepsány i výhrady čtenářů a uživatelů, na které konstruktéři odpovídají. Například: *Proč nejsou blokové operace?* Nejsou. Něco bylo třeba obětovat, navíc podle vývojářů je programové řešení blokových operací díky mocným možnostem indexace poměrně jednoduché.

Nebo: *Proč nejsou bitové operace?* Odpověď: prodražilo by to celý procesor, tak jsme je, vzhledem k tomu, že nejsou často používané, oželeli. *Proč je násobení, ale není dělení?* Protože násobení je mnohem častější (třeba při přístupu k dvojrozměrným polím).

Potěšila mě devátá výtka: „*Proč jste přidali všechny ty adresní módy? Já je nikdy nepoužiju!*" Jako bych si četl diskuse na některých programátorských webech…

Na otázku *proč nemá registr DP vlastní instrukce LOAD a STORE?* odpověděli tvůrci šalamounsky: jeho obsah je natolik důležitý, že by si programátor měl být vědom toho, že mění něco podstatného, tak to není úplně přímočaré.

Spousta drobností navrch

Procesor 6809 má třeba dvojici instrukcí TFR / EXG. TFR (transfer) přesouvá obsah z jednoho registru do druhého, EXG vzájemně vymění obsah dvou registrů. Platí to jak pro osmibitové (A, B, DP a CC – příznakový registr), tak pro šestnáctibitové (X, Y, S, U, SP a D).

Přerušení (hardwarové – IRQ i softwarové – instrukce SWI, SWI2 a SWI3) uloží automaticky na zásobník všechny registry. Instrukce návratu z přerušení pak zase všechny obnoví. Procesor má ale i vstup FIRQ (Fast IRQ), který vyvolá přerušení, ale registry neukládá.

Podmíněné skoky testují nejen čtyři základní příznakové bity, ale i jejich možné kombinace, takže můžete např. vyvolat skok, pokud je první hodnota větší nebo rovna druhé hodnotě v bezznaménkové aritmetice (BHS – Higher or Same), ale můžete kontrolovat totéž pro čísla se znaménkem (BGE – Greater or Equal).

Většina instrukcí je velmi rychlá, zabírá jen pár taktů. Na rozdíl od procesorů z rodiny 8080/Z80, které třeba pro načtení operačního kódu a jeho dekódování zaberou čtyři takty, u 6809 je vše mnohem rychlejší: NOP dva takty, podmíněný relativní skok 3 takty, aritmetické či logické operace 6 taktů, násobení 11 taktů (6809 totiž obsahuje hardwarovou násobičku 8×8 bitů). Samozřejmě pokud je potřeba pracovat s indexovaným adresním módem a načítat post byte, prefix či offset, musíte si patřičný počet taktů přičíst…

Kam s ním?

6809 přišel na trh na konci 70. let, čtyři roky po Z80. Mohly tyhle čtyři roky znamenat, že 6809 už nijak výrazně do osmibitové éry nepromluvil? Dost možná ano. I když všechna ta Spectra a Atari přišla až začátkem 80. let, měl 6809 smůlu. Pravděpodobným důvodem byla cena, a pak další náklady na přepsání software nebo učení nové instrukční sady.

A tak vlastně jediný „známější“ počítač, který používal 6809, byl TRS-80 CoCo, neboli „Tandy/RadioShack Color Computer“, který přišel na trh v roce 1980. (Číslice 80 v označení TRS-80 odkazuje nikoli na rok výroby, ale na předchozí model tohoto počítače, který používal Z80.) Postupně vznikly tři verze, v Británii se vyráběly podobné počítače pod označením Dragon 32 / Dragon 64. Z dalších osmibitů používaly 6809 francouzské počítače Thomson MO5 / TO7 / TO8, japonské FM7 a FM8 od Fujitsu, nebo počítač Aamber Pegasus (1981), který svým minimalistickým návrhem připomínal trochu ZX80/ZX81.

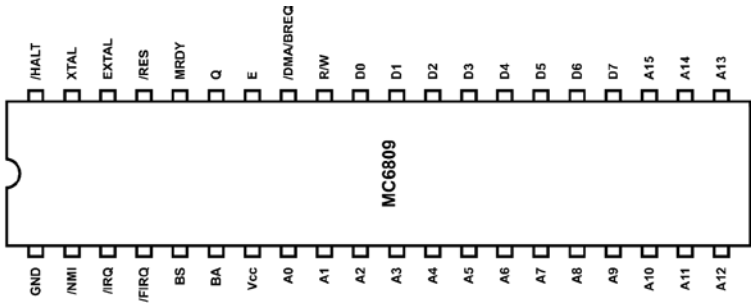
Všimněte si, že počítače s 6809 byly na trhu dřív než třeba Sinclairovy geniální stroje se Z80. Na chvíli se zasněte a představte si, jak by vypadalo Spectrum s 6809…

Pro 6809 vznikl i zajímavý software. Kromě FORTHu nebo operačního systému OS-9 (později upraven jako NitrOS-9) to byl třeba i Microsoft BASIC – ten ještě v době, kdy Bill Gates

psal sám programy. Gates později v rozhovoru označil 6809 za nejlepší osmibitový procesor vůbec.

Škoda jen, že přišel (asi) pozdě. Sice bylo možné pomocí transpileru přeložit zdrojové programy 6809 pro mikroprocesor 68000, ale to už byla spíš jen labutí píseň, stejně jako procesor Hitachi HD6309, o kterém rozhodně stojí za to se zmínit. Příslušnou kapitolu najdete v příloze.

14.2 Zapojení MC6809



Možná nepřekvapí jistá podobnost s procesorem 6502. Je to logické, protože, jak jsme si už několikrát řekli, 6502 i 6809 vycházejí ideově z téhož předchůdce, 6800.

Popis vývodů

Procesor má vyvedené obě hlavní sběrnice, jak datovou (D0 – D7), tak adresní (A0 – A15). Tady nás nečeká žádné překvapení. Signály nejsou multiplexované a jsou plně k dispozici.

Vstup /RESET slouží k inicializaci procesoru do výchozího stavu. Na vstupu je Schmittův klopný obvod, takže stačí jednoduché zapojení s kondenzátorem a rezistorem, podobně jako u předchozích počítačů. Po RESETu se načte adresa (2 bajty) z adres FFFEh a FFFFh a získaná hodnota se použije k nastavení programového čítače PC. *Nezapomeňte, že 6809 používá ukládání ve stylu Big Endian, tedy nejprve vyšší část adresy, poté nižší. Vyšší část tedy bude na adrese FFFEh, nižší na FFFFh.*

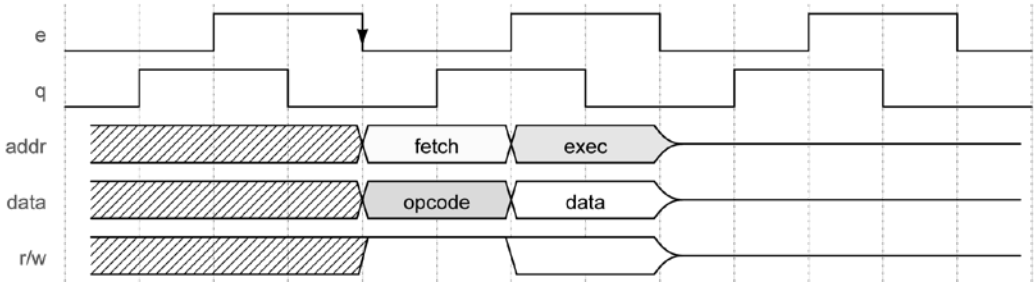
Výstup R/W informuje okolí o tom, jestli procesor hodlá číst (=1), nebo zapisovat (=0). Opět, podobně jako u 6502, se nerozlišují periferie a paměť, pro procesor je vše „paměť".

Pro připojení krystalu se používají vývody XTAL a EXTAL. Pokud použijete externí generátor časovacích pulsů, přiveďte signál na vstup EXTAL a XTAL nechte nezapojený.

Uvnitř poběží procesor na frekvenci, která je rovna čtvrtině frekvence krystalu (popřípadě přivedené frekvenci). U OMEN Kilo jsem tedy použil frekvenci 7,3728 MHz, která po podělení čtyřmi dává naši oblíbenou pracovní frekvenci 1,8432 MHz.

Existují tři varianty tohoto procesoru, které se liší podle maximální pracovní frekvence: 6809, 68A09 a 68B09. Varianta bez písmene pracuje na frekvenci 1 MHz, varianta A používá 1,5 MHz a varianta B může pracovat až s frekvencí 2 MHz (to znamená, že externí krystal může mít frekvenci 4, 6, resp. 8 MHz).

Pro řízení vnějších obvodů slouží hodinové signály E a Q. E má stejnou funkci, jakou má u 6502 výstup PHI2. Sestupnou hranou na výstupu E oznamuje procesor, že začíná operační cyklus. Poté přijde vzestupná hrana na výstupu Q a oznámí, že na adresní sběrnici je platná adresa. Následuje vzestupná hrana signálu E, sestupná signálu Q (bez speciálního významu) a celý cyklus ukončuje (a zároveň zahajuje nový) sestupná hrana signálu E, při níž přečte procesor data ze sběrnice.



Obrázek 49: časování 6809

Pro většinu jednoduchých případů můžeme signál Q (Quadrature time) ignorovat a pracovat pouze se signálem E, stejně jako u 6502.

Pozor na variantu 6809E! Tato varianta nemá interní oscilátor, vyžaduje externí generátor dvoufázových hodin a zapojení vývodů je lehce odlišné.

Výstupy BA a BS informují o tom, zda je procesor odpojený od sběrnice (ve stavu HALT / DMA) či zda reaguje na požadavek na přerušení apod. Za normálního chodu jsou oba v 0 a pokud nepotřebujete nějak speciálně reagovat na výjimečné stavy, můžete oba ignorovat.

Vstup MRDY slouží k témuž, k čemu sloužily vstupy READY, /WAIT či RDY – informuje o tom, že externí obvody nezpracovaly požadavek a že potřebují, aby procesor počkal. Pokud je tento vstup v log. 0, hodiny se zastaví ve stavu E=1, Q=0 a čeká se, až bude MRDY opět 1. Teprve pak procesor dokončí čtení dat, popřípadě přestane posílat data k zápisu, a pokračuje se dál.

Vstup /DMA/BREQ slouží k přerušení práce procesoru a jeho odpojení od sběrnice. Pokud je na tomto vstupu na konci aktuálního cyklu 0, procesor se uvede do stavu DMA a uvolní sběrnici. Uvolnění sběrnice signalizuje nastavením BS = BA = 1. Okolní zařízení teď má až 15 cyklů na to, aby si udělalo, co je potřeba. Po 15 cyklech si procesor vezme dva cykly pro vnitřní refresh a na tu dobu nastaví BS = BA = 0. Pokud stále trvá požadavek na DMA, celý postup se opakuje.

Vstup /HALT zastaví procesor podobně jako předchozí vstup. Do stavu HALT se procesor přepne poté, co dokončí aktuálně prováděnou instrukci. Opět se odpojí od datové i adresní sběrnice, odpojí i výstup R/W, a nastaví BS = BA = 1.

Přerušení
Poslední tři vstupy, o nichž jsem se ještě nezmínil, jsou vstupy /NMI, /FIRQ a /IRQ. Jejich stav je vyhodnocován ve chvíli sestupné hrany Q (E=1).

/NMI je dobře známé „nemaskovatelné přerušení". Logická 0 na tomto vstupu vyvolá přerušení, které programátor nemůže programově zamaskovat. Jediná výjimka je těsně po RESETu, dokud není nastaven obsah v registru ukazatele zásobníku (S). Po detekování požadavku na NMI je na zásobník automaticky uložen obsah registrů PC, U, Y, X, DP, A, B a CC a procesor si z adres FFFCh a FFFDh načte adresu obsluhy nemaskovatelného přerušení. Toto přerušení má zároveň nejvyšší prioritu.

Téměř identicky se chová požadavek /IRQ. Dva rozdíly tu ale jsou. Zaprvé: adresa obsluhy přerušení se bere z adres FFF8h a FFF9h. A zadruhé: pokud je bit I v registru CC roven 1, přerušení se nevykoná.

Vstup /FIRQ (Fast IRQ) funguje obdobně jako IRQ. I tento vstup vyvolá přerušení (adresa obsluhy je v paměti na FFF6h a FFF7h), i toto přerušení lze zamaskovat bitem F v registru CC, ale hlavní rozdíl je, že neukládá na zásobník žádné jiné registry, jen programový čítač PC a registr příznaků CC.

14.3 6809E

Zmínil jsem se o této variantě výš, tak jen pro představu, jak odlišný je to procesor. Sám jsem se s ním setkal, když se mi dostala do rukou várka procesorů, označená jako 68B09, ale tvářila se jako nefunkční. Když jsem je z čiré zvědavosti zkusil zapojit jako 6809E, ukázalo se, že jsou funkční, jen špatně označené!

Z hlediska programátora jsou oba typy totožné. Z hlediska hardware je šest pinů, jejichž funkce je rozdílná.

Pin	6809	6809E
33	/DMA (vstup)	Busy (výstup)
34	E (výstup)	E (vstup)
35	Q (výstup)	Q (vstup)
36	MRDY (vstup)	AVMA (výstup)
38	EXTAL	LIC (výstup)
39	XTAL	TSC (vstup)

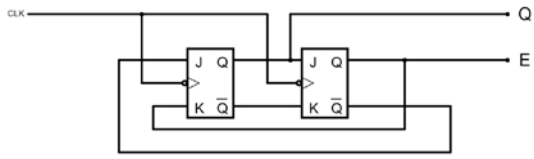
Nový výstup BUSY hodnotou 1 oznamuje, že procesor provádí operaci „načti-a-zapiš“ a během té doby by neměl žádný jiný obvod měnit obsah paměti, popřípadě že načítá dvoubytový operand.

AVMA oznamuje, že procesor bude požadovat přístup ke sběrnici. Pokud je v log. 0, oznamuje, že je procesor ve stavu HALT nebo SYNC (stojí a čeká).

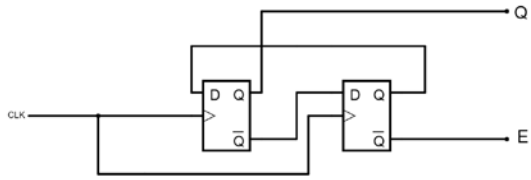
LIC oznamuje, že procesor vykonává poslední instrukční cyklus (Last Instruction Cycle). Sestupná hrana na tomto výstupu znamená, že během následujícího taktu se bude číst operační kód další instrukce.

Vstup TSC řídí datovou sběrnici, adresní sběrnici a výstup R/W. Pokud je v log. 0, procesor funguje normálně, pokud je v log. 1, procesor přepne zmíněné vývody do třetího stavu (odpojeno).

A konečně vývody E a Q změnily směr. Zatímco u 6809 to jsou výstupy, u 6809E to jsou vstupy a o jejich generování se musí postarat vnější obvody. Nejčastější zapojení je pomocí dvojice klopných obvodů J-K nebo dvojice klopných obvodů D:



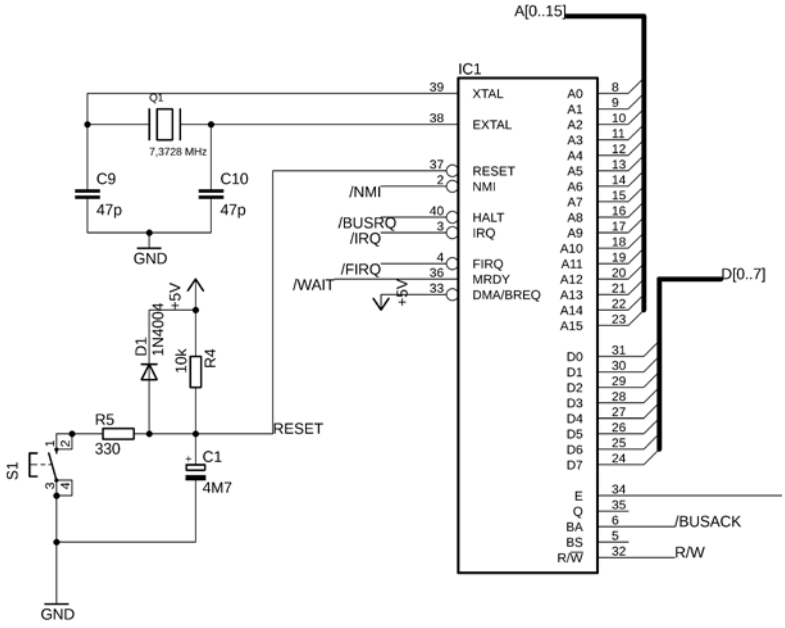
Obrázek 50: generování hodin Q, E pomocí 74LS72



Obrázek 51: generování hodin Q, E pomocí 74HC74

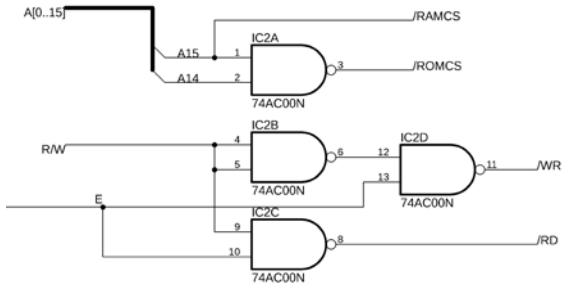
14.4 OMEN Kilo CPU

Základní deska vychází z koncepce OMEN Bravo. Obsahuje jen procesor, 32 kB RAM, 8 kB EEPROM, dekodér signálů a nezbytné obvody okolo (RESET, krystal). Nakonec jsem k „nezbytným obvodům“ přidal i nám důvěrně známý 68B50 pro sériovou komunikaci. Díky tomu může být OMEN Kilo použit jako jednodeskový počítač, bez jakýchkoli dalších obvodů.



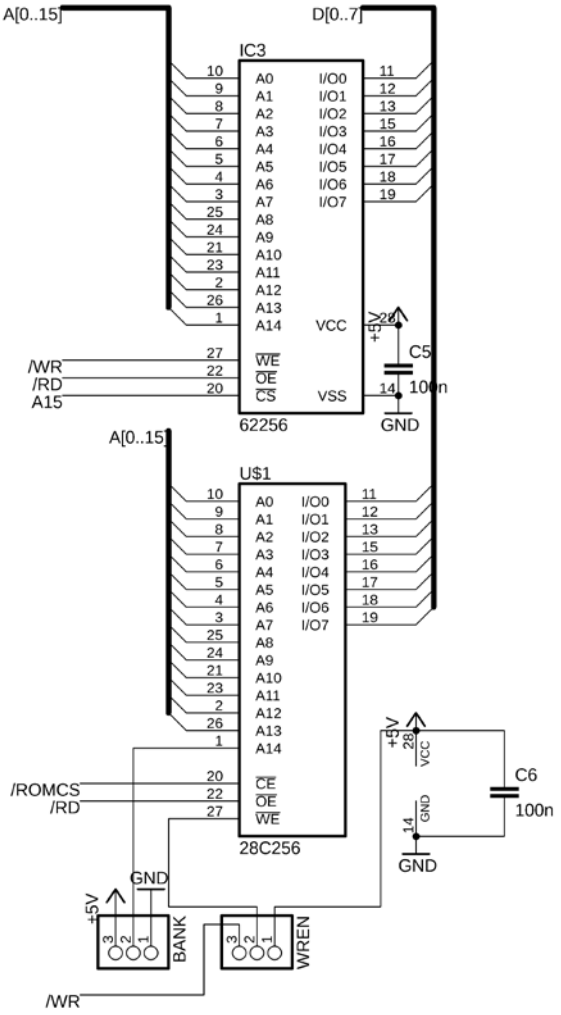
Obrázek 52: Kilo, CPU

Na zapojení procesoru není nic, co bychom už neviděli. Vstup DMA zůstane nepoužitý, je proto připojen na napájecí napětí, výstupy Q a BS jsou ponechány nezapojené.



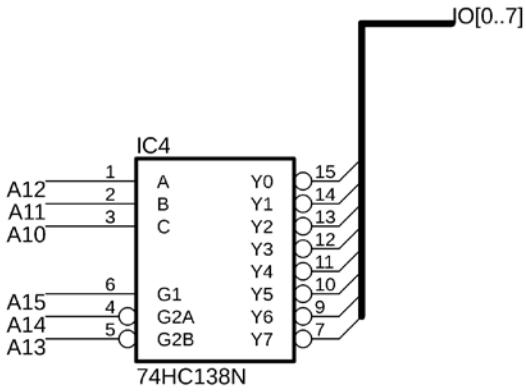
Obrázek 53: Kilo, dekodér

Dekodér pamětí a signálů /RD, /WR funguje naprosto stejně jako u počítače Bravo. Opět stačí jeden obvod 7400. Jen místo výstupu PHI2 je použit výstup E se stejnou funkcí.



Obrázek 54: Kilo, paměti

Paměti jsou vyřešené doslova stejně jako u předchozích počítačů. Na místě EEPROM jsem použil obvod 28C64 s kapacitou 8 kB, ale schéma je připravené i pro 28C256 – pak je potřeba přepínačem BANK zvolit vybranou polovinu paměti.

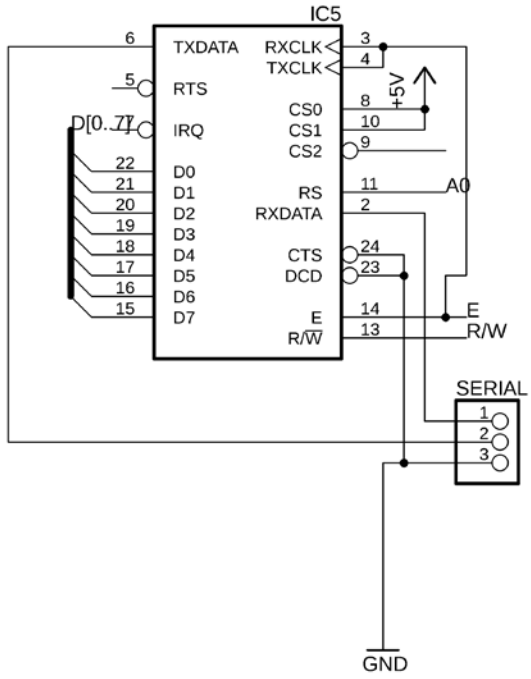


Obrázek 55: Kilo, dekodér periferií

Dekodér pro periferie je zapojen tak, aby využíval adresy, kde je A15 = 1 a A14 = A13 = 0, tedy 8000h až 9FFFh. Pro každou periferii je vyhrazen prostor 1 kB, a to takto:

Periferie	Adresa
IO0	8000h – 83FFh
IO1	9000h – 93FFh
IO2	8800h – 8BFFh
IO3	9800h – 9BFFh
IO4	8400h – 87FFh
IO5	9400h – 97FFh
IO6	8C00h – 8FFFh
IO7	9C00h – 9FFFh

Periferii IO0 věnujeme, podobně jako u Brava, sériovému portu.



Obrázek 56: Kilo, sériové rozhraní s ACIA 6850

Vstup RS, výběr registru, připojíme opět k A0, takže se sériový port objeví jako dvojice registrů na adresách 8000h – 83FFh. Na sudých to bude řídicí registr, na lichých datový. Opět doporučuju použít takové adresy, které mají v nedekódovaných bitech 1, tedy využít adresy 83FEh (řídicí) a 83FFh (datový).

Poslední část je připojení na sběrnici. Nevyvádím kompletní systémové sběrnice, pouze datovou a z adresní jen tři nejnižší bity. Kromě těchto signálů na sběrnici vyvádím i signály /WR, /RD, E, RESET, vstupy /WAIT (jiný název pro MRDY), /BUSRQ (jiný název pro HALT), /IRQ a /FIRQ.

14.5 Koncept „backplane“

Poloprofesionální a profesionální počítače z 80. let často používaly tento koncept, čímž se lišily od většiny „domácích počítačů“, které bývaly celé na jedné desce plošných spojů. Koncept „backplane“ pracoval s tím, že existovalo několik desek, zapojených do společné sběrnice. Tato sběrnice byla nejčastěji „vzadu“ v konstrukci počítače, proto „backplane“.

Počítač pak sestával z nějaké základní desky, na níž byl procesor a nezbytné obvody okolo, většinou i paměť RAM a EPROM. Dále pak bývaly k dispozici desky paralelních portů, sériových portů, rozhraní pro klávesnici, displej, pro diskety, rozšiřující paměť atd.

Například český počítač SAPI-1 sestával z několika základních desek: JPR-1, která obsahovala procesor a minimum paměti, ARB-1, což byla deska backplane sběrnice, REM-1, která rozšiřovala paměť, ANK-1 pro připojení alfanumerické klávesnice, AND-1, což byl alfanumerický televizní displej. K této sestavě existovalo velké množství dalších desek pro řízení, měření, ovládání, dokonce i alternativní desky JPR-1A (bez paměti RAM) a JPR-1Z (s procesorem Z80)…

Podobná stavebnicová koncepce počítače přináší vyšší variabilitu – můžete si poskládat sestavu tak, jak vám bude vyhovovat. Ostatně podobná koncepce stojí za úspěchem systémů IBM PC a dodnes se s ní u PC setkáváme. Jen místo „backplane“ říkáme „motherboard“ a sběrnice už dávno nejsou takové, jaké byly v době osmibitové.

OMEN Kilo se tak vlastně promění v sérii malých a jednoduchých desek, pospojovaných do jednoho systému. Díky tomu bude celý systém variabilnější. Komu stačí připojení přes sériový port, ten použije jen desku procesoru a sériového portu, kdo bude chtít víc, použije třeba modul pro připojení externí paměti, displeje apod.

15 Základy programování v assembleru 6809

15 Základy programování v assembleru 6809

Už jsem zmínil, že 6809 je mnohými vývojáři považován za nejlépe navržený osmibitový procesor své doby. Pojďme se podívat, čím je okouzlil.

Mnoho věcí, s nimiž se v assembleru 6809 potkáte, vám bude povědomých z kapitol o 6502. Jak už jsem psal: oba tyto procesory mají společného předka. U 6809 přímého, u 6502 spíš jako inspiraci.

Procesor 6809 má i svou vylepšenou variantu od Hitachi s označením 6309. Protože jsou oba modely stále k sehnání, rozhodl jsem se i 6309 zahrnout do popisu.

15.1 Adresní módy 6809

Podobně jako u 6502 má i u 6809 každá instrukce několik různých adresních módů, podle nichž se určuje, kde je uložený operand instrukce, tedy data, s nimiž instrukce pracuje.

Efektivní adresa

Před popisem samotných adresních módů je na místě říci si pár slov o konceptu *efektivní adresy*. U procesoru 6809 se pod *efektivní adresou* myslí adresa v paměti, kde je uložený operand – data, s nimiž se bude pracovat. Protože 6809 intenzivně využívá práci s pamětí (na rozdíl od 8080 a jemu podobných nemá tolik vnitřních registrů), musí vnitřní logika nejprve zjistit, s jakou adresou v paměti se má pracovat. Někdy bývá zadána přímo, ale často je výsledkem kratšího výpočtu.

Inherent

První adresní mód je mód **inherent** – odpovídá tomu, co jsme si u 6502 nazvali módem implicitním. Tedy přímo v samotné instrukci je zakódováno, s jakými daty a odkud pracuje. nejčastěji jde o data v některém z registrů. Kupříkladu instrukce CLRB nuluje obsah registru B. U tohoto adresování nemá smysl mluvit o efektivní adrese.

Značení registrů u 6809 je následující:

Registr	Označení	Jméno (v assembleru)
Akumulátor A	ACCA	A
Akumulátor B	ACCB	B
Spojený akumulátor A:B	ACCD	D
Indexový registr X	IX	X
Indexový registr Y	IY	Y

Registr	Označení	Jméno (v assembleru)
Ukazatele zásobníku	SP, US	S, U
Programový čítač	PC	PC
Registr stránky pro přímý přístup	DPR	DP
Registr příznaků	CCR	CC

Delší pojmenování se používá v případech, kdy se popisují jednotlivé registry. Kratší notaci používají assemblery.

Immediate

Druhý adresní mód je nazvaný **immediate**, v češtině se používá označení „přímý". Přímý operand může být osmibitový nebo šestnáctibitový a následuje přímo za operačním kódem instrukce. Příkladem mohou být instrukce pro přičítání konstant k akumulátorům nebo pro zápis konstantních hodnot do registrů. Assemblery používají symbol # (hash) pro označení, že instrukce má pracovat přímo se zadanou hodnotou:

```
0823  86 12      LDA #$12
0825  8E FE DC   LDX #$FEDC
0828  10 42      LDY #$42
```

Znakem $ se v mnoha assemblerech označuje hexadecimální hodnota (obdoba „céčkovské" sekvence 0x…) *U procesoru 6809 (nebo Z80) je to zažitá konvence, proto ji budu ve výpisech používat místo konvence 1234h, jakou jsem používal třeba u procesoru 8080.*

Efektivní adresa je u tohoto adresování rovna adrese, na níž se nachází operand. U výše ukázaných instrukcí to je tedy po řadě 0824h, 0826h a 0829h.

Extended

Rozšířená adresa – **extended** – odpovídá módu, který měl u 6502 název *absolute*. Za operačním kódem instrukce následuje přímo kompletní efektivní adresa, tedy dva bajty adresy.

```
0200   B6 12 34   LDA $1234
0203   BE FE DC   LDX $FEDC
```

První instrukce načte do registru A obsah na adrese 1234h. Druhá instrukce načítá obsah do registru IX – a protože je tento registr šestnáctibitový, budou se načítat dva bajty. První, vyšší část, z adresy FEDCh, druhý bajt z FEDDh. Vlastně se jedná o přímo zapsanou kompletní efektivní adresu.

Extended indirect

Nepřímá varianta výše uvedeného adresování. Za operačními kódy instrukce následuje adresa, z níž se načtou dva bajty efektivní adresy. Až tato adresa určuje, kde leží operand.

```
0200   BE 02 13     LDX test
0203   AE 9F 02 13  LDX [test]
…
0213   02 04        test: DW $204
```

První instrukce LDX načte do X hodnotu, která je uložena na adrese test, tedy 204h (rozšířené adresování). Druhá instrukce LDX, která používá nepřímé adresování (vyznačeno hranatými závorkami), nejprve načte adresu (2 bajty) z adresy test (=204h). To je efektivní adresa této instrukce, a na této adrese bude hledat operand. Zde to je 9F02h (obsah z adres 204h, 205h). V registru IX bude tedy hodnota 9F02.

Direct

Adresní mód **direct** je podobný módu *extended*, ale nepoužívá plnou 16bitovou adresu, ale pouze zkrácenou, osmibitovou. Těchto osm bitů je použito jako spodní polovina efektivní adresy. Horní polovina je tvořena obsahem registru DPR (Direct Page Register).

Pokud je v DPR nula, je tento mód ekvivalentní módu „zero page“, jaký známe z 6502.

Assembler se často rozhoduje, jaký mód použije, podle toho, kam směřuje cílová adresa. U 6502 to je jednoduché: pokud je cílová adresa v rozmezí 0-255, jde o zero page a lze použít speciální mód. U 6809 může být „direct page“ kdekoli. Assembler ale neví, kam je právě nastavený registr DPR, a proto je mu potřeba pomoci. Slouží k tomu direktiva SETDP.

SETDP má jeden parametr, a to je číslo v rozsahu 0 – 255. Toto číslo odpovídá hornímu byte adresy, a assembler bude předpokládat, že stejná hodnota je v registru DPR. Viz následující příklad:

```
0001                        .ORG   1
0001   00           TEST0:  DB   0
0102                        .ORG   $102
0102   00           TEST1:  DB   0
0103                           ;
0203                        .ORG   $203
0203   00           TEST2:  DB   0
0204                           ;
1000                        .ORG   $1000
1000                           ;
1000   96 00                LDA   0
```

```
1002                              ;
1002   96 01                  LDA   test0

1004   B6 01 02               LDA   test1
1007   B6 02 03               LDA   test2
100A                              ;
100A                           SETDP   1
100A   B6 00 01                LDA   test0
100D   96 02                   LDA   test1
100F   B6 02 03                LDA   test2
1012                              ;
1012                           SETDP   2
1012   B6 00 01                LDA   test0
1015   B6 01 02                LDA   test1
1018   96 03                   LDA   test2
```

Jsou definovány tři paměťové buňky: test0 na adrese 0001h, test1 na adrese 0102h a test2 na adrese 0203h. Překladač překládá tři instrukce LDA, a volí mezi módy direct / extended podle toho, co mu řekne direktiva SETDP.

Důležité je, že direktiva SETDP slouží jen pro informaci překladače. Sama nijak obsah registru DPR nenastavuje ani nemění.

Register

Tento mód opět nepoužívá efektivní adresu. Operandem jsou některé z registrů.

```
TFR X,Y
EXG A,B
PSHS A,B,X,Y
PULU X,Y,D
```

První instrukce přesune obsah registru IX do registru IY (všimněte si, že pořadí je opačné, než na jaké jsme byli dosud zvyklí – přiřazuje se zleva doprava). Druhá vymění obsah registrů A a B. Třetí uloží na systémový zásobník registru ACCA, ACCB, IX a IY, čtvrtá přečte z uživatelského zásobníku registry IX, IY a D.

Indexed

Adresace s indexem je u procesoru 6809 velmi komplexní a silný mód. Obecně lze říct, že indexovaná adresace pracuje s hodnotou v některém z šestnáctibitových registrů IX, IY, SP, US, někdy i s programovým čítačem PC, k níž se přičítá offset, aby se získala efektivní adresa (na které je samotný operand).

Operační kód instrukce, která používá tento mód adresace, je dvoubajtový. Za samotným operačním kódem následuje takzvaný **postbyte**, který říká, o jaký typ indexace jde. Někdy následuje i další bajt či dva samotného indexu.

Nejjednodušší index je **nulový index**. K hodnotě v indexovém registru (nemusí být jen IX a IY, ale i ukazatelé zásobníku, viz výše)

```
LDD 0,X
LDA Y
```

První instrukce znamená „vezmi hodnotu v X, přičti nulu a výsledek udává efektivní adresu". Druhá instrukce udělá totéž, ale zápis je zkrácený, vynechává se „0,".

Další typ indexace je **konstantní index**. Konstantní index je vždy se znaménkem a může být

- pětibitový (-16 až +15) – v takovém případě je hodnota uložena přímo v postbyte
- osmibitový (-128 až +127) – za postbyte následuje jeden bajt
- šestnáctibitový (-32768 až 32767) – za postbyte následují dva bajty

Assembler vybírá nejvhodnější variantu sám, takže programátor se většinou nemusí starat o to, který index použít.

```
LDA 42,X
LDX -5,U
LDY 5000,X
```

Instrukce má tvar *index,registr*.

Třetí typ indexu, **akumulátorový**, je podobný předchozímu, ale index není konstantní, nýbrž je v některém z akumulátorů ACCA, ACCB, ACCD (16bitová varianta). Hodnota v registru je opět brána jako hodnota se znaménkem, je přičtena k hodnotě v indexovém registru, a tak je získána efektivní adresa.

```
LDA B,Y
LDX D,U
```

Poslední typ indexu je **index s postinkrementací / predekrementací**. Tento mód je ideální pro procházení tabulek hodnot nebo pro práci se zásobníkem. Postinkrementace funguje tak, že z indexového registru je načtena hodnota efektivní adresy a poté je hodnota v indexovém registru zvýšena o 1 nebo 2:

```
LDA ,X+
LDD ,Y++
```

První instrukce tedy vezme adresu z registru IX, do registru ACCA přesune hodnotu z této adresy a pak obsah v registru X zvýší o 1. U druhé instrukce se bere efektivní adresa z registru Y, hodnota z této adresy je načtena do registru ACCD (16 bytů) a pak je hodnota v IY zvýšena o 2.

Predekrementace funguje obdobně, jen s tím rozdílem, že hodnota v indexovém registru je *nejprve* snížena o 1 či 2, a teprve poté je s ní nakládáno jako s efektivní adresou.

```
LDA ,-X
LDD ,--Y
```

Představte si, že provedete operaci STX, která ukládá obsah registru X do paměti, a přitom použijete režim s postinkrementací:

```
STX ,X++
```

Co se stane? Nejprve je do dočasného registru efektivní adresy načtena hodnota v IX. Poté je obsah registru IX zvýšen o 2, a tato nová hodnota je uložena do paměti na efektivní adresu. Pokud je před provedením instrukce v registru IX hodnota 1200h, je výsledek ten, že se na adresy 1200h a 1201h uloží šestnáctibitová hodnota 1202h. Tatáž hodnota zůstane i v registru IX.

Indexed indirect

Připadá vám, že teď už je sada možných způsobů adresování kompletní? Tak si prosím ke všem předchozím možnostem, s výjimkou postinkrementu/predekrementu o 1 a pětibitového konstantního offsetu, přidejte ještě možnost nepřímé adresace. Tedy takové, že z efektivní adresy, kterou byste získali prostou indexovou adresací, načtete dva bajty, a teprve tyto vám dají reálnou efektivní adresu. Příklad:

```
0200 LDA [$10,X]
…
0310 03
0311 12
0312 55
…
```

Dejme tomu, že v registru IX je hodnota 300h. První krok, indexace, vezme tuto hodnotu, přičte k ní 10h a získá tak efektivní adresu 310h.

U indexové adresace by se z této adresy načetl jeden bajt (03h), uložil do registru ACCA a hotovo. U nepřímé indexové adresace (indikují ji hranaté závorky) se z této adresy načtou dva bajty

(0312h), které dají novou efektivní adresu. Až z ní je teď načtena hodnota (55h) a uložena do registru ACCA.

Relative

Procesor 6809 nabízí možnost relativní adresace vzhledem k obsahu programového čítače. K jeho obsahu je přičten osmi-, nebo šestnáctibitový offset, a výsledek udává efektivní adresu. Na tuto adresu můžete skákat pomocí instrukcí skoků, popřípadě pracovat s daty zde uloženými.

Relative indirect

Nepřímá varianta předchozího adresování. Získaná adresa slouží jen jako ukazatel do paměti, z níž je přečtena teprve reálná efektivní adresa.

15.2 Jak to ten procesor dělá?

U procesorů 8080 a 6502 platilo, že co instrukce, to jeden jednobajtový kód. U Z80 jsme se setkali s takzvanými *prefixy*, což byly speciální kódy, které znamenaly: Pozor, následující kód má pozměněný význam. U 6809 je koncepce odlišná. Není tu vlastně velké množství různých instrukcí, jen velké množství adresních módů a instrukcí mnohem méně. Návrháři tedy pro každou instrukci, u níž to má smysl, vyhradili několik operačních kódů. Nejčastěji jeden pro přímý operand, druhý pro přímou adresu (1 bajt adresy, Direct Page), třetí pro plnou přímou 16bitovou adresu (Extended) a čtvrtý pro všechny možné indexované varianty. V takovém případě následuje za operačním kódem již zmiňovaný postbyte, v němž je zakódováno, který mód se přesně používá.

Postbyte je použit i u instrukcí ukládání na zásobník / čtení ze zásobníku / přenosů dat mezi registry. U těchto instrukcí udává, jakých registrů se operace týká.

15.3 Postbyte

7	**6**	**5**	**4**	**3**	**2**	**1**	**0**	**M**	**T**	**#**
0	R	R	F	F	F	F	F	(+/- 4 bit offset),R	1	0
1	R	R	0	0	0	0	0	,R+	2	0
1	R	R	I	0	0	0	1	,R++	3	0
1	R	R	0	0	0	1	0	,-R	2	0
1	R	R	I	0	0	1	1	,--R	3	0
1	R	R	I	0	1	0	0	,R	0	0
1	R	R	I	0	1	0	1	(+/- B),R	1	0

1	R	R	I	0	1	1	0	(+/- A),R	1	0
1	X	X	X	0	1	1	1	Illegal	u	u
1	R	R	I	1	0	0	0	(+/- 7 bit offset),R	1	1
1	R	R	I	1	0	0	1	(+/- 15 bit offset),R	4	2
1	X	X	X	1	0	1	0	Illegal	u	u
1	R	R	I	1	0	1	1	(+/- D),R	4	0
1	X	X	I	1	1	0	0	(+/- 7 bit offset),PC	1	1
1	X	X	I	1	1	0	1	(+/- 15 bit offset),PC	5	2
1	X	X	X	1	1	1	0	Illegal	u	u
1	0	0	1	1	1	1	1	[address]	5	2

Sloupec T udává, kolik taktů hodin je potřeba přidat k základní době trvání instrukce, sloupec # pak říká, kolik bajtů dat následuje za postbyte.

Bity, označené F, udávají hodnotu offsetu (pětibitové číslo, resp. čtyřbitové se znaménkem). Bit I udává, jestli se jedná o adresování nepřímé (1), nebo přímé (0). Bity R udávají indexový registr, s nímž se pracuje: 00 je X, 01 je Y, 10 je U a 11 je S.

Postbyte mají i instrukce pro záměnu a přenos (TFR a EXG). Postbyte těchto instrukcí se dělí na horní část (bity 4 – 7), která udává zdroj, a dolní část (bity 0 – 3), která udává cíl.

0000	D
0001	X
0010	Y
0011	U
0100	S
0101	PC
1000	A
1001	B
1010	CC
1011	DP

Postbyte používají i instrukce PSHx a PULx. V tomto postbyte jsou označené registry, které se ukládají nebo čtou.

7	6	5	4	3	2	1	0
PC	S/U	Y	X	DP	B	A	CC

15.4 Instrukce pro přesun dat

Začneme opět tím nejjednodušším tématem. Tedy zdánlivě, ve skutečnosti jsou, díky bohatosti adresních módů, přesuny dat u 6809 velmi silný nástroj, ačkoli by se mohlo zdát, že nabízejí jen dvě základní instrukce, totiž LDr a STr, kde r je registr (osmibitový nebo šestnáctibitový)

LDr, STr

LD (Load) jsou instrukce pro přesun z paměti do registru, instrukce ST (Store) slouží k přesunu dat z registru do paměti. Instrukce LD nabízí adresu přímou (Direct), rozšířenou (Extended) nebo indexovanou (všechny možné varianty), popřípadě přímou hodnotu operandu (Immediate) pro načtení konstanty. Instrukce ST nabízejí stejnou sestavu módů, s výjimkou módu immediate, který u ukládání nedává smysl (jak uložit obsah registru do konstanty?).

Osmibitové adresy pracují s registrem A nebo B: LDA, LDB, STA, STB. Mají jeden operand:

```
LDA #$12
LDA $12
LDA $1234
LDA ,X
LDA -X
STB $12
STB $1234
STB Y+
```

Šestnáctibitové varianty pracují s registry D (což jsou ve skutečnosti spojené registry A a B, kde registr A tvoří vyšší byte), S, U, X a Y. Jinak platí totéž, co u osmibitových variant: jméno registru tvoří součást názvu instrukce (LDD, STD, LDS, STS, LDU, STU, LDX, STX, LDY a STY) a instrukce má jediný operand, totiž efektivní adresu, ze které (nebo na kterou) se čtou / zapisují data.

LEAr

Autorům instrukční sady přišlo jako dobrý nápad, a on to dobrý nápad docela je, nabídnout instrukci, která se chová skoro jako LD, totiž spočítá efektivní adresu operandu v paměti, ale místo toho, aby si na tuto adresu sáhla pro operand, udělá to, že tuto adresu uloží do 16bitového registru.

Instrukce se jmenuje LEA (Load Effective Address) a má opět čtyři varianty podle toho, do kterého šestnáctibitového registru efektivní adresu ukládá: LEAX, LEAY, LEAS, LEAU.

```
LEAX $12
LEAX Y++
LEAX [$0E,U]
```

TFR

Probrali jsme instrukce přesunů z paměti a do paměti. Ano, jsou vlastně jen dvě, s jedním drobným bonusem pro vyhodnocování efektivní adresy. Co chybí, to jsou nějaké instrukce pro přesuny mezi vnitřními registry. Asi nepřekvapí, že je opět jen jedna, totiž TFR (Transfer). Instrukce má dva operandy, zdrojový a cílový registr:

```
TFR zdroj, cíl
```

Pozor! Pořadí operandů je přesně opačné, než bývá zvykem např. u assembleru 8080, Z80 nebo x86. Nejprve je zdroj, poté cíl. Zápis tedy neodpovídá obvyklé formě „a = b“, známé z programovacích jazyků, ale spíš „b->a“. Ale hodně záleží na konvencích překladače – maně si vzpomínám, že inline assembler x86 v překladači GCC používá stejný zápis.

Operandy musí mít stejnou velikost. Tedy buď 16 bitů (D, X, Y, U, S, PC), nebo 8 bitů (A, B, DP, CC).

Pozor, trik! U 16bitových přenosů můžeme instrukci TFR nahradit instrukcí LEA s indexovým adresováním, která bývá rychlejší. Například místo TFR Y,U pro přenos obsahu registru Y do registru U můžeme použít „LEAU ,Y“ – která spočítá efektivní adresu (= Y) a přenese ji do registru U. Obě instrukce mají 2 bajty, ale LEA trvá jen 4 cykly, zatímco TFR 6.

EXG

Když už je instrukce TFR, proč nenabídnout i její variantu, která nepřesouvá data ze zdroje do cíle, ale navzájem prohodí obsah zdrojového a cílového registru (i když zde označení „zdroj“ a „cíl“ ztrácí smysl). Jinak platí to, co bylo napsáno o TFR. Bohužel, interní přenos probíhá přes skrytý osmibitový pracovní registr, takže výměna obsahu mezi šestnáctibitovými registry probíhá jako dvě osmibitové výměny:

```
EXG A,B
EXG X,U
```

```
TFR A,CC
TFR B,DP
```

15.5 Operace se zásobníkem

Když se podíváte na sestavu indexových adresních módů, napadne vás, že vlastně přímé operace se zásobníkem nemají příliš význam. Uložení registru A na zásobník vlastně zastane instrukce STA –S, načtení obsahu registru D ze zásobníku zařídíme pomocí LDD S++, a je hotovo.

Jenže jak jsem zmiňoval v úvodu: vývojáři si před návrhem instrukční sady udělali analýzu mnoha tisíců programů, takže jim neuniklo, že instrukce PUSH a POP chodí často spolu. Po vstupu do podprogramu se ukládají registry na zásobník, při návratu se zase načítají. Proto navrhli instrukce tak, aby pracovaly rovnou se sadou více registrů.

Za instrukčním kódem PUSH a PULL (tak se u 6809 nazývá instrukce, jinde známá jako POP) následuje jeden bajt, který udává, jakých registrů se operace týká. Registrů může být až osm: PC, U, Y, X, DP, B, A, CC. První čtyři jsou šestnáctibitové, poslední čtyři osmibitové. Pokud je v informačním bajtu příslušný bit nastaven, registr se ukládá, pokud je bit nulový, registr se přeskočí.

Instrukce ukládání na zásobník postupuje ve výše zmíněném pořadí, od PC k CC, instrukce načtení postupuje, logicky, v obráceném pořadí.

A protože má procesor 6809 dva ukazatele zásobníku, nazvané S a U (Systémový a Uživatelský), jsou i dvě dvojice instrukcí: PSHS, PULS pro práci se systémovým zásobníkem, PSHU a PULU pro práci s uživatelským zásobníkem.

Drobný rozdíl je ten, že instrukce PSHU / PULU neumožňuje uložit obsah registru U, namísto toho ukládá obsah registru S. U těchto instrukcí je tedy sada registrů lehce obměněná: PC, S, Y, X, DP, B, A, CC.

```
PSHS A,B,X,Y
PSHU X,Y,S,PC
```

Na pořadí registrů při zápisu nezáleží, pořadí je pevně dané a odpovídá tomu, co jsme si popsali výše – ukládá se od PC a končí se CC.

Všimněte si, že mezi registry je i programový čítač. Pokud instrukce PULS, PULU obnovuje i obsah registru PC, je jasné, že se tím zároveň provede de facto skok na uloženou adresu, tedy vlastně návrat z podprogramu:

```
SUB1: PSHS X,Y,A,B
      … nějaké operace
      PULS X,Y,A,B
      RTS ;návrat z podprogramu
          ; načte ze zásobníku SP obsah
          ; čítače instrukcí PC
```

Tato konstrukce může být vlastně přepsána následujícím způsobem:

```
SUB1: PSHS X,Y,A,B
      … nějaké operace
      PULS X,Y,A,B,PC
```

Poslední PULL obnoví uložené registry, a jako poslední si přečte uložený obsah registru PC, což je návratová adresa.

15.6 Příznaky

O registru příznaků (CC, Condition Code) jsem se zmiňoval už v kapitole o architektuře, ale zopakujme si, že jeho bity jsou (od nejvyššího):

Bit	Název	Funkce
7	E	Entire – při obsluze přerušení říká, zda je na zásobníku kompletní sada registrů, nebo jen PC a CC
6	F	FIRQ mask: když je 1, je zakázáno přerušení FIRQ
5	H	Half carry – přenos mezi horním a dolním půlbyte během sčítání
4	I	IRQ mask: když je 1, je zakázáno přerušení IRQ
3	N	Negative – záporný výsledek
2	Z	Zero – nulový výsledek
1	V	Overflow – přetečení
0	C	Carry – přenos

Bity 6 a 4 nastavuje programátor a pomocí nich může maskovat přerušení IRQ a FIRQ. Bit 7 informuje o tom, jestli při vstupu do obslužné rutiny přerušení byl na zásobník uložen stav všech registrů (jako u IRQ nebo NMI), nebo jen PC a CC. S tímto příznakem pracuje instrukce RTI (návrat z přerušení) a podle něj správně obnoví obsah uložených registrů.

Zbývající bity jsou klasické příznaky, jaké známe z jiných procesorů.

Příznak Z (zero) je nastaven, pokud výsledek předchozí operace byl 0. Tento příznak nastavují nejen aritmetické a logické operace, ale i instrukce load a store!

Příznak N (Negative) je 1, pokud je výsledek předchozí operace záporný. A protože procesory používají znaménkovou aritmetiku tak, že nejvyšší bit udává znaménko čísla, je tento příznak roven nejvyššímu bitu výsledku. Tento příznak opět nastavují nejen aritmetické a logické operace, ale i instrukce LD a ST.

Příznak C (Carry) označuje přenos z nejvyššího bitu při aritmetických operacích. Pokud sčítáme dvě jednobajtová čísla (bez znaménka) a výsledek je větší než 255, je nastaven přenos. Podobně tento příznak udává, že při odečítání (a podobně u porovnávání, což je technicky shodné s odečítáním) výsledek klesl pod 0 a bylo nutno si „vypůjčit" jedničku do nejvyššího bitu. Tento příznak nastavují pouze aritmetické instrukce.

Příznak V (Overflow) je podobný příznaku Carry, ale s tím rozdílem, že udává přetečení / podtečení nikoli z nejvyššího bitu, ale z druhého nejvyššího. U operací nad čísly se znaménkem nahrazuje příznak C, protože nejvyšší bit u těchto čísel udává znaménko. Proto nedává smysl řešit přenos z nejvyššího bitu.

Příznak H (Half carry) je podobný příznaku C, ale místo přenosu z nejvyššího bitu udává tento bit, zda došlo k přenosu mezi 3. a 4. bitem výsledku (tedy v polovině bajtu). Jeho hodnota se používá při práci s čísly v kódu BCD a nastavují ho správně pouze instrukce sčítání (operace odčítání jej ale mění také).

Instrukce pro změnu příznaků

Zatímco dříve probírané procesory měly speciální instrukce pro nastavování toho či onoho příznaku, procesor 6809 to celé spláchl osmibitovými variantami instrukcí AND a OR, nazvanými ANDCC a ORCC. Tyto instrukce mají pouze jediný adresní model, totiž přímý operand, a dělají to, co byste čekali: operace AND nebo OR mezi obsahem registru CC a přímo zadaným operandem. To stačí, protože pomocí OR snadno nastavíme bity do log. 1 a pomocí AND je nastavíme do log. 0. U OR platí, že výsledek operace bude mít 1 všude tam, kde má operand logické 1, u AND platí, že výsledek má log. 0 tam, kde má operand logické 0. U nastavování proto použijeme operand s jedničkami tam, kde chceme mít ve výsledku 1, ostatní bity budou nulové. Při mazání bude mít operand nuly tam, kde chceme vynulovat hodnotu bitu, ostatní budou 1.

Nastavení bitů F a I pro zamaskování obou přerušení tedy zařídíme snadno pomocí ORCC #$50 (50h = 01010000b), vynulování příznaku I (což je bit 4) vyřešíme pomocí ANDCC #$EF (=11101111b).

15.7 Skoky

Procesor 6809 nabízí standardní sadu skokových instrukcí. Jsou to instrukce nepodmíněného skoku (JMP), nepodmíněného volání podprogramu (JSR), návratu z podprogramu (RTS, RTI) a podmíněných skoků (Bxx, LBxx).

JMP

Základní instrukce nepodmíněného skoku. Umožňuje skok na zadanou adresu. Adresa může být krátká, dlouhá (direct, extended), popřípadě indexovaná, kde asi největší smysl dávají skoky na adresy relativní k obsahu registru PC. Máme tak k dispozici plně relativní skoky v kompletním rozsahu.

JSR

Instrukce je obdobou předchozí, s tím rozdílem, že před skokem uloží na zásobník SP hodnotu PC, která ukazuje na následující instrukci.

RTS

Logický doplněk předchozí instrukce. Přečte ze zásobníku S uložený obsah registru PC, čímž se vyvolá návrat z podprogramu. Samozřejmě – pokud jste tam mezitím uložili něco jiného, skočí se někam úplně jinam.

RTI

Tato instrukce je podobná instrukci RTS s tím rozdílem, že před návratem obnoví ze zásobníku uložené registry. Podle obsahu příznaku E se obnovují buď registry PC a CC (E=0), nebo kompletní sada: CC, A, B, DP, X, Y, U a PC

15.8 Podmíněné skoky

Procesor nabízí i instrukce podmíněných relativních skoků, nazvaných Branch (Bxx). Mají krátkou variantu s osmibitovou relativní adresou, nebo dlouhou variantu (LBxx) s plným 16bitovým offsetem.

Varianty jsou:

Instrukce	Podmínka	Instrukce	Podmínka
BCC / BHS	C = 0	BCS / BLO	C = 1
BNE	Z = 0 (R != n)	BEQ_	Z = 1 (R == n)
BPL	N = 0 (R >= 0)	BMI	N = 1 (R < 0)
BVC	V = 0	BVS	V = 1

Instrukce	Podmínka	Instrukce	Podmínka
BHI	R > n	BLS	R <= n
BHS	R >= n	BLO	R < n
BGT	R > n (signed)	BLE	R <= n (signed)
BGE	R >= n (signed)	BLT	R < n (signed)

Podmínky jsou buď pojmenované podle testovaného bitu, nebo podle toho, jaký je očekávaný výsledek předchozí operace srovnání.

U bitových testů (V, C) je přidáno písmeno, které označuje, jestli očekáváme bit vynulovaný (clear), nebo nastavený na 1 (set): BCC (Branch if C is Clear), BCS (Branch if C is Set), BVC, BVS.

Instrukce BMI a BPL má mnemoniku „Branch if MInus“ / „Branch if PLus“.

Instrukce, které testují bit Z, se nejmenují ZC / ZS, ale odkazují k tomu, že nula je výsledkem porovnávací operace CMP, pokud obsah registru a testovaná hodnota jsou stejné. Proto se jmenují BEQ (Branch if EQual, Z = 1) a BNE (Branch if Not Equal, Z = 0)

Instrukce CMP totiž, podobně jako u jiných procesorů, dělá to, že od obsahu akumulátoru odečte zadanou hodnotu, podle výsledku nastaví příznakové bity a výsledek zahodí.

Zbývající skoky už netestují jednotlivé bity, ale jejich kombinace, a názvy odkazují k výsledkům porovnání:

CMPA #n	Signed	Unsigned
A > n	BGT (Greater Than)	BHI (Higher)
A >= n	BGE (Greater or Equal)	BHS (Higher or Same)
A < n	BLT (Less Than)	BLO (Lower)
A <= n	BLE (Less or Equal)	BLS (Lower or Same)

Ve skutečnosti třeba instrukce BGT testuje podmínku (Z or (N xor V) == 0), BLE její opak. BLS / BHI testují výsledek C or Z , BGE a BLT testují výraz N xor V. Instrukce BLO a BHS dokonce testují pouze bit C, a tak jsou to synonyma k BCC / BCS.

Kromě těchto instrukcí máme i dvě speciální: BRA a BRN. První testuje podmínku, která je vždy pravda, a tak skočí vždy (BRanch Always). Druhá testuje vždy nepravdivou podmínku, a tak neskočí nikdy (BRanch Never). BRA je vlastně totéž jako JMP s krátkou relativní adresou (a LBRA je totéž jako JMP s dlouhou relativní adresou), jen s tím rozdílem, že BRA je o 1 bajt kratší než JMP s relativní adresou a o 1 takt rychlejší.

Poslední instrukce z rodiny relativních skoků je instrukce BSR. Technicky odpovídá instrukci JSR, tedy skoku do podprogramu, jen s tím rozdílem, že tato instrukce je vždy relativní, a proto zabírá méně bajtů a je o něco rychlejší.

Všechny výše jmenované instrukce mají dlouhou variantu LBxx, která se liší tím, že není omezena na relativní rozsah -128 .. +127, ale nabízí plné relativní adresování v rozsahu -32768 .. +32767.

15.9 Aritmetické instrukce

Co by to bylo za procesor, kdyby nenabízel instrukce pro počítání. Samozřejmě je nabízí, a rozhodně neskrblí.

INC, DEC

Instrukce pro zvýšení obsahu registru nebo paměti o 1, popřípadě snížení o 1. Každá nabízí tři varianty, buď pro práci s akumulátorem (INCA, INCB, DECA, DECB), nebo pro práci s osmibitovým operandem v paměti (DEC, INC). Při práci s pamětí máme stejné možnosti výběru adresního módu jako třeba u instrukce STA – přímou adresu, plnou adresu, popřípadě indexovanou adresu.

Moment, říkáte si. Kde jsou instrukce INC a DEC pro práci s šestnáctibitovými registry? Nejsou. Jak to?

Většinou to totiž není potřeba. Šestnáctibitové registry se často používají pro přístup do paměti a nejčastější situace, kdy je potřeba zvýšit takový registr o 1 nebo snížit o 1, je tehdy, když se přistupuje do paměti. Tam je nejjednodušší použít adresní mód s postinkrementací nebo predekrementací, který je ještě o něco flexibilnější než INC a DEC.

Pokud opravdu potřebujeme zvýšit obsah registru (třeba X) o 1, můžeme použít LEAX 1,X. Připomeňme si: Tato instrukce spočítá hodnotu efektivní adresy (X + 1) a uloží ji do X. LEA můžeme pro podobné účely použít nejen pro přičítání a odčítání 1 – až do rozsahu -32 .. +31 zvládneme vše v rámci dvoubajtové instrukce, a pokud nevadí bajt či dva navíc, můžeme přičítat a odčítat naprosto jakoukoli hodnotu.

CLR

Instrukce CLR nuluje zadaný registr. Mohlo by se zdát, že to sem tak moc nepatří, ale pokud vám vadí, že je tato instrukce zařazená mezi aritmetické, představte si, že dělá operaci „násobení

hodnoty registru nulou“. Opět máme varianty podobné instrukcím INC, DEC. Tedy instrukce CLRA a CLRB, které nulují zadaný registr, a instrukci CLR, která nuluje bajt na zadané adrese (krátká / dlouhá / indexovaná).

COM, NEG

COM má stejné varianty jako předchozí CLR, totiž COMA, COMB a COM. Tato instrukce nenuluje, ale počítá dvojkový doplněk (complement). Vlastně udělá operaci R = –R. Udělá to tak, že hodnotu registru nebo paměti neguje bit po bitu a přičte jedničku.

NEG je podobná COM, se stejnými variantami (NEGA, NEGB, NEG) a možnostmi, ale místo doplňku počítá negaci, tedy změní všechny bity 0 na 1 a obráceně. R = NOT R.

TST

Instrukce TST (Test) má opět stejné varianty (TSTA, TSTB, TST) i adresovací možnosti jako COM či NEG. Tato operace zkontroluje hodnotu v registru či v zadané buňce paměti a podle ní nastaví příznaky Z a N.

ADD, ADC

Sčítání existuje ve verzi bez přenosu (ADDA, ADDB, ADDD) a s přenosem (ADCA, ADCB – 16bitová verze není). Operandem může být buď přímá hodnota, nebo adresa (přímá či rozšířená), popřípadě indexovaná adresa. Tedy stejné možnosti jako u instrukce LD.

SUB, SBC

Analogicky ke sčítání existují instrukce pro odčítání. Jsou zrcadlovým obrazem předchozích dvou, takže máme instrukce pro odčítání bez přenosu (SUBA, SUBB, SUBD) a s přenosem (SBCA, SBCB), obě se stejnými možnostmi zadávání operandu.

CMP

Porovnávání je technicky stejná operace jako odčítání. Od obsahu registru se odečte operand, nastaví se správně příznaky a výsledek se zahodí.

Instrukce CMP existuje v několika variantách. Kromě těch, co známe z ADD a SUB, tedy CMPA, CMPB a CMPD, nabízí i šestnáctibitové varianty CMPS, CMPU, CMPX a CMPY. Všechny mají operand definovaný stejně jako třeba SUB – tedy přímá hodnota, nebo krátká/dlouhá adresa, nebo indexovaná adresa.

ABX

Tato instrukce je bez operandů a bez variant. Jediné, co udělá, je, že k obsahu registru X přičte obsah registru B (bez znaménka, tedy např. 255 je opravdu 255 a ne -1).

Funkce je shodná jako u instrukce LEAX B,X, jen ABX zabírá o 1 bajt méně a je o 2 takty rychlejší.

SEX

Já vím, jak to vypadá, ale to to není. Jedná se o rozšíření čísla se znaménkem (Sign Extended). Operand je v registru B a výsledek bude v registru D. Instrukce dělá to, že do registru A uloží 0, pokud je B v rozsahu 0 – 127. Pokud je v rozsahu -128 až -1 (80h – FFh), tak do A uloží hodnotu FFh. Vlastně zkopíruje nejvyšší bit v registru B do všech bitů registru A.

DAA

Instrukce, která se vám bude hodit při práci s čísly ve formátu BCD. Její funkce je taková, že po operaci součtu, když jsou správně nastavené příznaky H a C, přičte k obsahu registru A hodnoty tak, aby zůstalo zachováno správné formátování BCD číslic. V praxi to znamená, že se přičítá 0, 6, 60h, 66h nebo 76h.

Proč zrovna 6? Odpověď je jednoduchá: vychází to z vlastností BCD čísel, které pro každou polovinu bajtu využívají jen 10 čísel z 16 možných kombinací. Pokud tedy například sčítáme 05h a 06h, je výsledek 0Bh, což je správně (=11), ale není to ve formátu BCD. V kódu BCD je to správně 11h (tedy 17). Ergo je potřeba přičíst těch 6 pro přeskočení hodnot A-F.

MUL

Instrukce násobení dvou osmibitových čísel bez znaménka. Vynásobí obsah registru A obsahem registru B a výsledek uloží do registru D.

Sice patří mezi jedny z nejpomalejších (11 taktů), ale stále je to proti procesorům, které jsme si představovali dosud, neskutečný luxus.

15.10 Logické instrukce, rotace a posuny

Další část instrukcí jsou instrukce, které nepracují s čísly, ale s jednotlivými bity.

AND, OR, EOR

Operace logického násobení (and, disjunkce), sčítání (or, konjunkce) a výhradního OR (exclusive OR, xor, nonekvivalence) mají vždy dvě varianty, pro registr A a pro registr B. Tedy ANDA, ANDB, ORA, ORB, EORA, EORB. Možnosti zadání operandu jsou stejné jako u aritmetických instrukcí: přímá hodnota, nebo krátká / dlouhá / indexovaná adresa.

Instrukce AND a OR mají i dříve zmíněné varianty ANDCC a ORCC, ale pouze s přímou hodnotou.

BIT

Instrukce BIT se má k instrukci AND jako instrukce CMP k instrukci SUB. Tedy méně krypticky: BIT je obdoba AND, ale po provedení operace a nastavení příznakových registrů se výsledek zahodí. Opět jsou dvě varianty, BITA a BITB, a čtyři možnosti zadání operandu.

ROL, ROR

Instrukce ROL a ROR rotují bitově obsah registrů A, B, nebo paměti. ROL doleva, ROR doprava. Při rotaci se rotuje přes příznak C.

Varianty jsou opět, jako u aritmetických instrukcí, ROLA, ROLB, ROL, RORA, RORB, ROR. Adresní módy jsou buď implicitní (když se pracuje s registry A, B), nebo „svatá trojice" krátká / dlouhá / indexovaná.

ROL posune bity o 1 doleva. To znamená, že bit 0 se přesune na pozici 1, bit 1 na pozici 2, bit 2 na pozici 3 a tak dál, a bit 7, který nám vypadne zleva ven, je zapsán do příznaku C, a původní hodnota z C je přesunuta do pozice 0 v registru A. ROR funguje stejně, jen obráceným směrem. Graficky to vypadá nějak takto:

Operace	Bity v registru / paměti								Příznak
	7	6	5	4	3	2	1	0	C
Výchozí stav	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	C
ROL	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	C	Bit 7
ROR	C	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0

LSL, LSR, ASL, ASR

Posuny se od rotací liší v tom, že „vypadnuvší" bit je přesunut do příznaku C, ale původní hodnota tohoto příznaku je zahozena a místo ní vstoupí zpět hodnota 0.

Logické posuny (LSL, LSR) posouvají doleva / doprava, a do uvolněného bitu doplňují log. 0. Aritmetický posun doprava (ASR) funguje stejně jako logický (LSR) a jedná se o synonymní označení pro stejnou instrukci. Aritmetický posun doprava funguje jako logický, ale v nejvyšším bitu je ponechána jeho původní hodnota.

Varianty opět stejné jako u rotací: ASLA, ASLB, ASL; ASRA, ASRB, ASR; LSLA, LSLB, LSL; LSRA, LSRB, LSR. Adresní módy také stejné jako u rotací.

Matematicky odpovídá posun doleva vynásobení hodnoty dvojkou, posun doprava pak celočíselnému dělení 2 (zbytek je v příznaku C). Grafické znázornění zde:

Operace	Bity v registru / paměti								Příznak
	7	6	5	4	3	2	1	0	C
Výchozí stav	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	C
ASL LSL	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	„0“	Bit 7
ASR	Bit 7	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
LSR	„0“	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0

15.11 Přerušovací a speciální instrukce

NOP

Instrukce, která nedělá nic. Nic nemění, nic neukládá, nic nepočítá, jen trvá dva takty.

Mimochodem, u procesoru 8085 trvala instrukce NOP 4 takty. U OMEN Alpha jich za sekundu proběhne při pracovní frekvenci 1,8432 MHz rovných 460 800. U OMEN Kilo s procesorem 6809 dvojnásobek, tedy 921 600. U Brava s procesorem 6502, který běží na kmitočtu 3,6864 MHz, je těch NOPů dokonce dvakrát tolik než u Kila (taky běží na dvojnásobné frekvenci).

SWI, SWI2, SWI3

Instrukce jsou v lecčems podobné instrukcím RST u procesoru 8080. SWI vyvolá „softwarové přerušení“, jehož chování odpovídá hardwarovému přerušení signálem /IRQ. Tedy: uloží se na zásobník kompletní obsah registrů kromě S, nastaví se bity I a F (tedy aby nemohlo přijít další přerušení) a zavolá se rutina, jejíž adresa je uložená na FFFAh (2 bajty).

SWI2 a SWI3 se liší v tom, že nemaskují další přerušení (neovlivňují bity I a F) a že svou adresu berou z jiného místa. SWI2 z adresy FFF4h a SWI3 z adresy FFFF2h (vždy 2 bajty).

Tabulka vektorů

Je vhodná chvíle shrnout si tabulku vektorů pro různé situace.

Adresa	Vektor pro
FFFEh, FFFFh	/RST
FFFCh, FFFDh	/NMI

Adresa	Vektor pro
FFFAh, FFFBh	SWI
FFF8h, FFF9h	/IRQ
FFF6h, FFF7h	/FIRQ
FFF4h, FFF5h	SWI2
FFF2h, FFF3h	SWI3
FFF0h, FFF1h	Vyhrazeno (využito u HD6309)

SYNC

Instrukce podobná instrukci HLT u 8080 nebo HALT u Z80. Zastaví chod procesoru a čeká na přerušení. Pokud přijde signál přerušení, není zamaskované a trvá alespoň 3 takty, ke vyvolána obsluha přerušení a procesor pokračuje v práci.

Pokud je přerušení zamaskované nebo impuls kratší, stane se pouze to, že se procesor probudí a pokračuje v práci.

CWAI

Instrukce je podobná předchozí instrukci SYNC s tím rozdílem, že CWAI má jeden přímo zadaný operand (immediate). Nejprve provede ANDCC s tímto operandem, aby programátor mohl případně vynulovat bity I a F, pak uloží na zásobník kompletní obsah registrů jako u ostatních přerušovacích rutin a čeká na přerušení. Po jeho příchodu zavolá příslušnou přerušovací rutinu.

Co se stane, když přijde požadavek /FIRQ, který, jak známo, očekává, že na zásobník se ukládá jen CC a PC? Pokud programátor nevyžaduje žádné speciální operace a používá standardní instrukce pro obsluhu přerušení, nestane se nic, protože instrukce RTI si zkontroluje bit E (který CWAI nastavuje) a správně obnoví obsah všech registrů i v rutině pro obsluhu /FIRQ.

15.12 Další instrukce

Už žádné další nejsou. To je vše. Vážně. Tohle, co jsme si teď ukázali, donutilo Billa Gatese k citovanému výroku. Posuďte sami, nakolik oprávněně.

15.13 Pseudoinstrukce (Direktivy)

U assembleru 680x se používaly mírně odlišné direktivy. Shrnuje je následující tabulka:

Význam	8080	Z80	6809
Uložení bajtů	DB	DEFB	FCB
Uložení slov	DW	DEFW	FDB
Vyhrazený prostor	DS	DEFS	RMB

15.14 Tipy

- Instrukce, které pracují implicitně s obsahem registru X (STX, LDX, CMPX) jsou kratší a rychlejší než jejich varianty s registrem Y (STY, LDY, CMPY). U indexovaného adresování je jedno, jestli použijete X nebo Y, tam to vliv nemá.
- Instrukce, které používají adresu z některého registru, jsou rychlejší a kratší než s přímo zadanou kompletní adresou. Pokud přistupujete několikrát po sobě ke stejnému místu v paměti, je dobré využít některý z šestnáctibitových registrů.
- Pokud potřebujete často využívat offsetů 0 až 31, zkuste popřemýšlet, jestli by nešlo hodnotu indexového registru zvýšit o 16 a používat offsety -16 až +15
- Hledáte šestnáctibitové sčítání? Co třeba použít LEAX D,X?

Nulování paměti

Dejme tomu, že máme za úkol do bloku 8 kB paměti od adresy 6000h nahrát samé nuly. Tedy na adresy 6000h až 7FFFh. Představte si, že je třeba například rychle smazat obsah obrazové paměti.

Instrukce CLR by samozřejmě šla použít, ale byla by pomalá. Pojďme využít rychlejší variantu:

```
      LDX #$6000
      LDU #0
LOOP: STU ,X++              ; 8 T
      CMPX #$6000+$2000     ; 4 T
      BNE LOOP              ; 3 T
```

Ve smyčce se instrukcí STU nulují dva bajty najednou a pomocí postdekrementace registru X máme postaráno o posunutí ukazatele. Smyčka tedy proběhne 4096x a zabere 4096 x 15 = 61440 T. U OMEN Kilo to tedy stihneme třicetkrát během sekundy, což je dostatečné pro mazání obrazovky, ale nic jiného už nestihneme.

Nejznámější způsob urychlení programu je obětovat pár bytů paměti a vnitřek smyčky „rozbalit“. Můžeme instrukci STU opsat 16x, to znamená, že jeden průchod smyčkou nuluje 32 bajtů a budeme potřebovat 256 průchodů.

```
      LDX #$6000
      LDU #0
      CLRA  ;potřebujeme v A mít 256, což je nula
LOOP: STU ,X++             ; 8 T
      STU ,X++             ; 8 T
      (16x celkem)
      DECA                 ; 2 T
      BNE LOOP             ; 3 T
```

Tato rutina bude vyžadovat 34048 T. Za sekundu ji Kilo stihne vykonat 54krát, což je skoro dvojnásobně lepší výsledek.

Pomocí ABX a indexovaného adresování můžeme napsat ještě rychlejší variantu.

```
      LDX #$6000
      LDU #0

      LDD #$0010 ;potřebujeme v B mít 16
LOOP: STU ,X               ; 5 T
      STU 2,X              ; 6 T
      STU 4,X              ; 6 T
      STU 6,X              ; 6 T
      STU 8,X              ; 6 T
      STU 10,X             ; 6 T
      STU 12,X             ; 6 T
      STU 14,X             ; 6 T
      ABX                  ; 3 T
      STU ,X               ; 5 T
      STU 2,X              ; 6 T
      STU 4,X              ; 6 T
      STU 6,X              ; 6 T
      STU 8,X              ; 6 T
      STU 10,X             ; 6 T
      STU 12,X             ; 6 T
      STU 14,X             ; 6 T
      ABX                  ; 3 T
      DECA                 ; 2 T
      BNE LOOP             ; 3 T
```

Všimněte si, že používáme indexovaný přístup s offsety 0 až 14, což se vejde do postbyte a přinese jen jeden T navíc, a během smyčky dvakrát přičteme B k registru X velmi rychlou instrukcí ABX. V registru A je opět počítadlo průchodů. Tato rutina zabere 26880 T. Za sekundu ji tedy stihneme 68krát.

Ještě o něco rychlejší varianta používá absolutní adresování v rozsahu -16 až +14. Tím srazíme čas na 26368 T.

Nejrychlejší metoda používá instrukci PUSH (PSHU), která potřebuje k uložení dvoubajtové hodnoty 5 + 2n cyklů, kde N je počet ukládaných dvoubajtových hodnot.

```
      LDU #$6000 + $2000 ; od konce
      LDD #0
      LDX #0
      LEAY ,X              ;rychlejší než TFR
LOOP: PSHU D,X,Y           ; 11 T
      PSHU D,X,Y           ; 11 T
      PSHU D,X,Y           ; 11 T

      PSHU D,X,Y           ; 11 T
      PSHU D,X,Y           ; 11 T
      PSHU D               ; 7 T
      CMPU #$6000          ; 5 T
      BHI LOOP             ; 3 T
```

Ve smyčce se opět ukládá 16 x 2 bajty = 32 bajtů. Je použito pět instrukcí PSHU se třemi dvoubajtovými registry (D, X, Y) a jednou PSHU D na doplnění do celkového počtu 16 ukládání.

Když si spočítáme náročnost této smyčky, dojdeme k tomu, že trvá 17920 T. Za sekundu tedy proběhne stodvakrát…

Další možnost optimalizace je využití registru S a ukládání PSHU D, X, Y, S. Tady je potřeba si uložit obsah registru S do paměti a pak jej opět načíst. Máme šanci ušetřit přes 2500 T.

16 Další periferie

16 Další periferie

16.1 Než začneme rozšiřovat…

Následující řádky platí nejen pro Kilo, ale i pro další konstrukce.

Nejjednodušší způsob rozšiřování osmibitového počítače je vyvést kompletní sběrnice – datovou, adresní i řídicí, minimálně signály pro čtení, zápis a přerušení. Důležité je ale pamatovat na to, že pracujeme se signály s megahertzovými frekvencemi, a takové signály potřebují alespoň trochu péče.

Zaprvé: vedení nesmí být příliš dlouhé. Jednotky, maximálně nízké desítky centimetrů.

Zadruhé: cokoli, co se připojuje a odpojuje, představuje riziko. Nejen při samotném připojování a odpojování (to lze vyřešit pravidlem „připojovat a odpojovat pouze při vypnutém napájení"), ale třeba tím, že v případě více periferií nedokáže procesor dát dostatek proudu pro signály. To se projeví nejprve nenápadnými chybami, později tím, že systém přestane pracovat.

Proto je víc než vhodné před rozšiřovací konektory zapojit budiče a posilovače sběrnice. U datové sběrnice nezapomeňte na to, že je potřeba vyřešit její obousměrnost.

Můžete použít například obvod 74HC245, který jsme použili v konstrukci OMEN Alpha. Směr určí signál /RD – pouze pokud procesor čte, tak by měl obvod pouštět informace do systému. Ale pozor – pouze při čtení z oblasti, která není obsazená nějakou vnitřní pamětí či periferií, jinak se data znehodnotí. Druhá možnost je zapojit tento obvod až k samotné periferní desce a využít její dekodér.

U adresní sběrnice můžete využít dva obvody 74HC244 – to jsou osminásobné datové budiče s třístavovým výstupem. Řízení třístavového výstupu nechte buď v log. 0 (na výstupech pak bude stále adresa, kterou posílá procesor), nebo jej připojte k signálu Bus Acknowledge, tedy k tomu, který potvrzuje odpojení procesoru od sběrnice (u 8080 je to HLDA, u Z80 /BUSACK – nutno invertovat, u 6809 je to signál BA).

Vhodné je posílit i řídicí signály a případně hodinový signál. Hodinový signál je pro práci celého systému esenciální, a jeho případné zkreslení či rušení může způsobit opět nefunkčnost, případně chybovost celého systému.

Obecně platí, že s obvody paměti a dvěma periferiemi si nemusíte ještě moc lámat hlavu. Třetí periferie už je na hraně a trošku sázka do loterie, a pokud plánujete čtvrtou a další, už byste měli posílení sběrnice řešit.

Některé periferní obvody jsou konstruované tak, že je lze připojit přímo k procesoru. Kromě standardní výbavy té které procesorové řady (většinou sériový obvod, paralelní, časovač, řadič pře-

rušení, řadič DMA a další) mezi ně patří i další obvody, namátkou hodiny reálného času, displeje nebo disky.

Za chvíli se podíváme na připojení paměťové karty Compact Flash. Tyto paměťové karty mají standardní rozhraní ATA (IDE), a když se podíváte na návrh tohoto rozhraní, zjistíte, že vypadá velmi podobně, jako třeba rozhraní obvodů PIA. Máte datovou sběrnici (u IDE má 16 bitů), několik adresních signálů (u IDE jsou čtyři), signály zápisu a čtení, signál výběru periferie (chip select)…

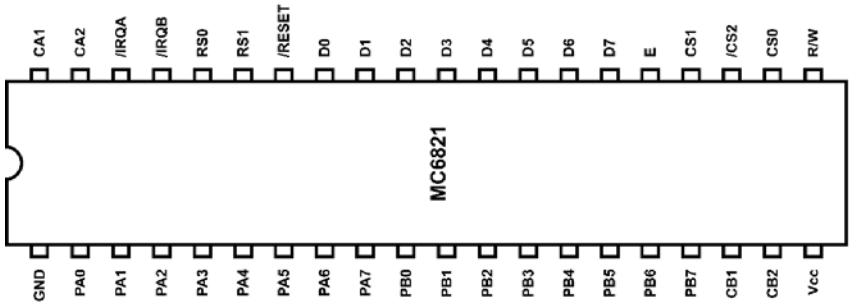
Totéž platí třeba i pro známé znakové LCD displeje 1602, 2004 apod. Tyto displeje mají rovněž osmibitovou datovou sběrnici a řídicí signály a vlastně nám nic nebrání připojit je přímo k procesoru.

Jiné periferie vyžadují speciální rozhraní, a ty je lepší připojit k nějakému dedikovanému komunikačnímu obvodu (VIA, PIA apod.) a ovládat je až pomocí tohoto obvodu.

Někdy může být výhodné připojit i periferie jako jsou displeje nebo zmíněný IDE disk prostřednictvím obvodu paralelního rozhraní (8255, 6820). Ušetříte si práci s posilovačem sběrnice a oddělíte je tím od zbytku systému. Případná chyba periferie tak neovlivní práci celého počítače. Na druhou stranu je práce s takto připojenou periferií o něco pomalejší, což někdy může vadit.

16.2 Paralelní port PIA (6821)

Použili jsme už obvod PPI 8255 (u OMEN Alpha) i VIA 6522 (u Brava). Obvod PIA 6821 (v rodině 65xx má analogický obvod 6521 – liší se pouze označením některých vývodů, např. místo E je PHI2) je jednodušší verze obvodu VIA 6822 / 6522. Obsahuje dva osmibitové paralelní porty s možností nastavit každý bit každého portu nezávisle jako vstup nebo výstup.



Opět vidíme ze strany procesoru standardní rozhraní: datovou sběrnici se signály D0 – D7, dva vstupy pro výběr registrů (RS0 a RS1 – připojte je k adresní sběrnici, třeba na A0 a A1), signály R/W a E, které odpovídají signálům u procesoru 6809, vstup /RESET, tři vybavovací vstupy CS0,

CS1 a /CS2 (obvod reaguje, pokud CS0 = CS1 = 1 a /CS2 = 0) a dva přerušovací výstupy /IRQA a /IRQB. Tyto výstupy jsou v provedení „s otevřeným kolektorem“, tak je můžete bez problémů spojit dohromady a připojit oba ke vstupu /IRQ, /FIRQ nebo /NMI (nezapomeňte na pull up rezistor).

Ze strany portů jsou k dispozici port A (PA0 – PA7), port B (PB0 – PB7) a dva řídicí signály pro každý port (CA1, CA2, CB1, CB2).

Řídicí signály CA1, CB1 slouží jako zdroj požadavků na přerušení. Lze nastavit, jestli reagují na vzestupnou, nebo sestupnou hranu signálu, případně lze přerušení od těchto vstupů zamaskovat.

Řídicí signály CA2, CB2 mohou být naprogramovány buď jako zdroj přerušení, stejně jako v předchozím případě, nebo jako datový výstup. Výstup můžete řídit buď přímo, nebo můžete určit, že se výstup chová jako signalizace „data na portu připravena“. Signály Cx2 mají zapojený interní pull-up rezistor.

Porty A a B nejsou identické. Port A je vybavený pull-up rezistory a při čtení vrací vždy hodnotu, která je na vstupním pinu.

Port B má namísto toho klasický třístavový výstup, kompatibilní s úrovněmi TTL, a může budit vyšší zátěž. Při čtení se chová různě podle toho, jestli je nastaven jako vstup, nebo jako výstup. Pokud je nastaven jako vstup, čte se hodnota z pinu. Když je nastaven na výstup, při čtení z tohoto portu získáte data, která jste si uložili do registru.

Obvod PIA má šest interních registrů, ale přímo adresovatelné jsou jen čtyři (máme dva adresní vstupy RS0 a RS1). Jedná se o řídicí registr (CR), datový registr (PORT) a registr směru toku dat (DDR), všechny ve variantách pro port A a port B: CRA, CRB, PORTA, PORTB, DDRA, DDRB.

Jak návrháři vyřešili problém přístupu k šesti registrům, když jsou adresovatelné jen čtyři? Je to prosté: Na adresách 1 a 3 jsou řídicí registry CRA a CRB, na adresách 0 a 2 jsou datové a směrové registry (PORT/DDR). Bit 2 příslušného řídicího registru (CRA2, CRB2) udává, jestli na adrese 0, resp. 2, je k dispozici datový registr (PORTA, PORTB – když bit 2 = 1), nebo registr řízení směru (DDRA, DDRB – bit 2 = 0).

RS1	RS0	CRA2	CRB2	Registr
0	**0**	1	X	PORTA
0	**0**	0	X	DDRA
0	**1**	X	X	CRA
1	**0**	X	1	PORTB
1	**0**	X	0	DDRB
1	**1**	X	X	CRB

Registry PORT nastavují hodnoty, které jsou poslané na výstupní piny. Při čtení z těchto registrů získáte buď hodnoty na pinech (port A, port B v režimu vstup), nebo hodnoty předtím uložené do registru PORT (port B v režimu výstup).

Každý jednotlivý pin lze nastavit individuálně jako vstupní, nebo jako výstupní, a to zápisem hodnoty 0 nebo 1 do příslušného bitu registru DDR. Hodnota 0 znamená, že se příslušný pin chová jako vstup, hodnota 1 znamená výstup.

Řídicí registr CRx ovládá, kromě už zmíněného přepínání mezi datovým a směrovým registrem, hlavně chování vývodů Cx1, Cx2. Zároveň udržuje dva příznaky pro přerušení.

Bit 0 udává, jestli je povolené přerušení vstupem Cx1. Pokud je 0, je přerušení zakázané.

Bit 1 říká, jestli je přerušení vyvolané sestupnou (0), nebo vzestupnou (1) hranou na vstupu Cx1.

Bit 2 přepíná, jak už bylo zmíněno, přístup k registru DDR (0) nebo k PORT (1).

Bit 5 nastavuje chování vývodu Cx2. Pokud je 0, je Cx2 vstup, pokud je 1, je Cx2 výstup.

Bit 6 oznamuje, že došlo k přerušení od vstupu Cx2. Je automaticky vynulovaný při čtení datového registru, nebo RESETem. Pokud je Cx2 nakonfigurovaný jako výstup, je tento bit 0.

Bit 7 oznamuje, že došlo k přerušení od vstupu Cx1. Vynulovaný je při čtení datového registru a při RESETu.

Vynechal jsem bity 3 a 4. Tyto bity ovládají vývody Cx2 a jejich role se liší podle toho, jestli je tento vývod nakonfigurovaný jako vstup (bit 5 = 0), nebo jako výstup (bit 5 = 1).

Pokud je bit 5 = 0 (Cx2 je vstup), fungují analogicky jako bity 0 a 1. Tedy bit 3 zakazuje (0) nebo povoluje (1) přerušení od vstupu Cx2, bit 4 aktivuje přerušení od sestupné (0) nebo vzestupné (1) hrany.

Pokud je bit 5 = 1, tak můžeme buď přímo ovládat úroveň na tomto výstupu (pokud je bit 4 = 1, tak na výstup Cx2 je zapsána hodnota z bitu 3), nebo nastavit Cx2 jako strobovací signál (bit 4 = 0).

Strobovací mód je asi nejsložitější, navíc se liší u portu A a portu B.

U portu A funguje jako příznak „procesor přečetl data“. Jakmile přečte data z registru PORTA, nastaví se při sestupné hraně vstupu E pin CA2 do log. 0. Zpátky do log. 1 je nastaven podle toho, co je v bitu 3 řídicího registru CRA. Pokud 0, čeká se na aktivní přechod na vstupu CA1 (který přechod je aktivní určuje bit 1 řídicího registru). Pokud je tam 1, čeká se na první sestupnou hranu signálu E.

U portu B funguje jako příznak „data připravena ke čtení“. Jakmile procesor zapíše data do registru PORTB, nastaví se při následující vzestupné hraně vstupu E pin CB2 do log. 0. Zpátky do log. 1 je nastaven podle toho, co je v bitu 3 řídicího registru CRB. Pokud 0, čeká se na aktivní přechod na vstupu CB1. Pokud je tam 1, čeká se na první vzestupnou hranu signálu E.

Já jsem si udělal malý modul, který obsahuje pouze tento obvod a pinové lišty. Pomocí switche přepínám mezi adresami IO1 – IO7. Tento obvod lze docela dobře připojit ke všem dosud probíraným konstrukcím. Doporučuju variantu 68B21, která zvládá vyšší rychlosti (verze 6821 do 1 MHz, 68A21 do 1,5 MHz, 68B21 do 2 MHz).

Díky tomuto modulu tak získávám velmi užitečné rozhraní, na které mohu pohlížet i jako na 16 konfigurovatelných vývodů. Díky nim mohu připojovat moderní periferie s rozhraním SPI (4 vývody) a I^2C (3 vývody)…

16.3 Moderní periferie (SPI)

Několik jsem jich zmínil v Hradlech, voltech. Spousta dalších je ale použitelná i s „novými starými osmibity“. Jako příklad mě napadá třeba SD karta nebo některé displeje s rozhraním SPI.

Staré osmibity bohužel nemají přímo rozhraní SPI, to vzniklo až mnohem později. Dokonce nejsou ani široce dostupné obvody, které by člověk mohl k těmto procesorům připojit a ony by fungovaly jako řadiče sběrnice SPI.

Existuje řešení, implementované v obvodech CPLD (Xilinx XC9532 apod.), popřípadě přímo obvodově, pomocí posuvných registrů a čítačů. Výhodou takového řešení je vysoká přenosová rychlost, klidně i vyšší, než je pracovní rychlost procesoru.

Jednodušší řešení nechává většinu námahy na procesoru a programátorovi. Nejčastěji se použije paralelní port, buď PPI 8255, nebo PIA/VIA, popsané v minulé kapitole, a jeho piny se využijí na vstup a výstup dat. U PIA/VIA je to o něco snazší, u 8255 musíme myslet na to, že jako vstup či výstup může být nastavena vždy celá brána A, B, případně polovina brány C.

Potřebné průběhy signálů pak musíte generovat programem, a to včetně hodinových pulsů SCK. Není to nijak složité, jen to zabere nějaký čas a paměťový prostor.

Zkusím hrubý odhad: na vyslání jednoho bitu bude u procesoru 6809 potřeba cca 16 – 20 taktů hodin. Je potřeba nejprve připravit datový bit a hodinový puls do registru, jeho obsah poslat do PIA, negovat bit s hodinami, opět poslat obsah do PIA a rotovat data. Vyslání jednoho bajtu tak bude trvat 128 – 160 taktů. Počítejme 150. Teoreticky tedy stihneme za jednu sekundu poslat 12 kilobajtů. Čtení bude ještě o něco pomalejší.

Pokud by rozhraní SPI běželo na kmitočtu, rovném hodinovému, stihli byste za sekundu poslat 225 kilobajtů.

Je to málo? Je to moc? Pokud budete přes SPI připojovat třeba externí paměť, tak asi není problém počkat pár sekund na načtení celých 32 kB do paměti RAM. Pokud ale zvažujete připojit přes SPI například displej, tak na nějaké akční divoké hrátky s obrazem spíš zapomeňte – nebo sáhněte po obvodovém řešení.

16.4 Hodiny reálného času (RTC)

Oblíbený obvod DS1307, který znáte z Arduina, jde použít, pokud si postupem z minulé kapitoly implementujete rozhraní I^2C. Je to ale poněkud nepohodlné. Můžeme sáhnout k obvodu, který se používal dřív, a nese označení RTC72421 (výrobce Epson). Tento obvod má rozhraní, kompatibilní s osmibitovými procesory, a navíc má přímo v pouzdru integrovaný krystalový oscilátor.

Jeho zapojení je trochu netradičnější, především proto, že datová sběrnice je pouze čtyřbitová. Obvod RTC se navenek chová jako paměť 16x4 bity. Má tedy čtyři adresní vstupy A0 – A3, čtyři datové obousměrné vývody D0 – D3, má vstupy /RD a /WR, které indikují zápis či čtení, má vstupy /CS0 a CS1 a vstup ALE, známý z procesoru 8085.

Vstup ALE můžete připojit k napájecímu napětí, obvod se pak chová zcela normálně jako každý jiný periferní obvod.

Vstupy /CS0 a CS1 můžete používat tak, jak jste zvyklí, ale nejčastější způsob je ten, že používáte pouze /CS0. CS1 je připojený na napájecí napětí a dokáže indikovat jeho výpadek (vynuluje bity HOLD a RESET).

Kromě výše popsaných signálů nabízí obvod i výstup, nazvaný STD.P (STanDard Pulse). Tento výstup používá otevřený kolektor (je tedy možné ho spojit například s výstupy INT u obvodu PPI). Na tomto výstupu se objevují pulsy s periodou jedné hodiny, jedné minuty, jedné sekundy nebo 1/64 sekundy (lze naprogramovat). V daný čas se STD.P přepne do logické 0, a zpět se vrátí buď po uplynutí 7.8125 ms (výstup s pevnou délkou pulsu), nebo poté, co procesor vynuluje interní příznak IRQF (vynuluje bit 2 registru Dh – režim přerušení).

Obvod pracuje s čísly ve formátu BCD a v registrech 0h – Bh udržuje informace o sekundách (registry 0, 1), minutách (2, 3), hodinách (4, 5), dnech (6, 7), měsících (8, 9) a letech (Ah, Bh). Vždy v nižším registru jsou jednotky, ve vyšším desítky. Registr Ch udržuje informaci o dnu v týdnu.

Z popsaného je vidět, že informace o roku je pouze dvouciferná (00 – 99), což asi nebude až tak velký problém. Obvod umí pracovat s přestupnými roky, ale letní čas za vás nevyřeší.

Registr Dh (řídicí registr D) obsahuje následující bity:

Bit 0 – HOLD. V tomto registru je za normálního provozu 0. pokud potřebujete nastavit nebo číst informace o času, nastavte tento bit na 1. Tím se pozastaví změna času až na jednu sekundu, což je dostatek času k tomu nastavit nebo přečíst obsah registrů 0 – Ch. Pokud po nastavení nevrátíte bit HOLD zpět do nuly, budou hodiny stát.

Bit 1 – BUSY. Pokud chcete zapsat čas, nastavíte HOLD na 1 a čtěte stav bitu BUSY. Měl by být 0. Pokud bude 1, znamená to, že se během operace změnila hodnota sekund. Nyní byste měli zapsat do bitu HOLD opět nulu, počkat alespoň 190 mikrosekund a pak postup opakovat. (BUSY je pouze ke čtení.)

Bit 2 – IRQF. Tento bit je nastaven do log. 1, pokud došlo k události časovače (výše zmíněná změna minut, hodin, sekund, nebo uplynutí intervalu jedné čtyřiašedesátiny sekundy). Pokud používáte režim „interrupt", zapište do tohoto bitu 0, aby se výstup STD.P vrátil zpět do log. 1.

Bit 3 – 30sec ADJ. Tuto funkci pamatujete možná ze starých digitálních hodin. Když jste čekali na časové znamení, mačkali jste průběžně tlačítko ADJUST, které vynulovalo počítadlo sekund. Pokud před stisknutím byla hodnota 30 – 59, přičetla se jednička k počítadlu minut. Přesně tak funguje tento bit. Pokud do něj zapíšete 1, provede se zaokrouhlení času na nejbližší minutu.

Registr Eh obsahuje bity, které řídí výstup STD.P.

Bit 0 – MASK. Zapsáním 1 do tohoto bitu deaktivujete výstup STD.P a ten zůstane stále v log. 1 (přesněji řečeno odpojen).

Bit 1 – ITRPT/STND. Logická 0 zapíná výstup s pevnou periodou (STD.P se vrací k 1 po 1/128 sekundy), logická 1 zapíná mód „přerušení". (STD.P se vrací k 1 po vynulování příznaku IRQF).

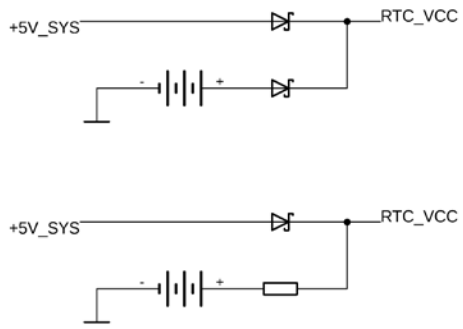
Bity 2 a 3 – Timing. Tyto bity nastavují frekvenci událostí na výstupu STD.P. OO nastavuje periodu 1/64 sekundy, 01 nastavuje periodu 1 sekunda, 10 znamená periodu jedné minuty a 11 je perioda jedné hodiny.

Poslední registr Fh obsahuje bity RESET (bit 0; pokud je 1, nuluje se interní čítač milisekund), STOP (bit 1; pokud je zde log. 1, hodiny stojí), bit 12/24 (bit 2; logická 1 znamená 24hodinový režim) a bit TEST (bit 3; tady musí být vždy 0).

Obvod RTC má smysl hlavně tehdy, pokud oplývá záložní baterií. Bez ní musíte po každém zapnutí počítače nastavovat hodiny znovu, a to není moc příjemné. Návrháři EPSONu s tímto počítali, a tak obvod RTC72421 má sice ve specifikaci napájecí a provozní napětí +5 V, ale k udržení informace o čase stačí pouhé 2 volty. V takovém případě má obvod udávanou maximální spotřebu 5 mikroampérů.

V praxi tedy stačí přivést na napájecí vstup jednak samotné napájecí napětí, ideálně přes Schottkyho diodu, a k němu paralelně přes další Schottkyho diodu připojit třívoltovou baterii.

Pokud máte k dispozici dobíjecí akumulátor NiCd, můžete jej připojit přes rezistor 500 – 800 ohmů a vynechat diodu mezi akumulátorem a napájecím napětím. Oddělovací rezistor je naprosto nezbytný, akumulátory se nesmí dobíjet velkými proudy. Pro NiCd akumulátory platí empirické pravidlo, že nabíjecí proud v mA musí být menší než desetina kapacity akumulátoru v mAh.



Obrázek 57: Zapojení záložní baterie a záložního akumulátoru pro RTC

16.5 Pořádně velké úložiště (CF IDE)

Zatím naše konstrukce umožňovaly komunikaci s obsluhou, nahrání programu po sériové lince, případně načtení zpět do počítače, ale chyběla možnost si svou práci uložit. Chyběl ekvivalent kazetového magnetofonu…

Samozřejmě že by šlo použít kazetový magnetofon, klidně vzít i přímo nějaké rutiny z tehdejších počítačů, šlo by i patřičně zpomalit sériový port a modulovat jeho výstup pro audio, jako to dělal třeba PMD-85, ale ruku na srdce: Vy máte ještě kazeťák a kazety? Jasně že by to bylo *vyntyč* a retro, ale zase poněkud nepraktické.

V posledních letech se objevuje mnoho konstrukcí, které k původním počítačům z 80. let doplňují možnost ukládání na moderní periferní zařízení, většinou paměťové karty typu SD nebo Compact Flash.

SD karta je velmi šikovná periferie. Můžete ji připojit přes standardní rozhraní SPI a nabízí naprosto perfektní poměr cena / kapacita. Daní za to je trochu složitější ovládání, a u konstrukcí s osmibitovými procesory pak nízká rychlost, protože nejsou dost dobře dostupné řadiče SPI a celý přenos pak většinou musí zvládnout samotný procesor pomocí řízení bitů paralelního portu.

Lepší možnosti v tomto směru nabízejí karty Compact Flash. Tyto karty mají standardní rozhraní IDE, které znáte z počítačů PC. Rozhraní IDE je vlastně velmi podobné periferním obvodům, jaké jsme si popisovali v předchozích kapitolách. Má datovou sběrnici, několik adresových vstupů, vstupy Chip Select, Write a Read – takže vlastně na úrovni například obvodu VIA 6522.

Trochu problém může být, že IDE je šestnáctibitové. Buď musíte připojit IDE třeba k obvodu 8255 a využít dva paralelní porty pro data, nebo prostě polovinu kapacity obětovat a použít jen nižší bity datové sběrnice. Ovšem velké množství Compact Flash karet umí pracovat v osmibitovém režimu. Pak je práce s takovou kartou opravdu na úrovni připojení jakéhokoli jiného periferního obvodu.

Lehká nevýhoda je interoperabilita. Pokud chcete využít standardní souborové systémy, jako třeba FAT, aby bylo možné přenášet soubory mezi vaším strojem a standardním počítačem, není to úplně jednoduché. Nikoli neřešitelné, ale poměrně náročné. A standardizované souborové systémy pro „malé soubory“ nejsou.

Různé konstrukce například emulují funkci starých disket pro systém CP/M pomocí souborů – na kartě je standardně zapsán třeba soubor disk000.dta s velikostí rovnou kapacitě tehdejší diskety, a obslužný software překládá přístup k „disketě“ na čtení a zápis v rámci tohoto souboru. Což ovšem předpokládá, že soubor není fragmentovaný.

Další problém je, že paměti, založené na technologii Flash, mají omezený počet cyklů zápis / mazání, a souborové systémy jako je FAT, kde se velmi často přepisuje jedno a totéž místo na disku (alokační tabulka), dokážou tyto cykly velmi brzy vyčerpat. Sofistikovanější média to řeší tím, že na pozadí používají vlastní souborové systémy, ale u jednoduchých CF nebo SD karet, nebo dokonce přímo u paměťových čipů, doporučuji použít jinou organizaci dat.

Pokud u CF nebo SD použijete ryze vlastní organizaci, tedy budete např. ignorovat MBR a rozdělení média na oddíly, přijdete tím o možnost přístupu k datům na PC, protože se karta bude hlásit jako poškozená. Když pro vlastní souborový systém použijete vyhrazený oddíl, nebude sice obsah viditelný, ale zbytek média můžete využít standardním způsobem.

Možnou inspirací mohou být takzvané logové souborové systémy (log-structured file system), jako je třeba YAFFS (Yet Another Flash File System). Tyto systémy pohlížejí na paměť jako na pole sektorů, které postupně plní. Když je potřeba nějakou informaci přepsat (nejen data, ale třeba i velikost souboru apod.), zapíší se nová data na konec aktuálních záznamů a nastaví se jim stejný identifikátor, jako u předchozího záznamu, jen se o 1 zvýší číslo „verze“ (zjednodušeně řečeno). Stará data se pak označí jako neplatná (nastaví se příznak na 0).

Nevýhodou podobného souborového systému je, že software musí procházet jednotlivé sektory a hledat, kde soubory začínají, aby je vůbec mohl vypsat, navíc musí kontrolovat, jestli není někde jinde novější verze téhož sektoru.

- https://z80project.wordpress.com/2015/07/06/z80-8-bit-compact-flash-card-interface-part-1/
- http://www.elektronika.kvalitne.cz/ATMEL/necoteorie/IDEtesty/rizeni_CF_HDD.html
- http://searle.hostei.com/grant/cpm/index.html
- http://www.waveguide.se/?article=8-bit-compact-flash-interface

16.6 Lepší zvuk

Pamatuju se, jak jsem, coby mladý Spectrista, experimentoval se zvukem. Jeden jediný bit, přesné časování, a ono to nakonec hrálo. Bylo to strašlivé a citlivější jedinci si zacpávali uši, ale správný Spectrista v tom škvrčení a pískání poznal i melodii. Kolegové, co měli Atari nebo Commodory, se mohli jen usmívat. Jejich počítače totiž obsahovaly speciální obvody na generování zvuku – u Atari se mu říkalo POKEY, u Commodora to byl známý obvod SID 6581 (později vylepšený 8580).

I Spectristi se dočkali s příchodem ZXS128 – ten obsahoval, rovněž známý, obvod AY-3-8912, což je zmenšená verze čipu AY-3-8910. Tentýž čip, respektive jeho plná verze, se objevoval i v nejrůznějších domácích konstrukcích. Interface „Melodik“ pro Didaktiky obsahoval přesně tento zvukový obvod. Yamaha vyráběla vlastní, mírně vylepšenou variantu pod označením YM2149 – byla použitá například v Atari ST.

SAM Coupé, duchovní nástupce ZX Spectra, obsahoval o něco výkonnější obvod SAA1099 – na rozdíl od tříkanálového AY produkoval SAA1099 šestikanálový zvuk.

K osmibitovým počítačům bylo možné připojit ještě pokročilejší obvody, například OPL2 od Yamahy (YM3812), ale to se, pokud vím, moc nedělo. Obvody OPL2 se objevily až ve zvukových kartách pro PC (AdLib, SoundBlaster).

Vlastní zvukový čip vyráběl i gigant mezi výrobci integrovaných obvodů, Texas Instruments. Pro svůj vlastní počítač TI-99/4A vyvinul čip, později prodávaný jako SN76489. Tento čip byl použit kromě zmíněného TI-99/4A i v některých konzolích, nebo v počítačích Sharp MZ-800 a Sord M-5, známých i u nás. Svými zvukovými schopnostmi odpovídá plus mínus zmíněnému obvodu AY, největší rozdíl je asi ten, že všechny tři zvukové kanály mixuje dohromady už uvnitř pouzdra, takže ven jde pouze jeden monofonní audiosignál.

Který je ten nejlepší?

To je dodneška spolehlivý startér nekonečných debat. Jak se říká: *každý jen tu svou má za jedinou* a je ochoten se za ni, respektive za něj, bít do roztrhání těla, občanského průkazu a pozbytí důstojnosti.

Vynechme novější čipy OPL, které nabízejí FM syntézu, devítikanálový zvuk a podobné vymoženosti. Zaměřme se na obvody, používané v osmibitech.

Zde je nekorunovaným králem asi Commodore SID. Nabízel tři kanály s rozsahem osmi oktáv, z nichž každý mohl používat jeden ze čtyř průběhů (pila, trojúhelník, obdélník a šum). Dále programovatelný filtr, obálky hlasitosti a tři kruhové modulátory. Díky všem těmto vymoženostem byl zvuk SIDu velmi bohatý a tehdejší hudebníci (namátkou: Martin Galway, Rob Hubbard, Chris Hülsbeck nebo Ben Daglish) dokázali z tohoto obvodu vyždímat melodie, které se staly už klasikou. SID zároveň obsahoval A/D převodníky, které se používaly k obsluze herních ovladačů či později myši.

SID má ale zásadní nevýhodu: je jich málo a jsou drahé. Na burzách není problém potkat SIDy za cenu přes 200 Kč. Existují ale náhrady, například SwinSID (který emuluje zvukový generátor pomocí procesoru ATmega) nebo ARMSID.

POKEY od Atari je zákaznický obvod, který měl na starosti nejen zvuk, ale především obsluhu klávesnice, joysticků a sériového rozhraní. Nabízí čtyři zvukové kanály s možností volby průběhu (obdélník nebo šum) a programovatelný filtr (horní propust). Použití POKEY ve vlastních konstrukcích je možné, ale obvod samotný je téměř nesehnatelný.

Philips SAA1099 je o něco novější než předchozí jmenované. Nabízí šest zvukových kanálů a stereofonní výstup. Zvukové generátory opět používají jediný typ průběhu, totiž obdélník. Obvod obsahuje i dva nezávislé generátory šumu. Každý z nich může využít jako „nosnou frekvenci“ (která ovládá „zabarvení“ šumu) buď jednu ze tří předdefinovaných, nebo frekvenci generátoru 0 (resp. 3). Máte k dispozici i dva generátory hlasitostních obálek.

Obvod SAA1099 je docela dobře sehnatelný a jeho cena se pohybuje v rozumném rozpětí 30 – 50 Kč. Je tedy vhodným kandidátem pro použití ve vlastní konstrukci.

Dobře sehnatelný je i SN76489. Tento obvod nabízí, jak už jsem se zmínil, tři zvukové kanály a jeden šumový generátor. Výsledný zvuk je uvnitř obvodu smíchán do jednoho výstupu, což je asi nejzásadnější rozdíl oproti obdobnému čipu AY-3-8910. Druhý rozdíl proti AY je ten, že SN nenabízí hlasitostní obálky.

Konečně AY-3-8910 nabízí tři kanály (se samostatnými výstupy), generátor šumu a generátor obálky. Kromě zvuku obvod obsahuje i dva osmibitové obousměrné paralelní porty, takže jej můžete v konstrukcích použít namísto obvodů 8255 či 6821. Obvod se stále dá sehnat za příznivé ceny, případně jeho ekvivalent YM2149.

Drobná nevýhoda je adresace. Zatímco předchozí obvody nabízely většinou standardní rozhraní, na které jsme zvyklí (/CE, /WE, A0 pro výběr řízení / dat), obvod AY má tři vstupy BDIR, BC1 a BC2. BC2 je navíc nadbytečný, může být připojen na napájecí napětí. Ovládání je pak na vstupech BDIR a BC1.

BDIR	BC1	Funkce
0	0	Neaktivní
0	1	Čtení dat
1	0	Zápis dat
1	1	Nastavení adresy registru

K adresování je pak nutné použít několik hradel, kterými si z obvyklých signálů vygenerujete tuto dvojici.

Obvod AY-3-8910 má svého menšího příbuzného AY-3-8912, který obsahuje pouze jediný paralelní port. Tato verze byla použitá v ZX Spectru 128k. Dnes se o něco obtížněji shání a je výrazně dražší.

Ještě menší varianta AY-3-8913, která neměla žádný paralelní port, se moc nepoužívala a dnes je téměř k nesehnání.

I pro AY existuje emulátor – hledejte klíčové slovo „avray“…

Pro vlastní konstrukci bych doporučil obvody SN76489 (jsou levné a jednoduché), popřípadě SAA1099 (jsou v dané kategorii asi nejvyspělejší). Obvody AY jsou sice dobře dostupné a za rozumnou cenu, ale tam je výrazným negativem jejich podivné adresování. Obvody SID či POKEY jsou buď nesehnatelné, nebo extrémně drahé. Z novější generace bych doporučil k experimentům obvody od Yamahy, jako OPL2, zjednodušený OPLL (YM2413) nebo OPN2 (YM2612).

Jak vlastně hrají?

Ačkoli se od sebe navzájem jednotlivé obvody výrazně liší, základní principy práce mají podobné.

Všechny jmenované obvody fungují tak, že mají několik vnitřních registrů (ty jednodušší třeba 8, 16, komplexní až 256), z nichž každý řídí některý parametr generování zvuku. Aby návrháři ušetřili adresní vstupy a zjednodušili zapojení, používá se u těchto obvodů dvoufázový přístup. V první fázi se do obvodu zapíše adresa registru, s nímž chcete pracovat, a v druhé fázi se z obvodu čte, nebo se do něj zapisuje.

V konkrétních detailech se obvody liší – většina má nějaký vstup, kterým odlišuje data a adresu, obvod SN takový vstup nemá a používá sedmibitová data, osmý bit rozhoduje, zda jde o data, nebo o data a adresu registru.

Každý z těchto obvodů má vstup, k němuž je zapotřebí připojit nějaký vnější generátor kmitočtu – většinou v řádech MHz. Uvnitř obvodu je tento kmitočet předděličkou snížen na kmitočet v řádech desítek kHz. Z tohoto kmitočtu se generují jednotlivé tóny.

Základním tvarem zvukových průběhů je u probíraných obvodů obdélník. Vzniká jednoduše podělením hlavní frekvence nějakou konstantou, která je uložena ve vnitřních registrech.

Většinou bývá tato konstanta deseti- či dvanáctibitová. Výsledná frekvence vzniká podělením pracovní frekvence hodnotami 1 až 1024 (popř. až 4096 u dvanáctibitových děličů).

Každý kanál má vlastní registr pro tuto hodnotu, která udává přímo frekvenci generovaného zvuku.

Dále mívá každý kanál vlastní registr pro hlasitost. Většinou bývá čtyřbitová, v rozsahu 0 – 15.

Obvody mívají i generátor šumu pro vytváření perkusních zvuků (činely apod.) – k tomuto generátoru bývá často možnost měnit charakteristiku šumu („frekvenci“).

Důležitý registr je takzvaný směšovač, který určuje, jestli v tom kterém kanálu zní generovaný tón, nebo šum.

U většiny obvodů najdeme i takzvaný generátor obálky. Obálka hlasitosti (volume envelopment) je křivka průběhu hlasitosti, která výrazně ovlivňuje charakter zvuku. Někdy se označuje též ADSR podle svých základních prvků.

Obecně lze u zvuků pozorovat čtyři fáze, které popisujeme jako náběh (Attack), pokles (Decay), výdrž (Sustain) a doznívání (Release). Například při úderu kladívka do struny v pianu se struna nejprve prudce rozezní a pak její hlasitost poklesne na standardní hodnotu. Tam drží, dokud je klávesa stisknuta. Jakmile klávesu pustíte, tón se ztrácí, ale ještě krátkou dobu doznívá.

Každý druh nástroje má vlastní charakteristiku. Některé mají výše popsanou, některé mají náběh pomalý, některé mají dlouhé doznívání, některé mají téměř okamžitý náběh, rychlý pokles a výdrž nulovou…

Zvukové generátory mívají generátor takové obálky, který můžete zapnout pro konkrétní kanál. Parametry ADSR se většinou udávají jako čtyřbitová čísla, kde A, D a R je čas náběhu nebo doběhu a S je úroveň hlasitosti. Většina generátorů ale nemá až takové možnosti nastavení a nabízí jen možnost vybrat předpřipravený tvar a určit jeho rychlost.

Valčík na rozloučenou

Máte svůj vlastní zvukový generátor, připojený ke svému vlastnímu počítači. Reprobedny fungují. Přeložili jste si vlastní obslužný program, který nastaví hodnoty do registrů. Spouštíte – a zní tón!

Od tónu k melodii je ale ještě dlouhá cesta. V melodii se tóny různě střídají a každý má jinou dobu trvání. K tomu, abyste hráli opravdu hudbu, musíte mít především rytmus. Musíte odpočítat, že tento tón má hrát určitou dobu a tento tón dvojnásobnou – jinak se vám celá melodie rytmicky rozsype. Jak toho dosáhnout?

Buď použijete stejný postup, jako u jednobitové hudby, totiž že si časovacími smyčkami odměřujete čas a počítač během hraní nedělá nic jiného, nebo využijete periodické přerušení. Ve výše uvedených konstrukcích bohužel nemáme nic, co by šlo využít jako zdroj periodického přerušení. U domácích počítačů s výstupem na televizi se často používalo přerušení, svázané s generováním obrazu, třeba s frekvencí 50 Hz. Přehrávání hudby je pak jednoduché – každou padesátinu sekundy se zavolá rutina, která zkontroluje, zda je potřeba zahrát další tón, a pokud ano, nastaví příslušné registry.

U našich počítačů můžeme využít třeba hodiny reálného času – zmíněný RTC od firmy EPSON umí generovat pulsy s frekvencí 64 Hz. Popřípadě můžeme podělit pracovní frekvenci procesoru nějakým vhodným čítačem – třeba čtrnáctibitovým 74HC4020 (CMOS varianta CD4020). Při pracovní frekvenci 1,8432 MHz získáme podělením tímto obvodem frekvenci 112,5 Hz.

Pro všechny výše zmiňované zvukové čipy existuje spousta studijního materiálu na internetu. Od ukázkových melodií po zajímavé zvukové rutiny pro ovládání toho kterého čipu.

16.7 A co dál?

Dál už to je, milí čtenáři, na vás. Na vaší fantazii a chuti ponořit se do bohatého světa programování a mikroelektroniky ve stylu osmdesátých let.

Pokud vás tato cesta zaujala, tak vás zvu k četbě dodatků, v nichž najdete ještě hlubší ponor do světa mikroprocesorů a syntézy elektronických obvodů prostřednictvím moderních součástek FPGA.

Já samozřejmě pracuju dál na uvedených konstrukcích a připravuju další periferie i programové vybavení. Vše naleznete na stránkách této knihy, kam vás tímto srdečně zvu.

https://8bt.cz

Doslov

Doslov

Když jsem dopisoval předchozí knihu (Hradla, volty, jednočipy), do závěru jsem psal, že jsem si spoustu témat nechal na případné pokračování. To jste právě dočetli.

Díky nabídce od Edice CZ.NIC jsem tak mohl splnit svůj sen, o kterém jsem se zmiňoval někdy od roku 2015, tedy napsat knihu, která by shrnula číslicovou a procesorovou techniku ve formě dostupné pro domácí bastlení. Nakonec z toho byly knihy dvě, plus dodatky, ale to nevadí.

Snažil jsem se popsat a předat vše, co jsem se o procesorech a číslicové technice naučil od roku 1982, kdy jsem poprvé otevřel Amatérské rádio a v něm viděl schéma počítače. Byl jsem jako očarovaný – a to jsem tehdy nemohl tušit, že očarování vydrží dalších pětatřicet let.

Úmyslně jsem vynechal modernější technologii – tedy až na Arduino. Modernější procesory si totiž tak úplně neosaháte, na koleni si z nich nic nepostavíte, a pro výuku mají jeden podstatný handicap. Jasně, pro praxi je moc fajn, když víte, jak pracuje Core i7 a jak se programuje v assembleru. Ale začínat s učením assembleru přímo u Core nebo u ARMu je problém. Buď vás bude stát obrovské úsilí pochopit, proč jsou některé věci tak, jak jsou, nebo na pochopení rezignujete a budete umět programovat na uživatelské úrovni.

Proto jsem si řekl, že limit jsou osmibitové procesory, používané hlavně v 80. letech. Zvolil jsem takové, které jsou dodnes k sehnání. Pro mladší jsou skvělé na učení, pro starší navíc zafunguje i trocha nostalgie.

Ostatně právě vzpomínky lidí z mé generace na jejich první domácí počítače byly docela hezkou motivací pro napsání obou knih. „To byly skvělé časy u Spectra. Já vždycky chtěl vědět, jak to uvnitř funguje…“ Doufám, že se mi podařilo trochu přiblížit, jak to uvnitř funguje, a především posílit chuť si něco takového zkusit vytvořit. Vždyť to přeci není vůbec těžké!

Opravdu.

A rozhodně to není ani zbytečné.

— Doslov

Přílohy

Přílohy

Instrukce procesoru 8085

Opcode	Instrukce	#	T	Příznaky	Funkce
0x00	NOP	1	4		
0x01	LXI B,D16	3	10		B = byte 3, C = byte 2
0x02	STAX B	1	7		(BC) = A
0x03	INX B	1	6	--K----	BC = BC+1
0x04	INR B	1	4	SZKAPV-	B = B+1
0x05	DCR B	1	4	SZKAPV-	B = B-1
0x06	MVI B, D8	2	7		B = byte 2
0x07	RLC	1	4	-----VC	A = A << 1; bit 0 = prev bit 7; CY = prev bit 7
0x08	*DSUB	1	10	SZKAPVC	HL = HL - BC
0x09	DAD B	1	10	-----VC	HL = HL + BC
0x0a	LDAX B	1	7	--K----	A = (BC)
0x0b	DCX B	1	6		BC = BC-1
0x0c	INR C	1	4	SZKAPV-	C = C+1
0x0d	DCR C	1	4	SZKAPV-	C =C-1
0x0e	MVI C,D8	2	7		C = byte 2
0x0f	RRC	1	4	-----0C	A = A >> 1; bit 7 = prev bit 0; CY = prev bit 0
0x10	*ARHL	1	7	-----0C	CY = L bit 0; HL = HL >> 1
0x11	LXI D,D16	3	10		D = byte 3, E = byte 2
0x12	STAX D	1	7		(DE) = A
0x13	INX D	1	6		DE = DE + 1
0x14	INR D	1	4	SZKAPV-	D = D+1
0x15	DCR D	1	4	SZKAPV-	D = D-1
0x16	MVI D, D8	2	7		D = byte 2
0x17	RAL	1	4	-----VC	A = A << 1; bit 0 = prev CY; CY = prev bit 7
0x18	*RDEL	1	10	-----VC	CY = D bit 7; DE = DE << 1
0x19	DAD D	1	10	-----VC	HL = HL + DE
0x1a	LDAX D	1	7		A = (DE)
0x1b	DCX D	1	6	--K----	DE = DE-1
0x1c	INR E	1	4	SZKAPV-	E = E+1
0x1d	DCR E	1	4	SZKAPV-	E = E-1

Opcode	Instrukce	#	T	Příznaky	Funkce
0x1e	MVI E,D8	2	7		E = byte 2
0x1f	RAR	1	4	-----0C	A = A >> 1; bit 7 = prev bit CY; CY = prev bit 0
0x20	RIM	1	4		Čtení SID a přerušení
0x21	LXI H,D16	3	10		H = byte 3, L = byte 2
0x22	SHLD adr	3	16		(adr) =L; (adr+1)=H
0x23	INX H	1	6	--K----	HL = HL + 1
0x24	INR H	1	4	SZKAPV-	H = H+1
0x25	DCR H	1	4	SZKAPV-	H = H-1
0x26	MVI H,D8	2	7		H = byte 2
0x27	DAA	1	4	SZKAPVC	special
0x28	*LDHI D8	2	10		DE = HL+d8
0x29	DAD H	1	10	-----VC	HL = HL + HL
0x2a	LHLD adr	3	16		L = (adr); H=(adr+1)
0x2b	DCX H	1	6	--K----	HL = HL-1
0x2c	INR L	1	4	SZKAPV-	L = L+1
0x2d	DCR L	1	4	SZKAPV-	L = L-1
0x2e	MVI L, D8	2	7		L = byte 2
0x2f	CMA	1	4		A = !A
0x30	SIM	1	4		Zápis přerušovací masky a SOD
0x31	LXI SP, D16	3	10		SP.hi = byte 3, SP.lo = byte 2
0x32	STA adr	3	13		(adr) = A
0x33	INX SP	1	6	--K----	SP = SP + 1
0x34	INR M	1	10	SZKAPV-	(HL) = (HL)+1
0x35	DCR M	1	10	SZKAPV-	(HL) = (HL)-1
0x36	MVI M,D8	2	10		(HL) = byte 2
0x37	STC	1	4	------1	CY = 1
0x38	*LDSI D8	2	10		DE = SP + d8
0x39	DAD SP	1	10	-----VC	HL = HL + SP
0x3a	LDA adr	3	13		A = (adr)
0x3b	DCX SP	1	6	--K----	SP = SP-1
0x3c	INR A	1	4	SZKAPV-	A = A+1
0x3d	DCR A	1	4	SZKAPV-	A = A-1
0x3e	MVI A,D8	2	7		A = d8
0x3f	CMC	1	4	------C	CY=!CY

Opcode	Instrukce	#	T	Příznaky	Funkce
0x40	MOV B,B	1	4		B = B
0x41	MOV B,C	1	4		B = C
0x42	MOV B,D	1	4		B = D
0x43	MOV B,E	1	4		B = E
0x44	MOV B,H	1	4		B = H
0x45	MOV B,L	1	4		B = L
0x46	MOV B,M	1	7		B = (HL)
0x47	MOV B,A	1	4		B = A
0x48	MOV C,B	1	4		C = B
0x49	MOV C,C	1	4		C = C
0x4a	MOV C,D	1	4		C = D
0x4b	MOV C,E	1	4		C = E
0x4c	MOV C,H	1	4		C = H
0x4d	MOV C,L	1	4		C = L
0x4e	MOV C,M	1	7		C = (HL)
0x4f	MOV C,A	1	4		C = A
0x50	MOV D,B	1	4		D = B
0x51	MOV D,C	1	4		D = C
0x52	MOV D,D	1	4		D = D
0x53	MOV D,E	1	4		D = E
0x54	MOV D,H	1	4		D = H
0x55	MOV D,L	1	4		D = L
0x56	MOV D,M	1	7		D = (HL)
0x57	MOV D,A	1	4		D = A
0x58	MOV E,B	1	4		E = B
0x59	MOV E,C	1	4		E = C
0x5a	MOV E,D	1	4		E = D
0x5b	MOV E,E	1	4		E = E
0x5c	MOV E,H	1	4		E = H
0x5d	MOV E,L	1	4		E = L
0x5e	MOV E,M	1	7		E = (HL)
0x5f	MOV E,A	1	4		E = A
0x60	MOV H,B	1	4		H = B
0x61	MOV H,C	1	4		H = C

Opcode	Instrukce	#	T	Příznaky	Funkce
0x62	MOV H,D	1	4		H = D
0x63	MOV H,E	1	4		H = E
0x64	MOV H,H	1	4		H = H
0x65	MOV H,L	1	4		H = L
0x66	MOV H,M	1	7		H = (HL)
0x67	MOV H,A	1	4		H = A
0x68	MOV L,B	1	4		L = B
0x69	MOV L,C	1	4		L = C
0x6a	MOV L,D	1	4		L = D
0x6b	MOV L,E	1	4		L = E
0x6c	MOV L,H	1	4		L = H
0x6d	MOV L,L	1	4		L = L
0x6e	MOV L,M	1	7		L = (HL)
0x6f	MOV L,A	1	4		L = A
0x70	MOV M,B	1	7		(HL) = B
0x71	MOV M,C	1	7		(HL) = C
0x72	MOV M,D	1	7		(HL) = D
0x73	MOV M,E	1	7		(HL) = E
0x74	MOV M,H	1	7		(HL) = H
0x75	MOV M,L	1	7		(HL) = L
0x76	HLT	1	5		special
0x77	MOV M,A	1	7		(HL) = A
0x78	MOV A,B	1	4		A = B
0x79	MOV A,C	1	4		A = C
0x7a	MOV A,D	1	4		A = D
0x7b	MOV A,E	1	4		A = E
0x7c	MOV A,H	1	4		A = H
0x7d	MOV A,L	1	4		A = L
0x7e	MOV A,M	1	7		A = (HL)
0x7f	MOV A,A	1	4		A = A
0x80	ADD B	1	4	SZKAPVC	A = A + B
0x81	ADD C	1	4	SZKAPVC	A = A + C
0x82	ADD D	1	4	SZKAPVC	A = A + D
0x83	ADD E	1	4	SZKAPVC	A = A + E

Opcode	Instrukce	#	T	Příznaky	Funkce
0x84	ADD H	1	4	SZKAPVC	A = A + H
0x85	ADD L	1	4	SZKAPVC	A = A + L
0x86	ADD M	1	7	SZKAPVC	A = A + (HL)
0x87	ADD A	1	4	SZKAPVC	A = A + A
0x88	ADC B	1	4	SZKAPVC	A = A + B + CY
0x89	ADC C	1	4	SZKAPVC	A = A + C + CY
0x8a	ADC D	1	4	SZKAPVC	A = A + D + CY
0x8b	ADC E	1	4	SZKAPVC	A = A + E + CY
0x8c	ADC H	1	4	SZKAPVC	A = A + H + CY
0x8d	ADC L	1	4	SZKAPVC	A = A + L + CY
0x8e	ADC M	1	7	SZKAPVC	A = A + (HL) + CY
0x8f	ADC A	1	4	SZKAPVC	A = A + A + CY
0x90	SUB B	1	4	SZKAPVC	A = A - B
0x91	SUB C	1	4	SZKAPVC	A = A - C
0x92	SUB D	1	4	SZKAPVC	A = A - D
0x93	SUB E	1	4	SZKAPVC	A = A - E
0x94	SUB H	1	4	SZKAPVC	A = A - H
0x95	SUB L	1	4	SZKAPVC	A = A - L
0x96	SUB M	1	7	SZKAPVC	A = A - (HL)
0x97	SUB A	1	4	SZKAPVC	A = A - A
0x98	SBB B	1	4	SZKAPVC	A = A - B - CY
0x99	SBB C	1	4	SZKAPVC	A = A - C - CY
0x9a	SBB D	1	4	SZKAPVC	A = A - D - CY
0x9b	SBB E	1	4	SZKAPVC	A = A - E - CY
0x9c	SBB H	1	4	SZKAPVC	A = A - H - CY
0x9d	SBB L	1	4	SZKAPVC	A = A - L - CY
0x9e	SBB M	1	7	SZKAPVC	A = A - (HL) - CY
0x9f	SBB A	1	4	SZKAPVC	A = A - A - CY
0xa0	ANA B	1	4	SZKAPVC	A = A & B
0xa1	ANA C	1	4	SZKAPVC	A = A & C
0xa2	ANA D	1	4	SZKAPVC	A = A & D
0xa3	ANA E	1	4	SZKAPVC	A = A & E
0xa4	ANA H	1	4	SZKAPVC	A = A & H
0xa5	ANA L	1	4	SZKAPVC	A = A & L

Opcode	Instrukce	#	T	Příznaky	Funkce
0xa6	ANA M	1	7	SZKAPVC	A = A & (HL)
0xa7	ANA A	1	4	SZKAPVC	A = A & A
0xa8	XRA B	1	4	SZKAPVC	A = A ^ B
0xa9	XRA C	1	4	SZKAPVC	A = A ^ C
0xaa	XRA D	1	4	SZKAPVC	A = A ^ D
0xab	XRA E	1	4	SZKAPVC	A = A ^ E
0xac	XRA H	1	4	SZKAPVC	A = A ^ H
0xad	XRA L	1	4	SZKAPVC	A = A ^ L
0xae	XRA M	1	7	SZKAPVC	A = A ^ (HL)
0xaf	XRA A	1	4	SZKAPVC	A = A ^ A
0xb0	ORA B	1	4	SZKAPVC	A = A | B
0xb1	ORA C	1	4	SZKAPVC	A = A | C
0xb2	ORA D	1	4	SZKAPVC	A = A | D
0xb3	ORA E	1	4	SZKAPVC	A = A | E
0xb4	ORA H	1	4	SZKAPVC	A = A | H
0xb5	ORA L	1	4	SZKAPVC	A = A | L
0xb6	ORA M	1	7	SZKAPVC	A = A | (HL)
0xb7	ORA A	1	4	SZKAPVC	A = A | A
0xb8	CMP B	1	4	SZKAPVC	A - B
0xb9	CMP C	1	4	SZKAPVC	A - C
0xba	CMP D	1	4	SZKAPVC	A - D
0xbb	CMP E	1	4	SZKAPVC	A - E
0xbc	CMP H	1	4	SZKAPVC	A - H
0xbd	CMP L	1	4	SZKAPVC	A - L
0xbe	CMP M	1	7	SZKAPVC	A - (HL)
0xbf	CMP A	1	4	SZKAPVC	A - A
0xc0	RNZ	1	12/6		if NZ, RET
0xc1	POP B	1	10		C = (SP); B = (SP+1); SP = SP+2
0xc2	JNZ adr	3	10/7		if NZ, PC = adr
0xc3	JMP adr	3	10		PC <= adr
0xc4	CNZ adr	3	18/9		if NZ, CALL adr
0xc5	PUSH B	1	12		(SP-2)=C; (SP-1)=B; SP = SP - 2
0xc6	ADI D8	2	7	SZKAPVC	A = A + byte
0xc7	RST 0	1	12		CALL $0

Opcode	Instrukce	#	T	Příznaky	Funkce
0xc8	RZ	1	12/6		if Z, RET
0xc9	RET	1	10		PC.lo = (SP); PC.hi=(SP+1); SP = SP+2
0xca	JZ adr	3	10/7		if Z, PC = adr
0xcb	*RSTV	1	12/6		CALL $40 if V=1
0xcc	CZ adr	3	18/9		if Z, CALL adr
0xcd	CALL adr	3	18		(SP-1)=PC.hi;(SP-2)=PC.lo;SP=SP+2;PC=adr
0xce	ACI D8	2	7	SZKAPVC	A = A + data + CY
0xcf	RST 1	1	12		CALL $8
0xd0	RNC	1	12/6		if NCY, RET
0xd1	POP D	1	10		E = (SP); D = (SP+1); SP = SP+2
0xd2	JNC adr	3	10/7		if NCY, PC=adr
0xd3	OUT D8	2	10		special
0xd4	CNC adr	3	18/9		if NCY, CALL adr
0xd5	PUSH D	1	12		(SP-2)=E; (SP-1)=D; SP = SP - 2
0xd6	SUI D8	2	7	SZKAPVC	A = A - data
0xd7	RST 2	1	12		CALL $10
0xd8	RC	1	12/6		if CY, RET
0xd9	*SHLX	1	10		(DE) = L, (DE+1) = H
0xda	JC adr	3	10/7		if CY, PC=adr
0xdb	IN D8	2	10		special
0xdc	CC adr	3	18/9		if CY, CALL adr
0xdd	*JNK adr	3	10/7		if not K PC = adr
0xde	SBI D8	2	7	SZKAPVC	A = A - data – CY
0xdf	RST 3	1	12		CALL $18
0xe0	RPO	1	12/6		if PO, RET
0xe1	POP H	1	10		L = (SP); H = (SP+1); SP = SP+2
0xe2	JPO adr	3	10/7		if PO, PC = adr
0xe3	XTHL	1	16		L <=> (SP); H <=> (SP+1)
0xe4	CPO adr	3	18/9		if PO, CALL adr
0xe5	PUSH H	1	12		(SP-2)=L; (SP-1)=H; SP = SP - 2
0xe6	ANI D8	2	7	SZKAPVC	A = A & data
0xe7	RST 4	1	12		CALL $20
0xe8	RPE	1	12/6		if PE, RET
0xe9	PCHL	1	6		PC.hi = H; PC.lo = L

Opcode	Instrukce	#	T	Příznaky	Funkce
0xea	JPE adr	3	10/7		if PE, PC = adr
0xeb	XCHG	1	4		H <-> D; L <-> E
0xec	CPE adr	3	18/9		if PE, CALL adr
0xed	*LHLX	1	10		L = (DE), H = (DE+1)
0xee	XRI D8	2	7	SZKAPVC	A = A ^ data
0xef	RST 5	1	12		CALL $28
0xf0	RP	1	12/6		if P, RET
0xf1	POP PSW	1	10	SZKAPVC	flags = (SP); A = (SP+1); SP = SP+2
0xf2	JP adr	3	10/7		if P, PC = adr
0xf3	DI	1	4		special
0xf4	CP adr	3	18/9		if P, PC = adr
0xf5	PUSH PSW	1	12		(SP-2)=flags; (SP-1)=A; SP = SP – 2
0xf6	ORI D8	2	7	SZKAPVC	A = A | data
0xf7	RST 6	1	12		CALL $30
0xf8	RM	1	12/6		if M, RET
0xf9	SPHL	1	6		SP=HL
0xfa	JM adr	3	10/7		if M, PC = adr
0xfb	EI	1	4		special
0xfc	CM adr	3	18/9		if M, CALL adr
0xfd	*JK adr	3	10/7		if K PC = adr
0xfe	CPI D8	2	7	SZKAPVC	A – data
0xff	RST 7	1	12		CALL $38

Instrukce procesoru 65C02

Opcode	Instrukce	#	T	Mód
00	BRK	1	7	
01	ORA	2	6	izx
02	-	2	2	imm
03	-	1	1	
04	TSB	2	5	zpg
05	ORA	2	3	zpg
06	ASL	2	5	zpg
07	RMB0	2	5	zpg
08	PHP	1	3	
09	ORA	2	2	imm
0A	ASL	1	2	
0B	-	1	1	
0C	TSB	3	6	abs
0D	ORA	3	4	abs
0E	ASL	3	6	abs
0F	BBR0	3	5	zpr
10	BPL	2	2*	rel
11	ORA	2	5*	izy
12	ORA	2	5	izp
13	-	1	1	
14	TRB	2	5	zpg
15	ORA	2	4	zpx
16	ASL	2	6	zpx
17	RMB1	2	5	zpg
18	CLC	1	2	
19	ORA	3	4*	aby
1A	INC	1	2	
1B	-	1	1	aby
1C	TRB	3	6	abs
1D	ORA	3	4*	abx
1E	ASL	3	6*	abx
1F	BBR1	3	5	zpr

Opcode	Instrukce	#	T	Mód
20	JSR	3	6	abs
21	AND	2	6	izx
22	-	2	2	imm
23	-	2	1	
24	BIT	2	3	zpg
25	AND	2	3	zpg
26	ROL	2	5	zpg
27	RMB2	2	5	zpg
28	PLP	1	4	
29	AND	2	2	imm
2A	ROL	1	2	
2B	-	1	1	
2C	BIT	3	4	abs
2D	AND	3	4	abs
2E	ROL	3	6	abs
2F	BBR2	3	5	zpr
30	BMI	2	2*	rel
31	AND	2	5*	izy
32	AND	2	5	izp
33	-	1	1	
34	BIT	2	4	zpx
35	AND	2	4	zpx
36	ROL	2	6	zpx
37	RMB3	2	5	zpx
38	SEC	1	2	
39	AND	3	4*	aby
3A	DEC	1	2	
3B	-	1	1	
3C	BIT	3	4*	abx
3D	AND	3	4*	abx
3E	ROL	3	6*	abx
3F	BBR3	3	5	zpr

Opcode	Instrukce	#	T	Mód
40	RTI	1	6	
41	EOR	2	6	izx
42	-	2	2	imm
43	-	1	1	
44	-	2	3	zpg
45	EOR	2	3	zpg
46	LSR	2	5	zpg
47	RMB4	2	5	zpg
48	PHA	1	3	
49	EOR	2	2	imm
4A	LSR	1	2	
4B	-	1	1	
4C	JMP	3	3	abs
4D	EOR	3	4	abs
4E	LSR	3	6	abs
4F	BBR4	3	5	zpr
50	BVC	2	2*	rel
51	EOR	2	5*	izy
52	EOR	2	5	izp
53	-	1	1	
54	-	2	4	zpx
55	EOR	2	4	zpx
56	LSR	2	6	zpx
57	RMB5	2	5	zpg
58	CLI	1	2	
59	EOR	3	4*	aby
5A	PHY	1	3	
5B	-	1	1	
5C	-	3	8	abs
5D	EOR	3	4*	abx
5E	LSR	3	6*	abx
5F	BBR5	3	5	zpr
60	RTS	1	6	
61	ADC	2	6	izx

Opcode	Instrukce	#	T	Mód
62	-	2	2	imm
63	-	1	1	
64	STX	2	3	zpx
65	ADC	2	3	zpg
66	ROR	2	5	zpg
67	RMB6	2	5	zpg
68	PLA	1	4	imp
69	ADC	2	2	imm
6A	ROR	1	2	ima
6B	-	1	1	
6C	JMP	3	6	ind
6D	ADC	3	4	abs
6E	ROR	3	6	abs
6F	BBR6	3	5	zpr
70	BVS	2	2*	rel
71	ADC	2	5*	izy
72	ADC	2	5	izp
73	-	1	1	
74	STZ	2	4	zpx
75	ADC	2	4	zpx
76	ROR	2	6	zpx
77	RMB7	2	5	zpg
78	SEI	1	2	
79	ADC	3	4*	aby
7A	PLY	1	4	
7B	-	1	1	
7C	JMP	3	6	iax
7D	ADC	3	4*	abx
7E	ROR	3	6*	abx
7F	BBR7	3	5	abx
80	BRA	2	3*	rel
81	STA	2	6	izx
82	-	2	2	imm
83	-	1	1	

Opcode	Instrukce	#	T	Mód
84	STY	2	3	zpg
85	STA	2	3	zpg
86	STX	2	3	zpg
87	SMB0	2	5	zpg
88	DEY	1	2	imp
89	BIT	2	2	imm
8A	TXA	1	2	imp
8B	-	1	1	
8C	STY	3	4	abs
8D	STA	3	4	abs
8E	STX	3	4	abs
8F	BBS0	3	5	zpr
90	BCC	2	2*	rel
91	STA	2	6	izy
92	STA	2	5	izp
93	-	1	1	
94	STY	2	4	zpx
95	STA	2	4	zpx
96	STX	2	4	zpy
97	SMB1	2	5	zpg
98	TYA	1	2	
99	STA	3	5	aby
9A	TXS	1	2	
9B	-	1	1	
9C	STZ	3	4	abs
9D	STA	3	5	abx
9E	STZ	3	5	abx
9F	BBS1	3	5	zpr
A0	LDY	2	2	imm
A1	LDA	2	6	izx
A2	LDX	2	2	imm
A3	-	1	1	
A4	LDY	2	3	zpg
A5	LDA	2	3	zpg

Opcode	Instrukce	#	T	Mód
A6	LDX	2	3	zpg
A7	SMB2	2	5	zpg
A8	TAY	1	2	
A9	LDA	2	2	imm
AA	TAX	1	2	
AB	-	1	1	
AC	LDY	3	4	abs
AD	LDA	3	4	abs
AE	LDX	3	4	abs
AF	BBS2	3	5	zpr
B0	BCS	2	2*	rel
B1	LDA	2	5*	izy
B2	LDA	2	5	izp
B3	-	1	1	
B4	LDY	2	4	zpx
B5	LDA	2	4	zpx
B6	LDX	2	4	zpy
B7	SMB3	2	5	zpg
B8	CLV	1	2	
B9	LDA	3	4*	aby
BA	TSX	1	2	imp
BB	-	1	1	
BC	LDY	3	4*	abx
BD	LDA	3	4*	abx
BE	LDX	3	4*	aby
BF	BBS3	3	5	zpr
C0	CPY	2	2	imm
C1	CMP	2	6	izx
C2	-	2	2	imm
C3	-	2	1	izx
C4	CPY	2	3	zpg
C5	CMP	2	3	zpg
C6	DEC	2	5	zpg
C7	SMB4	2	5	zpg

Opcode	Instrukce	#	T	Mód
C8	INY	1	2	
C9	CMP	2	2	imm
CA	DEX	1	2	
CB	WAI	1	3	
CC	CPY	3	4	abs
CD	CMP	3	4	abs
CE	DEC	3	6	abs
CF	BBS4	3	5	zpr
D0	BNE	2	2*	rel
D1	CMP	2	5*	izy
D2	CMP	2	5	izp
D3	-	1	1	
D4	-	2	4	zpx
D5	CMP	2	4	zpx
D6	DEC	2	6	zpx
D7	SMB5	2	5	zpg
D8	CLD	1	2	
D9	CMP	3	4*	aby
DA	PHX	1	3	
DB	STP	1	3	
DC	-	3	4	abs
DD	CMP	3	4*	abx
DE	DEC	3	7	abx
DF	BBS5	3	5	zpr
E0	CPX	2	2	imm
E1	SBC	2	6	izx
E2	-	2	2	imm
E3	-	1	1	
E4	CPX	2	3	zpg
E5	SBC	2	3	zpg
E6	INC	2	5	zpg
E7	SMB6	2	5	zpg
E8	INX	1	2	
E9	SBC	2	2	imm

Opcode	Instrukce	#	T	Mód
EA	NOP	1	2	
EB	-	1	1	
EC	CPX	3	4	abs
ED	SBC	3	4	abs
EE	INC	3	6	abs
EF	BBS6	3	5	zpr
F0	BEQ_	2	2*	rel
F1	SBC	2	5*	izy
F2	SBC	2	5	izp
F3	-	1	1	
F4	-	2	4	zpx
F5	SBC	2	4	zpx
F6	INC	2	6	zpx
F7	SMB7	2	5	zpg
F8	SED	1	2	
F9	SBC	3	4*	aby
FA	PLX	1	4	
FB	-	1	1	
FC	-	3	4	abs
FD	SBC	3	4*	abx
FE	INC	3	7	abx
FF	BBS7	3	5	zpr

*) v poli T znamená:
- +1T, pokud adresa překročí hranici stránky
- +1T, pokud je vykonán podmíněný skok
- +1T u instrukcí ADC a SBC, pokud je D=1

Adresní módy:
- imm = #$00
- zpg = $00
- zpr = $00,$0000 (PC-relative)
- zpx = $00,X
- zpy = $00,Y
- izp = ($00)
- izx = ($00,X)
- izy = ($00),Y
- abs = $0000
- abx = $0000,X
- aby = $0000,Y
- ind = ($0000)
- iax = ($0000,X)
- rel = $0000 (PC-relative)

Instrukce procesoru 6809

Instrukční sada 6809																						
Instrukce	Tvar	Adresní mód															Popis	CC bit				
		Immediate			Direct			Indexed			Extended			Inherent				5	3	2	1	0
		Op	~	#	Op	~	#	Op	~	#	Op	~	#	Op	~	#		H	N	Z	V	C
ABX														3A	3	1	X = B+X (Bez znaménka)					
ADC	ADCA	89	2	2	99	4	2	A9	4+	2+	B9	5	3				A = A+M+C	+ +	+ +	+ +	+ +	+ +
	ADCB	C9	2	2	D9	4	2	E9	4+	2+	F9	5	3				B = B+M+C	+	+	+	+	+
ADD	ADDA	8B	2	2	9B	4	2	AB	4+	2+	BB	5	3				A = A+M	+	+	+	+	+
	ADDB	CB	2	2	DB	4	2	EB	4+	2+	FB	5	3				B = B+M	+	+	+	+	+
	ADDD	C3	4	3	D3	6	2	E3	6+	2+	F3	7	3				D = D+M:M+1		+	+	+	+
AND	ANDA	84	2	2	94	4	2	A4	4+	2+	B4	5	3				A = A && M		+	+	0	
	ANDB	C4	2	2	D4	4	2	E4	4+	2+	F4	5	3				B = B && M		+	+	0	
	ANDCC	1C	3	2													C = CC && IMM	?	?	?	?	?
ASL	ASLA													48	2	1	Posun vlevo	8	+	+	+	+
	ASLB													58	2	1		8	+	+	+	+
	ASL				08	6	2	68	6+	2+	78	7	3					8	+	+	+	+
ASR	ASRA													47	2	1	Posun vpravo	8	+	+		+
	ASRB													57	2	1		8	+	+		+
	ASR				07	6	2	67	6+	2+	77	7	3					8	+	+		+
BIT	BITA	85	2	2	95	4	2	A5	4+	2+	B5	5	3				Bit Test A (M&&A)		+	+	0	
	BITB	C5	2	2	D5	4	2	E5	4+	2+	F5	5	3				Bit Test B (M&&B)		+	+	0	
BSR	BSR													8D	7	2	Skok do podprogramu					
Bxx	BCC													24	3	2	C = 0 (alias: BHS)					
	BCS													25	3	2	C = 1 (alias: BLO)					
	BEQ													27	3	2	Z = 1					
	BGE													2C	3	2	N xor V = 0					
	BGT													2E	3	2	Z or (N xor V) = 0					
	BHI													22	3	2	C or Z = 0					
	BLE													2F	3	2	Z or (N xor V) = 1					
	BLS													23	3	2	C or Z = 1					
	BLT													2D	3	2	N xor V = 1					
	BMI													2B	3	2	N = 1					
	BNE													26	3	2	Z = 0					
	BPL													2A	3	2	N = 0					
	BRA													20	3	2	Vždy					
	BRN													21	3	2	Nikdy					
	BVC													28	3	2	V = 0					
	BVS													29	3	2	V = 1					
CLR	CLRA													4F	2	1	A = 0		0	1	0	0
	CLRB													5F	2	1	B = 0		0	1	0	0
	CLR				0F	6	2	6F	6+	2+	7F	7	3				M = 0		0	1	0	0

Instrukční sada 6809																										
Instrukce	Tvar	Adresní mód																Popis	CC bit				
		Immediate			Direct			Indexed			Extended			Inherent					5	3	2	1	0
		Op	~	#	Op	~	#	Op	~	#	Op	~	#	Op	~	#			H	N	Z	V	C
CMP	CMPA	81	2	2	91	4	2	A1	4+	2+	B1	5	3					Porovnání M s obsahem A	8	+	+	+	+
	CMPB	C1	2	2	D1	4	2	E1	4+	2+	F1	5	3					Porovnání M s obsahem B	8	+	+	+	+
	CMPD	10 83	5	4	10 93	7	3	10 A3	7+	3+	10 B3	8	4					Porovnání M:M+1 s obsahem D		+	+	+	+
	CMPS	11 8C	5	4	11 9C	7	3	11 AC	7+	3+	11 BC	8	4					Porovnání M:M+1 s obsahem S		+	+	+	+
	CMPU	11 83	5	4	11 93	7	3	11 A3	7+	3+	11 B3	8	4					Porovnání M:M+1 s obsahem U		+	+	+	+
	CMPX	8C	4	3	9C	6	2	AC	6+	2+	BC	7	3					Porovnání M:M+1 s obsahem X		+	+	+	+
	CMPY	10 8C	5	4	10 9C	7	3	10 AC	7+	3+	10 BC	8	4					Porovnání M:M+1 s obsahem Y		+	+	+	+
COM	COMA													43	2	1		A = complement(A)		+	+	0	1
	COMB													53	2	1		B = complement(B)		+	+	0	1
	COM				03	6	2	63	6+	2+	73	7	3					M = complement(M)		+	+	0	1
CWAI		3C	=> 20	2														CC = CC ^ IMM; Wait for Interrupt					7
DAA														19	2	1		Decimal Adjust A		+	+	0	+
DEC	DECA													4A	2	1		A = A 1		+	+	+	
	DECB													5A	2	1		B = B 1		+	+	+	
	DEC				0A	6	2	6A	6+	2+	7A	7	3					M = M 1		+	+	+	
EOR	EORA	88	2	2	98	4	2	A8	4+	2+	B8	5	3					A = A XOR M		+	+	0	
	EORB	C8	2	2	D8	4	2	E8	4+	2+	F8	5	3					B = M XOR B		+	+	0	
EXG	R1,R2	1E	8	2														Zaměnit R1,R2					
INC	INCA													4C	2	1		A = A + 1		+	+	+	
	INCB													5C	2	1		B = B + 1		+	+	+	
	INC				0C	6	2	6C	6+	2+	7C	7	3					M = M + 1		+	+	+	
JMP					0E	3	2	6E	3+	2+	7E	4	3					pc = EA					
JSR					9D	7	2	AD	7+	2+	BD	8	3					Skok do podprogramu					

Instrukce	Tvar	Adresní mód															Popis	CC bit				
		Immediate			Direct			Indexed			Extended			Inherent				5	3	2	1	0
		Op	~	#	Op	~	#	Op	~	#	Op	~	#	Op	~	#		H	N	Z	V	C
LD	LDA	86	2	2	96	4	2	A6	4+	2+	B6	5	3				A = M		+	+	0	
	LDB	C6	2	2	D6	4	2	E6	4+	2+	F6	5	3				B = M		+	+	0	
	LDD	CC	3	3	DC	5	2	EC	5+	2+	FC	6	3				D = M:M+1		+	+	0	
	LDS	10 CE	4	4	10 DE	6	3	10 EE	6+	3+	10 FE	7	4				S = M:M+1		+	+	0	
	LDU	CE	3	3	DE	5	2	EE	5+	2+	FE	6	3				U = M:M+1		+	+	0	
	LDX	8E	3	3	9E	5	2	AE	5+	2+	BE	6	3				X = M:M+1		+	+	0	
	LDY	10 8E	4	4	10 9E	6	3	10 AE	6+	3+	10 BE	7	4				Y = M:M+1		+	+	0	

Instrukce	Tvar	Adresní mód															Popis	CC bit				
		Immediate			Direct			Indexed			Extended			Inherent				5	3	2	1	0
		Op	~	#	Op	~	#	Op	~	#	Op	~	#	Op	~	#		H	N	Z	V	C
LEA	LEAS							32	4+	2+							S = EA					
	LEAU							33	4+	2+							U = EA					
	LEAX							30	4+	2+							X = EA			+		
	LEAY							31	4+	2+							Y = EA			+		
LSL	LSLA													48	2	1	Logický posun vlevo		+	+	+	+
	LSLB													58	2	1			+	+	+	+
	LSL				08	6	2	68	6+	2+	78	7	3						+	+	+	+
LSR	LSRA													44	2	1	Logický posun vpravo		0	+		+
	LSRB													54	2	1			0	+		+
	LSR				04	6	2	64	6+	2+	74	7	3						0	+		+
LBSR	LBSR													17	9	3	Skok do podprogramu					
LBxx	LBCC													10 24	5(6)	4	C = 0 (alias: BHS)					
	LBCS													10 25	5(6)	4	C = 1 (alias: BLO)					
	LBEQ													10 27	5(6)	4	Z = 1					
	LBGE													10 2C	5(6)	4	N xor V = 0					
	LBGT													10 2E	5(6)	4	Z or (N xor V) = 0					
	LBHI													10 22	5(6)	4	C or Z = 0					
	LBLE													10 2F	5(6)	4	Z or (N xor V) = 1					
	LBLS													10 23	5(6)	4	C or Z = 1					
	LBLT													10 2D	5(6)	4	N xor V = 1					
	LBMI													10 2B	5(6)	4	N = 1					
	LBNE													10 26	5(6)	4	Z = 0					
	LBPL													10 2A	5(6)	4	N = 0					
	LBRA													10 20	5(6)	4	Vždy					
	LBRN													10 21	5(6)	4	Nikdy					
	LBVC													10 28	5(6)	4	V = 0					
	LBVS													10 29	5(6)	4	V = 1					
MUL														3D	11	1	D = A*B (Bez znaménka)			+		9
NEG	NEGA													40	2	1	A = !A + 1	8	+	+	+	+
	NEGB													50	2	1	B = !B + 1	8	+	+	+	+
	NEG				00	6	2	60	6+	2+	70	7	3				M = !M + 1	8	+	+	+	+

Instrukce	Tvar	Adresní mód																Popis	CC bit				
		Immediate			Direct			Indexed			Extended			Inherent				5	3	2	1	0
		Op	~	#	Op	~	#	Op	~	#	Op	~	#	Op	~	#		H	N	Z	V	C
NOP														12	2	1	No Operation					
OR	ORA	8A	2	2	9A	4	2	AA	4+	2+	BA	5	3				A = A || M		+	+	0	
	ORB	CA	2	2	DA	4	2	EA	4+	2+	FA	5	3				B = B || M		+	+	0	
	ORCC	1A	3	2													C = CC || IMM	?	?	?	?	?
PSH	PSHS	34	5+	2													Push Registers on S Stack					
	PSHU	36	5+	2													Push Registers on U Stack					
PUL	PULS	35	5+	2													Pull Registers from S Stack					
	PULU	37	5+	2													Pull Registers from U Stack					
ROL	ROLA													49	2	1	Rotace vlevo přes CY		+	+	+	+
	ROLB													59	2	1			+	+	+	+
	ROL				09	6	2	69	6+	2+	79	7	3						+	+	+	+
ROR	RORA													46	2	1	Rotace vpravo přes CY		0	+		+
	RORB													56	2	1			0	+		+
	ROR				06	6	2	66	6+	2+	76	7	3						0	+		+
RTI														3B	6/15	1	Návrat z přerušení	?	?	?	?	?
RTS														39	5	1	Návrat z podprogramu					
SBC	SBCA	82	2	2	92	4	2	A2	4+	2+	B2	5	3				A = A – M – C	8	+	+	+	+
	SBCB	C2	2	2	D2	4	2	E2	4+	2+	F2	5	3				B = B – M – C	8	+	+	+	+
SEX														1D	2	1	Znaménkové rozšíření B do A		+	+	0	
ST	STA				97	4	2	A7	4+	2+	B7	5	3				M = A		+	+	0	
	STB				D7	4	2	E7	4+	2+	F7	5	3				M = B		+	+	0	
	STD				DD	5	2	ED	5+	2+	FD	6	3				M:M+1 = D		+	+	0	
	STS				10 DF	6	3	10 EF	6+	3+	10 FF	7	4				M:M+1 = S		+	+	0	
	STU				DF	5	2	EF	5+	2+	FF	6	3				M:M+1 = U		+	+	0	
	STX				9F	5	2	AF	5+	2+	BF	6	3				M:M+1 = X		+	+	0	
	STY				10 9F	6	3	10 AF	6+	3+	10 BF	7	4				M:M+1 = Y		+	+	0	
SUB	SUBA	80	2	2	90	4	2	A0	4+	2+	B0	5	3				A = A – M	8	+	+	+	+
	SUBB	C0	2	2	D0	4	2	E0	4+	2+	F0	5	3				B = B – M	8	+	+	+	+
	SUBD	83	4	3	93	6	2	A3	6+	2+	B3	7	3				D = D – M:M+1		+	+	+	+
SWI	SWI													3F	19	1	Software interrupt 1					
	SWI2													10 3F	20	2	Software interrupt 2					
	SWI3													11 3F	20	2	Software interrupt 3					
SYNC														13	>= 4	1	Synchronize to Interrupt					
TFR	R1,R2	1F	6	2													R2 = R1					
TST	TSTA													4D	2	1	Test A		+	+	0	
	TSTB													5D	2	1	Test B		+	+	0	
	TST				0D	6	2	6D	6+	2+	7D	7	3				Test M		+	+	0	

Seznam ilustrací

Seznam ilustrací

PORTY, BAJTY, OSMIBITY
Počítače na koleni

Martin Malý

Vydavatel:
CZ.NIC, z. s. p. o.
Milešovská 5, 130 00 Praha 3
Edice CZ.NIC
www.nic.cz

1. vydání, Praha 2019
Kniha vyšla jako 21. publikace v Edici CZ.NIC.
Tisk: Digitální tisk Praha s.r.o., Za Hřbitovem 1073/3, Dubeč, 107 00 Praha

ISBN 978-80-88168-39-3 (tištěná verze)
ISBN 978-80-88168-40-9 (ve formátu EPUB)
ISBN 978-80-88168-41-6 (ve formátu MOBI)
ISBN 978-80-88168-42-3 (ve formátu PDF)

Edice CZ.NIC

O knize Po knize Hradla, volty, jednočipy přichází autor s volným pokračováním, v němž se od základů číslicové techniky přesuneme ke konstrukci složitějších obvodů - totiž vlastních počítačů. Na příkladu tří podrobně popsaných konstrukcí se čtenář naučí základní principy fungování osmibitových počítačů a vyzkouší si i jejich praktický návrh a zapojování. Konstrukce používají legendární osmibitové procesory Intel 8085, MOS 65C02 a Motorola 6809. Kniha popisuje nejen jejich zapojení, ale věnuje se i jejich programování ve strojovém kódu (assembleru). Autor se nevyhnul ani pokročilejším tématům, jako je generování zvuku, zapojování periferií nebo různým způsobům rozšiřování paměti. Všechny konstrukce jsou prakticky ověřené, a ačkoli vycházejí z „ducha osmdesátých let", tak používají součástky, které lze i dnes poměrně dobře sehnat. Kniha je doplněna online dokumentací jednotlivých konstrukcí, ukázkami programového vybavení a dodatkovým materiálem.

O autorovi **Martin Malý** (známý též pod přezdívkou Arthur Dent či Adent), je český programátor, bloger, publicista a komentátor. V roce 2003 naprogramoval a spustil veřejnou blogovací službu Bloguje.cz, od roku 2007 psal pravidelný sloupek pro Digiweb - součást webu iHNed.cz. Později pracoval na pozici šéfredaktora webového magazínu Zdroják (vydávala společnost Internet Info, s.r.o.). V březnu 2013 nastoupil do vydavatelství Economia na pozici vedoucího týmu redakčních vývojářů iHNed.cz. Od roku 2015 popularizuje Internet věcí a DIY elektroniku, přednáší o těchto oborech a školí zájemce o platformu Arduino. Za svou popularizační činnost byl v roce 2016 nominován v anketě Křišťálová Lupa, v kategorii One (wo)man show. Od roku 2017 je zástupcem šéfredaktora iHNed.cz pro rozvoj a placený obsah. Elektronika je jeho koníčkem už od dětství. Nejraději má staré osmibitové počítače z osmdesátých let 20. století – vlastní jich několik desítek a stále je jimi fascinován natolik, že si staví jejich repliky a programuje jejich emulátory. Ve spolupráci se sdružením CZ.NIC přednáší v kurzu „Arduino pro učitele".

O edici Edice CZ.NIC je jedním z osvětových projektů správce české domény nejvyšší úrovně. Cílem tohoto projektu je vydávat odborné, ale i populární publikace spojené s Internetem a jeho technologiemi. Kromě tištěných verzí vychází v této edici současně i elektronická podoba knih. Ty je možné najít na stránkách knihy.nic.cz.

knihy.nic.cz

ISBN 978-80-88168-42-3

9 788088 168423 >