FAKULTA ELEKTROTECHNIKY A INFORMATIKY
TECHNICKÁ UNIVERZITA V KOŠICIACH

Emulátor 8-bitových CPU

Semestrálny projekt 1

Šk.rok: 2006/2007

Peter Jakubčo

Obsah

Zoznam obrázkov

1 Úvod

Tento projekt sa zaoberá riešením emulácie 8-bitových CPU programovou realizáciou. Má slúžiť hlavne učiteľom a študentom na pochopenie práce týchto procesorov, ale možno poslúži aj ako pomôcka pri vývoji reálnych aplikácií. Program obsahuje zatiaľ výkonný kompilátor assembleru a emulátor procesora *Intel i8080*.

Pri programovaní som použil jazyk *Java*, pretože mojou hlavnou požiadavkou je, aby bol emulátor čo najviac prenositeľný, a intuitívne jednoduchý. Výber tohto programovacieho jazyka má však určite negatívny dopad na výslednú rýchlosť programu, keďže je sám emulátor interpretovaný.

1.1 História mikroprocesorov

Úplne prvým komerčným samostatným CPU čipom sa stal *Intel 4004*, ktorý sa na trhu objavil v novembri roku 1971. Tento 4-bitový procesor bol vytvorený pre kalkulačky. Aj keď dokáže spracovať dáta vo veľkosti 4 bitov, jeho inštrukcie sú 8 bitov dlhé. Program a dáta sú separované.

V roku 1972, firma Texax Instruments trocha predbehla *Intel 4004/4040* so svojim 4-bitovým *TMS 1000*, ktorý ako prvý obsahoval dostatok RAM[1] pamäte, a priestor pre programovú ROM[2], a dovoľoval tak prácu bez použitia viacerých externých podporných čipov. Takisto podporoval novú vlastnosť — možnosť pridania vlastných inštrukcií do CPU.

Procesor *8080*, nasledovník procesora *8008*, bol na trh vypustený v apríli roku 1974. Kým *8008* mal 14-bitové programové počítadlo aj adresovanie, *8080* má 16-bitovú adresovú a 8-bitovú dátovú zbernicu. Vnútorná štruktúra obsahuje sedem 8-bitových registrov na všeobecné použitie, 16-bitový smerník [3] do operačnej pamäte, ktorý nahradil 8-úrovňový vnútorný zásobník procesora *8008*, a 16-bitové programové počítadlo. *8080* disponuje s 256 I/O portami[4], čiže zariadenia môžu byť pripojené súčasne bez nutnosti ich prepájania alebo rušenia adresného priestoru. Procesor *8080* bol použitý napr. v počítači *Altair 8800*, v prvom široko známom osobnom počítači (skôr domácom počítači, keďže definícia „prvého PC“ je nie celkom jasná).

Zlepšený návrh firmy Intel vyústil do procesora s označením *8085* (rok 1976), ktorý podporoval ďalšie dve inštrukcie na povolenie/zakázanie troch pridaných pinov prerušení a takisto sériových I/O pinov. Hardvér je zjednodušený tak, aby mu stačilo napätie +5V. Do čipu je pridaný hodinový generátor a obvody radiča zbernice.

Úmyslom vytvorenia procesora *Zilog Z-80* v roku 1976, bolo vylepšiť

[1] Random Access Memory
[2] Read Only Memory
[3] SP - stack pointer
[4] Input/Output — vstupno/výstupné porty

procesor *8080*, čo sa aj naozaj podarilo. Tiež používal 8-bitové dáta a 16-bitové adresovanie, a mohol vykonať všetky inštrukcie procesora *8080* (ale nie *8085*), no obsahuje o 80 viac inštrukcií. Sada registrov bola zdvojená — dve banky dátových registrov, ktoré mohli byť medzi sebou prepínané. Jednou z výhod bolo rýchle prepínanie medzi operačným systémom a prerušeniami. Do *Z-80* boli tiež pridané dva indexové registre a dva typy relokovateľných vektorových prerušení. To, čo naozaj spôsobilo, že sa tento procesor stal tak populárnym v dizajne, bolo jeho pamäťové rozhranie — CPU generoval vlastné signály pre obnovu obsahu RAM. Znamenalo to jednoduchší návrh a nižšiu cenu systému, čo bol rozhodujúci faktor pri výbere procesora pre počítač *TRS-80 Model 1*. Táto výhoda spolu s jeho kompatibilitou s *8080*, plus použitie *CP/M*, prvého štandardného operačného systému pre mikroprocesory, ho spravili prvou voľbou pre mnoho systémov.

Len krátko po uvedení procesora *Intel 8080*, v roku 1975, firma Motorola predstavila svoj procesor *6800*. Niekoľko návrhárov z tejto firmy odišlo, aby vytvorili spoločnosť MOS Technologies. Táto spoločnosť predstavila sériu procesorov s označením *650x*, ktorá zahŕňala procesor *6501* (pinovo kompatibilný s *6800*, stiahnutý z trhu skoro okamžite po jeho uvedení kvôli právnym nezhodám) a *6502*. Podobné sérii *6800*, boli vytvorené varianty, ktoré pridávali do procesorov nové vlastnosti, ako I/O porty, alebo boli lacnejšie s menšími adresovými zbernicami (*6507* mal 13-bitovú 8K adresovú zbernicu, použitý v počítači *Atari 2600*). Procesory *650x* používali malý byte-ový endián[5] (výhodu to malo takú, že nižší byte adresy mohol byť pridaný ku indexovému registru, zatiaľ čo bol vyberaný vyšší byte) a mal úplne odlišnú inštrukčnú sadu ako procesor *6800* s veľkým endiánom[6]. Návrhár Steve Wozniak z firmy Apple procesory *650x* popísal ako prvé čipy, ktoré môžte zohnať za menej ako sto dolárov (čo bolo vtedy asi štvrtina ceny procesora *6800*) — čipy sa stali CPU pre mnohé skoršie domáce počítače (8-bitové produkty *Commodore* a *Atari*).

Podobne ako *6502*, aj procesor *6809* je založený na procesore *Motorola 6800*, ale tento významne rozšíril jeho dizajn. Procesor *6809* mal dva 8-bitové akumulátory a mohol ich skombinovať do jedného 16-bitového registra. Disponoval dvomi indexovými registrami a dvomi smerníkmi na zásobník (stack pointers), čo umožnilo použitie veľmi pokročilých adresných módov. *6809* bol na úrovni zdrojového kódu kompatibilný s procesorom *6800*, aj keď mal *6800* 78 inštrukcií a *6809* iba okolo 59. Niektoré inštrukcie boli nahradené viac všeobecnými, ktoré assembler mohol preložiť, a niektoré boli dokonca nahradené adresnými módmi. Kým aj *6800*, aj *6502* mali rýchly 8-bitový mód na adresovanie prvých 256 bytov RAM pamäte (jedna „stránka“), procesor *6809* používal 8-bitový *Direct Page* register na nájdenie tejto „rýchlo-adresnej“ stránky nachádzajúcej sa kdekoľvek v 64K

[5] najmenej významný byte je uložený na najnižšej adrese
[6] najvýznamnejší byte je uložený na najnižšej adrese

veľkom adresnom priestore.

1.2 8-bitové CPU

Centrálna procesná jednotka (CPU[7]), alebo niekedy jednoducho procesor, je komponent digitálneho počítača, ktorý intrepretuje inštrukcie počítačového programu a spracováva dáta. CPU poskytujú fundamentálnu počítačovú vlastnosť programovateľnosti, a sú jedným z najpotrebnejších komponentov nachádzaných v počítačoch každej doby, spolu s pamäťou a I/O zariadeniami. CPU, ktorý je vyrábaný ako samostatný integrovaný obvod sa nazýva mikroprocesor.

Každý počítač má svoju „dĺžku slova“, ktorá je pre neho charakteristická. Ide o veľkosť jeho vnútorných pamäťových elementov a prepojovacích ciest (zberníc). Napríklad mikroprocesor, ktorého registre a zbernice môžu uložiť a preniesť 8 bitov informácie, má charakteristickú dĺžku slova 8 bitov a je označovaný ako 8-bitový paralelný[8] procesor. Takýto typ procesora pracuje najefektívnejšie s 8-bitovými binárnymi poľami dát, a preto je pamäť asociovaná s procesorom organizovaná na ukladanie 8-bitových binárnych čísel v jednej pamäťovej bunke, alebo ako čísel, ktoré sú celočíselným násobkom 8-mich bitov: 16 bitov, 24 bitov, atď. Toto charakteristické 8-bitové pole sa nazýva *bajt*[9]

CPU typicky pozostáva z nasledujúcich prepojených funkčných jednotiek:

- Registre
- Aritmeticko/logická jednotka (ALU)
- Riadiace obvody

Registre sa používajú ako dočasná, rýchla pamäť pre CPU. Niektoré registre, ako napr. programové počítadlo a inštrukčný register, majú špeciálne použitie. Ostatné registre, ako napr. akumulátor, slúžia pre viac všeobecné účely.

Akumulátor obyčajne ukladá jeden z operandov, s ktorým bude pracovať ALU. CPU obsahuje rad ďalších registrov pre všeobecné použitie, ktoré môžu byť použité na uloženie operandov alebo priamych dát. Dostupnosť týchto registrov eliminuje potrebu presúvať priame výsledky medzi pamäťou a akumulátorom, čiže sa tým zvyšuje rýchlosť a efektivita práce.

[7] angl. *Central processing unit*
[8] paralelný preto, že dokázal prenášať 8 bitov súčasne, tj. paralelne
[9] angl. *Byte*

Programové počítadlo Inštrukcie tvoriace program sú uložené v operačnej pamäti. Procesor odkazuje na jej obsah, aby sa rozhodol, ktorá inštrukcia je vhodná na vykonanie. Preto potrebuje vedieť pozíciu nasledujúcej inštrukcie. Pozície v pamäti sú číslované od 0. Každé číslo identifikujúce pozíciu v pamäti sa nazýva *adresa*. Programové počítadlo je register, ktorý obsahuje adresu nasledujúcej inštrukcie. Po prečítaní každej inštrukcie je toto počítadlo inkrementované.

Inštrukčný register a dekodér Každá operácia, ktorú môže procesor vykonať, je identifikovaná jedinečným bajtom dát nazývaným *inštrukčný kód*, alebo *operačný kód*. 8-bitové slovo použité ako inštrukčný kód môže rozlíšiť 256 rôznych úkonov, čo bolo vtedy viac ako postačujúce. Procesor vyberá inštrukciu v dvoch rozličných operáciách:

1. Najprv procesor vyšle adresu z programového počítadla do operačnej pamäte.
2. Pamäť následne odošle adresovaný bajt procesoru

CPU uloží tento inštrukčný bajt do inštrukčného registra. Dekodér inštrukciu dekóduje a tak sa spustí jej vykonávanie.

Adresový(é) register/registre CPU môže používať register alebo registrový pár na ukladanie adresy pamäte, ktorá bude prístupná dátam. Ak je adresný register programovateľný, program si môže „postaviť“ adresu pred tým, ako zavolá nejakú inštrukciu, ktorá pracuje s pamäťou.

Aritmeticko/logická jednotka (ALU) Ako vyplýva z názvu, ide o tú časť CPU, ktorá vykonáva aritmetické a logické operácie na binárnych dátach. Obsahuje sčítačku, ktorá je schopná kombinovať obsahy dvoch registrov s logikou binárnej aritmetiky. Jediná sčítačka stačí na to, aby skúsený programátor mohol napísať procedúry na odčítanie, násobenie aj delenie. Avšak v praxi väčšina ALU podporujú aj iné vstavané funkcie, ako hardvérové odčítanie, dvojkové logické operácie a posuvy. ALU obsahuje príznakové bity, ktoré špecifikujú dodatočné informácie o výsledku aritmetických a logických operácií.

Riadiace obvody sú základnou funkčnou jednotkou vo vnútri CPU. Pomocou hodinových impulzov udržujú riadiace obvody správne poradie vykonávania udalostí potrebných pre ľubovoľnú procesnú úlohu. Po vybratí a dekódovaní inštrukcie vyšlú riadiace obvody signály do vnútorných, aj vonkajších jednotiek počítača, ktoré spôsobia začiatok vykonávania daného úkonu. Často tieto obvody dokážu odpovedať na externé signály, ako napríklad žiadosti o prerušenie alebo čakanie. Žiadosť o prerušenie spôsobí dočasné prerušenie vykonávania hlavného

programu, skok na obsluhu prerušenia, a automatický návrat späť na hlavný program. Požiadavka na čakanie je často požadovaná od pamäte alebo I/O zariadenia, ktoré pracuje pomalšie ako CPU. Riadiace obvody zabezpečia zastavenie činnosti CPU, až kým sú dáta pamäte alebo I/O portu pripravené.

1.3 Neformálne o emulácii

1.3.1 Emulácia vs. simulácia

Emulácia je akýsi pokus imitovať vnútornú štruktúru, resp. dizajn zariadenia. Simulácia je na druhej strane pokus o imitovanie funkcie resp. funkcií zariadenia. Napríklad program, ktorý imituje hardvér starej hry *Pacman* [2] a dokáže tak spustiť skutočnú ROM pôvodnej hry *Pacman*, je emulátor. Na druhej strane hra *Pacman* prepísaná pre súčasné počítače, ale ktorá imituje grafiku pôvodnej hry, je simulátor.

1.3.2 Techniky emulácie

Podľa [1] existujú tri základné techniky emulovania kódu. Tieto môžu byť navzájom kombinované, aby sa dosiahol ten najlepší výsledok.

Interpretácia Emulátor načíta do pamäte emulovaný kód bajt po bajte, dekóduje ho, a postupne vykoná príslušné inštrukcie, ktoré menia emulované registre, pamäť a I/O. Všeobecný algoritmus pre prácu emulátora tohto typu je tu:

```
while(CPUIsRunning)
{
  Fetch OpCode
  Interpret OpCode
}
```

K výhodám tohto modelu patria:

- jednoduché ladenie
- portabilita (prenositeľnosť)
- jednoduchá časová synchronizácia (realizovaná počítaním teoreticky ubehnutých hodinových cyklov jednotlivých inštrukcií a následným čakaním času vypočítaného rozdielom skutočne ubehnutých hodinových cyklov od napočítaných teoretických)

Jediná, ale veľká nevýhoda je slabý výkon. Interpretácia zaberá množstvo CPU času a na to, aby sa dali emulovať programy trocha rýchlejšie, je potrebné mať dostatočne rýchly počítač.

V mojom riešení je použitý tento prístup.

Statická rekompilácia V tejto technike sa program napísaný v emulovanom kóde pokúsi preložiť do strojového kódu reálneho počítača. Výsledok je obyčajne spustiteľný súbor, ktorý je možné spustiť na reálnom počítači bez nejakých špeciálnych nástrojov. Aj keď statická rekompilácia znie veľmi dobre, nie je vždy možné ju uskutočniť. Napríklad sa nedá staticky rekompilovať reentrantný (samo-modifikovateľný) kód, pretože nie je možné povedať, ako bude vyzerať po spustení. Aby sa zabránilo takýmto situáciám, dá sa použiť interpretácia v kombinácii so statickou resp. dynamickou rekompiláciou.

Dynamická rekompilácia je v podstate rovnaká ako statická, ale vyskytuje sa počas vykonávania programu. Namiesto snahy prekompilovať celý kód naraz, sa to robí, keď vykonávanie narazí na inštrukcie *CALL* alebo *JUMP*. Na zvýšenie rýchlosti môže byť táto technika kombinovaná so statickou rekompiláciou.

2 Analýza riešenia

Pred návrhom riešenia som si vytýčil hlavné ciele a predstavy o projekte, ktoré mi pomôžu pri detailnom návrhu, aj v implementácii. Tieto ciele sa dajú zhrnúť do nasledujúcich bodov:

- jednoduchosť — program má slúžiť hlavne študentom a učiteľom na výuku, preto musí byť jednoduchý na pochopenie, z čoho vyplýva požiadavka na jasnú a prehľadnú grafickú interakciu s používateľom
- modulárnosť — je dobré písať modulárne programy, sú prehľadnejšie a ľahšie sa odhaľujú chyby. Okrem toho v budúcnosti plánujem do programu pridať možnosť emulácie ďalších procesorov a prídavných zariadení, čo modulárnosť uľahčuje
- prenositeľnosť — je výhodou, keď je program prenositeľný na viacero operačných systémov, pretože tým sa zabezpečí flexibilita programu na používateľský komfort. Program však môže byť prenositeľný na rôznych úrovniach, najnižšia úroveň je strojový kód. Všeobecne platí, že programy napísané v jayzku *Java* sú prenostiteľné na najnižšej možnej úrovni, skoro na úrovni strojového kódu[10]
- lokalizácia — programy, ktoré sú lokalizované do viacerých jazykov, majú nádej svojho väčšieho rozšírenia a zvyšujú pohodlie používania. Rozhodol som sa program preložiť zatiaľ len do angličtiny.

Nie všetky predstavy sa dajú vždy ľahko splniť, lebo závisia od rôznych iných okolností. Rovnako tak niektoré požadované vlastnosti sa môžu navzájom vylučovať, ako napríklad rýchlosť programu a prenositeľnosť. Profesionálne emulátory pracujú väčšinou v reálnom čase (alebo k nemu čo najbližšie) a preto si nemôžu dovoliť časové straty pri interpretácii. Keďže môj projekt slúži hlavne ako výukový materiál, nie je rýchlosť až taká podstatná.

2.1 Komponenty riešenia

Pri štúdiu aj iných, už existujúcich emulátorov (napr. [4], [5]) som si všimol, že je výhoda, keď sa zdrojový kód dá napísať a aj skompilovať priamo v emulátore, preto som sa rozhodol zahrnúť tieto možnosti aj do môjho programu.

Návrh programu som teda rozšíril o ďalšie dve veci. Vzniká tak samotná štruktúra programu, ktorej hlavné časti budem nazývať komponenty — **hlavný modul**, **kompilátor** a **emulátor**. Ďalší návrh pozostáva z návrhu týchto komponentov. Každý komponent je spracovaný oddelene, ale

[10] *Java* prekladá zdrojový kód do kódu virtuálnych inštrukcií (tzv. *bytecode*), ktorý sa interpretuje, nie je to natívny strojový kód

nie je realizovaný úplne samostatne[11]. Základná štruktúra komponentov je zobrazená na obr. 1

Hlavný modul zabezpečuje spoluprácu všetkých komponentov, vytvára používateľské rozhranie a grafický výstup výsledkov. Funkcie ostatných komponentov (ako napr. kompilovanie zdrojového kódu, alebo emuláciu) používateľ aktivuje z tohto modulu.

Obsahuje tiež textový editor na vytváranie zdrojového kódu, ktorý podporuje štandardné funkcie — otvoriť a uložiť súbor, a prácu s clipboardom (vystrihnúť, kopírovať, prilepiť text).

Kompilátor obsahuje funkciu na kompilovanie zdrojového kódu (asemblovanie), pričom sa volá odkazom z textového editora v hlavnom module. Výsledný strojový kód sa uloží do operačnej pamäte procesora (ktorá sa nachádza v emulátore). Ďalej obsahuje funkciu na disasemblovanie inštrukcií, aby ich bolo možné prehľadne zobraziť.

Emulátor je vrcholom môjho riešenia. Skladá sa z emulovaného CPU a operačnej pamäte. Jeho hlavnou úlohou je interpretácia emulovaného kódu uloženého v operačnej pamäti.

Obr. 1: Štruktúra emulátora

2.2 Používanie programu

Ako mohol čitateľ zistiť, každý komponent má svoje základné funkcie so svojimi vstupmi a výstupmi. Vstupy niektorých funkcií sú závislé od výstupu iných, preto je nutné ich volania zoradiť do určitej postupnosti, a nemôžu byť volané paralelne. Inými slovami — je potrebné v prvom rade zabezpečiť, aby sa neemuloval kód, ktorý nezodpovedá zdrojovému kódu v textovom editore. Je to z jednoduchého dôvodu — rozpor prispieva k chaosu, pretože je otázne, ktorý kód je aktuálny.

Používateľ má teda používať program nasledujúcim spôsobom:

1. v textovom editore napísať zdrojový kód programu
2. odkazom v hlavnom module skompilovať zdrojový kód
3. emulátorom sledovať vykonávanie programu

[11] V jazyku *Java* komponenty implementujem ako samostatné balíky (packages), ale nie ako pluginy načítateľné za behu programu

Keďže sa určite nájdu používatelia, ktorí sa pokúsia o obídenie postupu, v prípade akejkoľvek zmeny zdrojového kódu sa zablokuje emulátor, až kým používateľ znovu neprekompiluje kód.

2.3 Popis procesora Intel 8080A

Predtým, ako začnem detailne navrhovať jednotlivé komponenty, potrebujem poznať aspoň základy vnútornej štruktúry procesora, ktorého chcem emulovať. Požiadavka poznania štruktúry nezasahuje iba do samotného emulátora, ale aj do kompilátora, pretože formát inštrukcií a ich mnemonický zápis sčasti definuje aj gramatiku jazyka.

Centrálna procesná jednotka (CPU) pozostáva z nasledujúcich funkčných jednotiek:

- Pole registrov a adresná logika
- Aritmeticko-logická jednotka (ALU)
- Inštrukčný register a riadiaca jednotka
- Obojsmerná, 3-stavová dátová zbernica

Obrázok 2 ilustruje funkčné bloky vnútra CPU.

Obr. 2: Blokový diagram procesora

2.3.1 Registre

Sekcia registrov pozostáva zo statického poľa pamäti RAM organizovanej do šiestich 16-bitových registrov:

- Programové počítadlo (PC)
- Ukazovateľ zásobníka (SP]
- Šesť 8-bitových registrov na všeobecné použitie zoradené v pároch, označované *B,C*; *D,E* a *H,L*
- Pomocný registrový pár (zápisník) nazývaný *W,Z*

Programové počítadlo uchováva adresu v pamäti nasledujúcej inštrukcie a je inkrementovaný automaticky počas každého výberu inštrukcie.

Ukazovateľ zásobníka uchováva adresu nasledujúcej voľnej zásobníkovej pozície v pamäti. Môže byť inicializovaný na hocijakú hodnotu v rámci operačnej pamäte. Ukazovateľ zásobníka je dekrementovaný, keď sú do zásobníka vložené dáta a inkrementovaný, keď sú z neho dáta vyberané[12].

Šesť registrov na všeobecné použitie môžu byť použité ako samostatné registre (8-bitové), alebo ako registrové páry (16-bitové). Pomocný registrový pár *W,Z* nie je programovo adresovateľný[13] a je použitý iba pre vnútorné operácie vykonávania inštrukcií.

2.3.2 Aritmeticko-logická jednotka

ALU obsahuje nasledujúce registre:

- 8-bitový akumulátor
- 8-bitový pomocný akumulátor (ACT)
- 5-bitový príznakový register: nula, prenos, znamianko, parita a pomocný prenos
- 8-bitový pomocný register (TMP)

2.3.3 Inštrukčný cyklus

Riadenie funkcie procesora sa deje riadiacimi signálmi z bloku časovania a riadenia v závislosti od konkrétnej inštrukcie, ktorá je uložená v registri inštrukcií a je zakódovaná pre jednotlivé operačné kroky v dekodéri inštrukcií.

Výber a prevedenie jednej inštrukcie sa označuje ako inštrukčný cyklus. Každý inštrukčný cyklus sa skladá z 1 až 5 operačných cyklov (machine cycle) M1 až M5. Počet operačných cyklov sa rovná počtu oslovení vonkajšej pamäte alebo I/O zariadení. Jeden operačný cyklus sa skladá z 3 až 5 operačných krokov (*states*) T1÷T3 až T5. Práve operačné kroky sú dôležité pri emulovaní v reálnom čase, pretože tie sú dané periódou hodinových impulzov. Pri emulácii sa vykonávané inštrukcie samozrejme nebudú deliť na časti pre vykonanie v operačných krokoch (ako je uvedené v [7] na str.2-7), ale vykonajú sa vcelku. Po vykonaní inštrukcie sa iba k celkovému počtu vykonaných operačných krokov pripočíta počet op. krokov vykonanej inštrukcie ako jedno číslo.

[12] t.j. zásobník rastie smerom k nižším adresám

[13] a preto v mojom emulátore ani nie je implementovaný

3 Návrh a implementácia

Táto kapitola sa zaoberá detailným návrhom môjho emulátora. Začnem návrhom kompilátora, pretože aby som mohol emulovať strojový kód inštrukcií, musím ho najprv získať jeho prekladom zo zdrojového kódu. Druhou časťou je návrh samotného emulátora a posledná časť sa zaoberá návrhom hlavného modulu.

3.1 Kompilátor

Úloha prekladača je nasledovná:

- analyzuje syntaktickú správnosť zdrojového kódu, nazývanú v teórii formálnych jazykov a automatov reťazcom terminálnych symbolov
- transformuje zdrojový kód do cieľového (strojového) kódu, zachovávajúc jeho význam

Tvorba prekladača (kompilátora) asembleru procesora 8080 pozostáva z návrhu a implementácie nasledovných častí:

- gramatiky v EBNF[14]
- sémantiky jazyka
- lexikálneho analyzátora
- syntaktického analyzátora s chybovým zotavením
- 2-prechodového prekladača

Celý kompilátor so všetkými jeho časťami je implementovaný v balíku *proc8080* ako trieda s názvom *asmCompiler*. UML diagram časti triedy (ochudobnený o deklaráciu použitých konštánt) je zobrazený na obr. 3.

Obr. 3: UML Diagram triedy *asmCompiler*

3.1.1 Gramatika

Formálna špecifikácia jazyka je daná jeho gramatikou. Gramatiku môžme definovať ako 4-icu:

Definícia 1 $G = (V_N, V_T, P, S)$, *kde*

- V_N — *je abeceda neterminálnych symbolov*

[14]Rozšírená Bacus-Naurova forma zápisu gramatiky, angl. *Extended Bacus-Naur Form*

- V_T — *je abeceda terminálnych symbolov*
- $S \in V_N$ — *je štartovací symbol*
- P — *je množina substitučných pravidiel* $p : \alpha \rightarrow \beta$, *tj.*

$$P \subseteq \{\alpha \rightarrow \beta | \alpha \in (V_N \cup V_T)^* - V_T^*, \beta \in (V_N \cup V_T)^*\}$$

$p : \alpha \rightarrow \beta$
α *je ľavá strana pravidla* p
β *je pravá strana pravidla* p

Formálny jazyk môžme definovať takto:

Definícia 2 *Jazykom gramatiky G nazývame množinu všetkých jej slov.*

Viac o gramatikách a jazykoch sa môže čitateľ dozvedieť napr. v [6], alebo v [9].

Podľa [3] ak pre návrh jazyka stačí poznať tieto formálne prostriedky návrhu jazyka:

- rozšírená Bacus-Naurova forma pre zápis gramatiky
- denotačná sémantika pre vyjadrenie významu prvkov jazyka

tak jazyk musí byť bezkontextový jazyk *LL(1)*. O aký jazyk ide, sa môže čitateľ dozvedieť v [3].

Teraz uvádzam gramatiku jazyka pre assembler procesora 8080 v EBNF. Gramatiku som navrhol tak, aby pokrývala celú škálu inštrukcií a takisto aby bolo možné používanie ďalších pseudo-inštrukcií, ktoré majú vplyv na preklad.

Terminálne symboly (tzv. lexikálne jednotky) sú vyznačené v úvodzovkách „“. S je štartovacím symbolom.

```
S           -> { Comment | Eol } Org_def
               { Keyword | Comment | Org_def | Eol } Eof
Org_def     -> "org" Addr Eol
Comment     -> ";" { [^\r\n] } Eol
Eol         -> "\r" | "\n" | "\r\n"
Addr        -> Deka | Hexa | Octal | Binary
Number      -> Addr | Char
Keyword     -> Instruction Eol | Id ":" | Variable | Constant
Deka        -> (0|1|...|9) { (0|1|...|9) } ["d"]
Hexa        -> (0|...|9|a|...|f) { (0|...|9|a|...|f) } "h"
Octal       -> 0 { (0|1|...|7) } ["o"]
Binary      -> (0|1) { (0|1) } "b"
```

```
Instruction -> "mov" (Reg "," RegM | RegM "," Reg)
               | ("lda" | "sta" | "lhld" | "shld" | "jmp" |
                  "jnz" | "jz" | "jnc" | "jc" | "jpo" |
                  "jpe" | "jp" | "jm" | "call" | "cnz" |
                  "cz" | "cnc" | "cc" | "cpo" | "cpe" |
                  "cp" | "cm") (Addr | Id)
               | ("in" | "out" | "adi" | "aci" | "sui" |
                  "sbi" | "cpi" | "ani" | "ori" |
                  "xri") Number
               | ("add" | "adc" | "sub" | "sbb" | "inr" |
                  "dcr" | "cmp" | "ana" | "ora" |
                  "xra") RegM
               | ("ldax" | "stax") RegpBD
               | ("pop" | "push") RegpBDHP
               | ("dad" | "inx" | "dcx") Regpair
               | "mvi" Reg "," Number
               | "lxi" RegpBDHS "," Addr
               | "xchg" | "sphl" | "xthl" | "daa" | "cma"
               | "rlc" | "rrc" | "ral" | "rar" | "stc"
               | "cmc" | "pchl" | "ret" | "rnz" | "rz"
               | "rnc" | "rc" | "rpo" | "rpe" | "rp" | "rm"
               | "ei" | "di" | "hlt" | "nop"
               | "rst" (Deka | Hexa | Octal | Binary)
Id          -> (a|b|...|z|_) { (a|b|...|z|_) }
Variable    -> ("db" | "dw") Number
Constant    -> Id "equ" Number Eol
Reg         -> "a" | "b" | "c" | "d" | "e" | "h" | "l"
RegM        -> Reg | "m"
RegpBD      -> "bc" | "de"
RegpBDHS    -> RegBD | "hl" | "sp"
RegpBDHP    -> RegBD | "hl" | "psw"
Regpair     -> RegBDHS | "psw")
Char        -> "'" [^'] "'"
```

> **Poznámka:** Časť `[^\r\n]` v pravidle `Comment` vyjadruje 0 alebo 1 ľubovoľný znak okrem znakov `\r` a `\n`, podobne ako časť `[^']` v pravidle `Char` vyjadruje 0 alebo 1 ľub. znak okrem znaku `'`.
>
> Ešte treba dodať, že gramatika nie je citlivá na veľkosť znakov, teda napr. `mov` a `MOV` vyjadrujú tú istú inštrukciu.

3.1.2 Sémantika

Formálne sa sémantikou nebudem zaoberať, keďže sémantika je daná priamo významom jednotlivých inštrukcií (napr. [7]). Teraz vysvetlím význam po-

užitých pseudoinštrukcií.

org nastaví aktuálnu absolútnu adresu prekladača na konkrétnu hodnotu. Prekladač bude nasledujúce preložené inštrukcie umiestňovať do operačnej pamäti od tejto adresy. Ako vidno z gramatiky, jej jediný parameter je adresa.

Návestie: Táto pseudoinštrukcia sa neprejaví v preklade, označuje konkrétne miesto - adresu, na ktorú je možné použitím návestia bez dvojbodky (:) v zdrojovom kóde odkazovať.

db, dw definujú bajty (*db*), alebo číslo veľkosti dvoch bajtov (*dw*), ktoré nie sú inštrukcie. Môžu byť chápané ako premenné. Často sa označujú návestím.

3.1.3 Lexikálny analyzátor

Lexikálna analýza je fáza prekladu, pri ktorej sú lexikálne jednotky atomizované a podávané na vstup syntaktického analyzátora vo forme symbolov. Lexikálny analyzátor je programový modul realizujúci lexikálnu analýzu.

Trieda *asmCompiler* (obr. 3) obsahuje metódu *getsymbol*, ktorá je lexikálnym analyzátorom. Zo svojho vstupu (textového editora zdojového kódu v hlavnom module) číta symboly a identifikuje ich ako lexikálne jednotky do premennej s názvom *symbol*, resp. *symbol_group*:

Konštanty Na preklad konštánt využíva pomocnú tabuľku konštánt (implementovanú ako *ArrayList* s názvom *constTable*). Do tejto tabuľky sa ukladá binárny tvar konštanty. Konštanty sú prístupné pomocou premennej *constIndex*, ktorá ukazuje na index posledne pridanej konštanty v tejto tabuľke.

Ako vyplýva z gramatiky na str. 15, je možné použiť binárne, osmičkové, dekadické a hexadecimálne konštanty, niekedy aj znakové - uzavreté do jednoduchých úvodzoviek ’’.

Identifikátory využívajú pomocnú tabuľku identifikátorov (implementovanú ako *ArrayList*), do ktorej sa ukladajú postupne. Táto tabuľka býva implementovaná ako pole, ktorého položky nie sú napĺňané postupne, ale pomocou hash funkcie, aby to urýchlilo preklad. V mojom riešení nie je použitý takýto prístup.

Kľúčové slová Jedná sa predovšetkým o inštrukcie a o pseudoinštrukcie.

Lexikálny analyzátor pozorne sleduje pozíciu v texte, kde sa nachádza (riadok, stĺpec). Poslúži to hlavne pri syntaktickom analyzátore s chybovým zotavením na indikáciu miesta prípadnej chyby.

3.1.4 Syntaktický analyzátor s chybovým zotavením

Programová realizácia fázy syntaktickej analýzy zhora nadol vo forme prechodu - programového modulu v rekurzívnom jazyku je podmienená gramatikou, ktorá definuje bezkontextový jazyk *LL(1)*. Takáto syntaktická analýza sa tiež nazýva prediktívnou syntaktickou analýzou, pretože postup pri odvodení je presne určený (predikovaný) aktuálnym symbolom na vstupe - tj. symbolom podaným lexikálnym analyzátorom na základe volania metódy *getsymbol*.

Syntaktický analyzátor je možné konštruovať priamym prepísaním pravidiel gramatiky vo forme EBNF do programu, a to nasledujúcim spôsobom:

- každému pravidlu gramatiky v EBNF zodpovedá deklarácia metódy s názvom neterminálneho symbola na ľavej strane tohto pravidla
- neterminálnemu symbolu na pravej strane pravidla zodpovedá volanie metódy
- terminálnemu symbolu zodpovedá volanie metódy *getsymbol*, ktorá prečíta zo vstupného reťazca ďalší terminálny symbol
- postupnosti symbolov na pravej strane každého pravidla zodpovedá postupnosť volaní metód uvedených v predchádzajúcich bodoch
- operácii sumy `|` a výrazu v hranatých zátvorkách `[` a `]` zodpovedá príkaz vetvenia (`if`, `switch`). Vstup do príkazu vetvenia je bodom rozhodnutia.
- uzáveru `{...}` zodpovedá príkaz cyklu (`while`). Vstup do príkazu cyklu je bodom rozhodnutia.
- metóda spustenia kompilácie (*startCompile*) volá metódu odpovedajúcu štartovaciemu symbolu gramatiky

Na obr. 3 si čitateľ môže všimnúť, že trieda obsahuje niektoré metódy s rovnakými názvami ako sú ľavé strany pravidiel gramatiky na str. 15. Jedná sa o spomínaný prepis neterminálov na ľavej strane pravidiel na deklaráciu metód.

Body rozhodnutia sú mimoriadne dôležité pri rozšírení syntaktického analyzátora o zotavenie pri výskyte syntaktickej chyby, pretože podľa toho, aký je aktuálny symbol na vstupe, uskutoční sa výber vetvy príkazu vetvenia, resp. sa rozhodne o ďalšej iterácii cyklu.

Cieľom zotavenia pri výskyte syntaktickej chyby je zistiť počas syntaktickej analýzy maximálny počet syntaktických chýb s normálnym ukončením syntaktického analyzátora (a to aj v prípade, že syntaktický analyzátor je súčasťou prekladu), a umožniť transformáciu syntakticky nesprávneho vstupného kódu do syntakticky správneho výstupného kódu. Existujú podmienky

účinného zotavenia z chýb a spôsob, ako zotavenie implementovať. Týmito vecami sa však nebudem zaoberať, čitateľ ich môže nájsť napr. v [3].

3.1.5 Prekladač

Preklad je pojem označujúci transformáciu zdrojového kódu do cieľového (strojového) kódu. Vykonanie cieľového kódu, na rozdiel od interpretácie, nie je samozrejme súčasťou prekladu.

Prekladač je nevyhnutne 2-prechodový. V prvom prechode sa preložia inštrukcie, ktoré neodkazujú na žiadne návestia a je tak možné ich preložiť hneď do absolútneho kódu. Pri preklade sa eviduje aktuálna absolútna adresa v operačnej pamäti (premenná s názvom *memoryPosition*), ktorá sa inkrementuje po preložení každej takejto inštrukcie. Pri preklade pseudoinštrukcie `org` sa táto premenná nastavi na novú hodnotu.

Ešte v stále v prvom prechode sa pre ostatné inštrukcie (tie, ktoré odkazujú na návestia) vytvárajú dve pomocné tabuľky. Prvá sa nazýva tabuľka zámen (premenná s názvom *idReplacements*), ktorá sa napĺňa pri používaní návestí (už definovaných) a má dva stĺpce:

1. identifikátor (návestie) vo forme indexu v tabuľke identifikátorov
2. adresu v operačnej pamäti, ktorá má byť nahradená (ešte neznámou) adresou identifikátora

Druhou tabuľkou je tabuľka definícií (premenná *idDefinitions*), ktorá sa napĺňa pri definícii návestia a má nasledujúce stĺpce:

1. identifikátor (návestie) vo forme indexu v tabuľke identifikátorov
2. nahradzujúca konštanta ()

Pri napĺňaní tabuľky definícií ak sa vyskytne viac položiek s rovnakým identifikátorom, znamená to chybu - snahu o predefinovanie návestia.

V druhom prechode sa spracujú obe tabuľky - nahradia sa obsahy operačnej pamäte na adresách nachádzajúcich sa v tabuľke zámen pre všetky položky, ktoré majú odkaz na ten istý identifikátor v tabuľke definícií. Po každej zámene sa z tabuľky zámen odstráni spracovaná položka. Ak na konci druhého prechodu v tejto tabuľke ešte nejaké položky ostanú, znamená to chybu - nedefinované návestia.

3.1.6 Disasembler

Pre potreby grafického výstupu emulácie súčasne v interakcii s používateľom, je potrebné poznať mnemonické zápisy inštrukcií. Dalo by sa to vyriešiť použitím zápisov zo zdrojového kódu. Problém vznikne pri vytváraní samomodifikovateľného kódu, ktorý mení aktuálne inštrukcie, resp. pri skokoch na

miesta v operačnej pamäti, ktoré neboli zaplnené inštrukciami v zdrojovom kóde.

Tieto skutočnosti rieši disassembler (metóda *disassembleInstruction*), ktorý na svojom vstupe očakáva adresu operačnej pamäte a jeho výstupom je disasemblovaná inštrukcia vo forme poľa obsahujúceho tieto položky:

1. mnemonický zápis inštrukcie v tvare reťazca (*string*)
2. operačný kód inštrukcie v tvare reťazca (*string*)
3. odkaz na ďalšiu inštrukciu (adresa v operačnej pamäti)

3.2 Emulátor

Návrh samotného emulátora bola asi najzaujímavejšou časťou tvorby programu. Začnem s popisom implementácie hardvéru, potom prejdem k popisu samotnej emulácie. Celý emulátor je implementovaný ako samostatná trieda v balíku *proc8080* s názvom *Cpu*. UML diagram tejto triedy je na obr. 4. Trieda implementuje rozhranie *Runnable*, čo znamená, že ide o samostatné vlákno.

Obr. 4: UML Diagram triedy *Cpu*

3.2.1 CPU

Pri implementácii vnútornej štruktúry skutočného procesora sa postupuje úplne inak ako pri hardvérovom návrhu. Celá funkcia hardvéru sa nahradí pár príkazmi v programovacom jazyku. Aby však bola implementácia funkcii hardvéru čo najbližšie, je potrebné dbať na dodržanie rôznych obmedzení (ako napr. veľkosť premenných, resp. znamienko), aby sa softvér správal v rôznych situáciách rovnako alebo čo najpodobnejšie skutočnému, reálnemu správaniu hardvéru.

Registre CPU som implementoval iba tie, ktoré sú programom viditeľné, alebo ktoré sú nevyhnutné pre beh emulovaného procesora: **A**, **B**, **C**, **D**, **E**, **H**, **L**, **FR** (*Flag resgister*, register príznakov), **PC** (programové počítadlo) a **SP** (ukazovateľ zásobníka). Implementované registre sú ako samostatné (súkromné) dátové členy triedy *Cpu*. Všetky 8-bitové registre predstavujú premenné typu `short`, 16-bitové registre sú typu `int`. Nevýhodou je, že jazyk Java nepozná bezznamienkové typy, preto bolo nutné použiť typ, ktorého rozsah kladných hodnôt zahŕňa požadovaný počet bitov. Typ `short` je 16-bitový so znamienkom, tj. rozsah jeho hodnôt je od -32768 do $+32767$. Aj keď je tento typ trochu predimenzovaný na 8-bitové čísla (požadujú rozsah hodnôt od 0 do 255), je to najbližší „vyšší“ typ od typu `byte`, ktorý

je 8-bitový so znamienkom (a má teda rozsah od −128 do iba +127). Podobne to platí aj pre 16-bitové registre a im odpovedajúci 32-bitový typ `int`. Tento implementačný detail má však dôležité následky — pri akejkoľvek zmene registrov je potrebné zabezpečiť kontrolu, aby nová hodnota neprekročila počet bitov registra. Dá sa to jednoducho zabezpečiť logickým súčinom novej hodnoty s hodnotou maximálneho čísla, ktoré sa zmestí do daného počtu bitov[15]. Príznakový register je síce 5-bitový, ale príznaky nie sú v skutočnosti usporiadané do postupnosti 5-tich bitov, ale do 8-mich, kde určité pozície dvoch bitov majú pevnú hodnotu. Potom príznakový register s vynulovanými všetkými príznakmi má hodnotu binárne `01000010`.

Hodnoty registrov sa menia pri emulácii programu, ale dajú sa zmeniť, resp. získať aj explicitne zavolaním niektorej z metód *setReg??*, resp. *getReg??*, kde *??* predstavuje názov registra. V prípade programového počítadla sa jeho hodnota mení metódou *setPC*, resp. je ju možné získať metódou *getPC*. Register príznakov sa nedá meniť vcelku, ale existujú metódy na nastavenie, resp. získanie hodnoty jednotlivých príznakov. Viac viď obr. 4

Možnosti explicitného nastavenia hodnôt registrov som zaviedol z jednoduchého dôvodu. Myslím si, že každého zvedavého používateľa, ktorý sa učí ako funguje procesor, napadne otázka, že čo sa stane ak zmení hodnotu niektorého registra resp. príznaku na inú ako je očakávaná a tak ovplyvní prácu nasledujúcej inštrukcie. Ako sa čitateľ ďalej dozvie, dajú sa explicitne meniť aj hodnoty operačnej pamäte a tým pádom je možné modifikovať aj samotný program preložený v strojovom kóde. Nemá to však vplyv na zdrojový kód umiestnený v textovom editore.

Pre rýchlejšie počítanie príznakov niektorých inštrukcií som zostrojil 4 pomocné tabuľky:

parityTable obsahuje 256 položiek typu `boolean` [16], ktoré sú `True`, ak má index na nich odkazujúci v binárnom vyjadrení párny počet bitov, a `False`, ak nie. Túto tabuľku využívajú azda všetky inštrukcie, ktoré nejakým spôsobom môžu ovplyvniť zmenu paritného bitu v registri príznakov.

signTable tiež obsahuje rovnaký počet položiek typu `boolean`, ktoré sú `True`, ak je ich index väčší ako +127 a `False` inak. Ide o tabuľku znamienok. Znamienkový bit v celočíselnej aritmetike procesora 8080 je najvýznamnejší bit čísla, preto pre 8-bitové registre sú čísla od +128 vyššie chápané ako záporné.

inrTable Ide o tabuľku príznakov konkrétnej inštrukcie — `inr`, ktorá zvýši hodnotu registra (svojho argumentu) o 1. Položky sú typu `short`. Index tejto tabuľky vyjadruje číslo pred inkrementáciou. Použitie tejto

[15] pre 8-bitové registre je to 255, pre 16-bitové 65536

[16] môže nadobúdať len dve hodnoty: `True`, alebo `False`

tabuľky urýchľuje vykonanie inštrukcie.

dcrTable Ide o tabuľku príznakov konkrétnej inštrukcie — `dcr`, ktorá zníži hodnotu registra (svojho argumentu) o 1. Položky sú typu `short`. Index tejto tabuľky vyjadruje číslo pred dekrementáciou.

Keď do skutočného procesora privedieme napätie, začne okamžite pracovať. Obsahy registrov ako je programové počítadlo, ukazovateľ zásobníka a ostatné pracovné registre sú na začiatku nastavené náhodne a nemôžu byť špecifikované. Preto je potrebné začať spustenie procesora sekvenciou spustenou signálom **RESET**[17].

Externý signál **RESET** minimálneho trvania troch hodinových cyklov nastaví programové počítadlo na 0. Vykonávanie programu začne v operačnej pamäti na adrese nula, hneď po ukončení signálu RESET. Tento signál však nemá vplyv na príznaky, ako ani na ostatné registre. Obsah týchto registrov zostáva nedeterminovaný, až kým nie je explicitne inicializovaný programom (podľa [7], str. 2-13).

Pri implementácii tohto signálu som niektoré veci upravil, napr. som doplnil inicializáciu aj ostatných registrov. Trieda *Cpu* obsahuje metódu *Reset*, ktorá spôsobí nasledovné:

- vynuluje registre **SP**, **A**, **B**, **C**, **D**, **E**, **H**, **L**
- vynuluje príznaky (nastaví na implcitnú hodnotu $42h$ ($01000010b$))
- programové počítadlo nastaví na začiatok programu (nevynuluje ho). Je to z toho dôvodu, že na nulovej pozícii operačnej pamäte sa nachádzajú po inicializácii pamäte len inštrukcie `nop`, čiže vykonávanie by bežalo naprázdno, až kým by sa programové počítadlo nedostalo na začiatok programu.
- zakáže prerušenia (premenná *INTE*)
- uvedie procesor do stavu *breakpoint*

Metódu volá hlavný modul hneď po skompilovaní zdrojového kódu. Ako je možné vidieť, metóda nespôsobí zmazanie pamäte. Operačná pamäť sa aktualizuje iba po rekompilácii zdrojového kódu.

Po úplnom ukončení programu určitým spôsobom[18] sa beh procesora zablokuje (skutočnému procesoru odpovedá stav **HOLD**). Používateľ tak musí procesor uviesť akoby do stavu **READY**, čiže musí explicitne resetovať procesor. Pomocou hlavného modulu zavolaním metódy *Reset* sa procesor reinicializuje.

[17] do jednotky časovania a riadenia (obr. 2)

[18] výnimka je *breakpoint*, viac viď podkapitolu *Priebeh emulácie*

3.2.2 Operačná pamäť

Najľahší spôsob, ako implementovať operačnú pamäť, je pole bajtov. Prístup k operačnej pamäti sa tak realizuje takýmto spôsobom:

```
Data = Memory[Adr1]; /* čítanie položky z adresy Adr1 */
Memory[Adr2] = Data; /* zápis položky na adresy Adr2  */
```

Taký jednoduchý prístup k operačnej pamäti nie je však vždy možný, a to z niekoľkých dôvodov:

Stránkovaná pamäť Adresný priestor je rozdelený do prepínateľných stránok (alebo bánk). Je to používané hlavne vtedy, keď je adresný priestor príliš malý na adresovanie celej pamäte (64 kB).

Zrkadlená pamäť Jedno miesto v pamäti môže byť dostupné z niekoľko rôznych adries. Napríklad keď program zapíše údaje na adresu $4000h$, tak sa zmeny prejavia aj na miestach $6000h$ a $8000h$. ROM pamäte môžu byť takisto zrkadlené, kvôli nekompletnému dekódovaniu adries.

Ochrana pamäte ROM Niektoré staré kazetové programy (napr. MSX hry) sa najprv pokúšajú zapísať údaj do vlastnej ROM pamäte a ak sa im to podarí, zastavia svoju činnosť. Slúžilo to hlavne kvôli ochrane proti kopírovaniu. Aby sa podarilo emulovať aj tento softvér, je potrebné zabezpečiť, aby do ROM pamäte nebolo možné zapisovať.

Pamäťovo-mapované I/O zariadenia Systém môže obsahovať pamäťovo-mapované I/O zariadenia, čo pri reálnom hardvéri znamená, že prístup na tieto oblasti adries sa pristupuje k zariadeniam, nie k pamäti. Ignorovanie tejto skutočnosti spôsobí „špeciálne efekty“ a preto by pamäť mala byť sledovaná.

Aj keď v mojom emulátore je operačná pamäť implementovaná ako súvislé pole položiek typu `short` bez podpory vyššie uvedených možností, je pre budúce rozšírenia vhodné prístup k pamäti implementovať metódami. Celú operačnú pamäť som implementoval do balíka *devices* a samostatnej triedy *Memory*, viď UML diagram na obr. 5.

Obr. 5: UML Diagram triedy *Memory*

Na prácu s 8-bitovými položkami v triede slúžia verejné metódy *getCell*, *setCell*. Pre 16-bitové položky som implementoval metódy *getCellWord* a *setCellWord*, ktoré pracujú so 16-bitovými údajmi a do operačnej pamäte sú ukladané v malom endiáne. Metódy sú implementované tak, aby boli čo možno najrýchlejšie, pretože sa volajú veľmi často (hlavne čítanie).

3.2.3 Priebeh emulácie

Ako som uviedol v kapitole *Techniky emulácie* na strane 6, moja emulácia pracuje čistou interpretáciou kódu. Emulátor dokáže pracovať v dvoch režimoch:

Krok — emulovanie jednej inštrukcie a následné pozastavenie procesora (čo by skutočnému procesoru odpovedalo uvedeniu do stavu **WAIT**) použitím špeciálnej premennej *breakpoint*

Súvislé emulovanie kódu , až kým sa emulovanie nejakým spôsobm neukončí (popísané nižšie). Spustenie súvislej emulácie odpovedá spusteniu vykonávaniu vlákna. Ešte pred jej spustením je vytvorená pomocná tabuľka breakpointov, ktorá je implementovaná ako množina (*HashSet*). Jej položky obsahujú adresy, na ktorých má program zastaviť. Po vykonaní každej inštrukcie je testovaná na prítomnosť adresy, na ktorú ukazuje programové počítadlo. Ak ju tabuľka obsahuje, procesor je pozastavený ako v predošlom prípade.

Súvislá emulácia

Keďže emulátor nepracuje so zariadeniami, súvislé emulovanie kódu prebieha v jednoduchom cykle, ktorý je možné približne vyjadriť nasledovne:

```
running = true;
while(running) {
  evalStep();
  if (breaks.contains(PC) == true)
    running = false;
}
```

Najprv je formálne spustená emulácia nastavením premennej *running* na `True`. Táto premenná je verejná v celej triede. Nasleduje cyklus, ktorý bude bežať, až kým sa táto premenná nezmení na `False`, čiže pokým nebude formálne ukončená emulácia.

Premennú *running* ovplyvňuje metóda *evalStep*, ktorej úlohou je dekódovanie a emulovanie jednej inštrukcie (na adrese hodnoty programového počítadla **PC**). Používa sa aj pri krokovom režime emulovania.

Posledným krokom v cykle je kontrola, či množina breakpointov (s názvom *breaks*) obsahuje hodnotu programového počítadla - teda adresu nasledujúcej inštrukcie. Ak áno, beh emulácie je formálne zastavený nastavením premennej *running* na `False`.

Krok emulácie

Metóda *evalStep* pracuje takto:

```
definite = true;
```

```
instr = mem.getCell(PC++);
switch (instr) {
  case opCode1: ... break;
  case opCode2: ... break;
  case opCode3:
  ...
  default: definite = false;
}

if (definite == false) running = false;
if (PC > 0xFFFF) {
  PC &= 0xFFFF;
  running = false;
}
```

Na začiatku premenná *definite* nastavená na `True` určuje, že inštrukcia bola identifikovaná, teda hodnota operačnej pamäte na adrese programového počítadla obsahuje platný operačný kód. Riadok

```
instr = mem.getCell(PC++);
```

predstavuje načítanie operačného kódu z pamäte.

Nasleduje vetvenie, ktorého vetvy sú rozpoznané operačné kódy - teda inštrukcie. V týchto vetvách sa deje interpretácia inštrukcií. Špeciálna vetva `default` sa vykoná iba v prípade, že prečítaný operačný kód nie je platným kódom žiadnej inštrukcie. V tejto vetve sa tak premenná *definite* nastaví na `False`.

Po vetvení sa realizuje kontrola, či bol načítaný operačný kód skutočne platný. Ak nie, formálne sa ukončí emulácia a znamená to výpadok inštrukcie.

Poslednou operáciou je kontrola, či adresa programového počítadla presiahla svoju definovanú hranicu 16-tich bitov. Ak áno, formálne sa ukončí emulácia, a znamená to výpadok adresy.

Ukončenie emulácie

Práca procesora sa pozastaví, alebo úplne zastaví, ak sa vyskytne jedna z nasledujúcich okolností:

- narazenie na inštrukciu `hlt`, ide o normálne ukončenie vykonávania programu, teda o úplné zastavenie činnosti procesora
- výpadok inštrukcie - procesor sa úplne zastaví
- výpadok adresy - procesor sa úplne zastaví
- breakpoint - dočasné pozastavenie emulovania. Obnovenie práce je možné explicitným spustením kroku inštrukcie, alebo spustením súvislého vykonávania emulácie

V prípade, že sa procesor úplne zastavil, je ho možné znovu spustiť až po jeho resetovaní (reinicializovaní), čo môže urobiť používateľ explicitne z hlavného modulu.

3.2.4 Real-time emulácie

Real-time znamená reálny čas, teda ide o emuláciu, ktorá inštrukcie emuluje v rovnakých časových intervaloch, ako reálny hardvér. Úplne presné časové intervaly nemožno nikdy softvérovou emuláciou dosiahnuť a to hlavne z dôvodu, že interpretácia inštrukcií a hardvérová realizácia sa implementačne líšia. Riešenie emulácie v reálnom čase sa skladá z dvoch faktorov, ktoré treba riešiť:

- emulovaný softvér je príliš pomalý
- emulovaný softvér je príliš rýchly

Riešenie pomalosti emulátora môže vyplývať okrem iného aj zo skutočnosti, že na emuláciu používame príliš pomalý hardvér (možno pomalší ako emulovaný, prípadne nedostatočne rýchly). Riešenie tohto problému spočíva jedine v použití lepšieho hardvéru. V prípade, že hardvér nie je až tak pomalý (je niekoľko krát rýchlejší ako reálny hardvér, ktorý emulujeme), emulátor treba implementačne optimalizovať. Jedna vec je zvoliť správny programovací jazyk (asi najlepší je priamo assembler, resp. C) a druhá vec je použitie takých operácií, ktoré nie sú veľmi náročné na čas CPU.

Je vždy lepšie, ak je emulátor príliš rýchly, pretože riešenie je principálne jednoduché. Pointa je v tom, že ak poznáme teoretický čas, za ktorý by sa mala inštrukcia vykonať a odmeriame čas, za ktorý sa naozaj vykonala, treba jednoducho počkať (nič nevykonávať) čas rozdielu skutočného času od teoretického.

V mojom emulátore som sa stretol s riešením obidvoch faktorov, prvý vyplýva hlavne z toho, že samotný emulátor je len interpretovaný virtuálnym strojom *JVM*. Používanie grafických operácií na vykreslenie aktuálnej inštrukcie a odozva zmien v registroch a operačnej pamäti tiež značne prispeli k spomaleniu. Preto som bol nútený tieto veci počas behu emulácie (súvislej) neaktualizovať.

Popis detailnejšieho riešenia „spomalenia“ emulátora si vyžaduje trošku zamyslenia.

V reálnom procesore perióda hodinového impulzu je podľa katalógu [10] $0.48 \div 2.0\mu s$, ktorá vlastne predstavuje trvanie jedného operačného kroku. V [7], [10], je pri každej inštrukcii uvedený počet operačných krokov, koľko trvá. Je to výhoda, pretože sa nemusím zaoberať operačnými cyklami. Priemerný počet operačných krokov potrebných na vykonanie 1 inštrukcie je 8.84. Toto číslo ďalej využijem.

Definícia 3 *Hodinová frekvencia procesora sa vypočíta podľa vzťahu:*

$$f = \frac{1}{T}$$

kde T je perióda hodinového impulzu.

Bežná frekvencia reálneho procesora bola $1MHz$, čiže periódu hodinových impulzov mal $1\mu s$. Princíp zaistenia chcenej frekvencie (ktorá je samozrejme softvérovo nastaviteľná) spočíva vo vhodnom nastavení periódy hodinových impulzov a následnom čakaní.

Pozmením algoritmus zo str. 26, aby čo najjednoduchšie riešil emulovanie v reálnom čase:

```
running = true;
startTime = microSecondsTime();
while (running) {
  tStates = evalStep();
  endTime = microSecondsTime();
  startTime += (tStates * tStateMicroSecondsPeriode);
  if (startTime > endTime) {
    waitMicroSeconds(startTime - endTime);
  }
  if (breaks.contains(PC) == true)
  running = false;
}
```

V použitom algoritme vymyslená metóda `microSecondsTime` vráti čas v ms od nejakého iniciálneho bodu, napríklad od začiatku programu a metóda `waitMicroSeconds` počká čas, ktorý je očakávaný v ms.
Vymyslená premenná `tStatMicroSecondsPeriode` predstavuje periódu hodinových impulzov v ms, `tStates` je počet operačných krokov poslednej vykonanej inštrukcie.

Keďže v Jave existujú funkcie na čakanie s presnosťou asi na $10ms$ a na meranie času s presnosťou $1ns$, musím rátať s presnosťou menšou, tj. $10ms$ a preto sa oplatí uvedený prístup použiť iba v prípade, ak `tStates` > 10. Priemerne je `tStates` $= 8.84$ a preto musím „pustiť“ minimálne 2 inštrukcie bez kontroly, aby bola kontrola presnejšia. Riešenie bude:

```
running = true;
tStatesInSlice = 15;
tStates = 0;
startTime = microSecondsTime();
while (running) {
  tStates += evalStep();
```

```
    if (tStates >= tStatesInSlice) {
      endTime = microSecondsTime();
      startTime += (tStatesInSlice
                    * tStateMicroSecondsPeriode);
      tStates -= tStatesInSlice;
      if (startTime > endTime)
        waitMicroSeconds(startTime - endTime);
    }
    if (breaks.contains(PC) == true)
      running = false;
  }
```

Tento prístup je založený na poznatku, že hodinová frekvencia udáva počet vykonaných operačných krokov za časovú jednotku. Pre jednotku frekvencie kHz by to bol počet krokov za $1ms$. Z toho sa potom odvádza aj hodnota premennej `tStatesInSlice`, ktorá predstavuje počet operačných krokov, ktoré sa vykonajú bez časovej kontroly.

Definícia 4 *Formálna definícia hodnoty tejto premennej vyplýva z nasledujúcich požiadaviek:*

- *počet operačných krokov vykonaných bez kontroly sa počíta za jednu časovú jednotku (periódu hodinových impulzov)*
- *z toho vyplýva zvýšenie ubehnutého teoretického času o túto časovú jednotku, po ubehnutí tohto počtu krokov*

$tStatesInSlice = clockFrequency * sliceLength$, *kde* `sliceLength` *je iba časové doladenie presnosti (má hodnotu 1).*

Inak povedané, `tStatesInSlice` vyjadruje koľko operačných krokov za časovú jednotku sa má vykonať a tak teoretický ubehnutý čas po ich vykonaní sa má zväčšiť o túto časovú jednotku.

Potom moje konečné riešenie vyzerá takto:

```
running = true;
tStatesInSlice = clockFrequency * sliceLength;
tStates = 0;
startTime = System.nanoTime();
while(running) {
  tStates += evalStep();
  if (tStates >= tStatesInSlice) {
    startTime += 1000000;
    tStates -= tStatesInSlice;
    endTime = System.nanoTime();
```

```
        if (startTime > endTime)
          waitNanoTime(startTime - endTime);
      }
      if (breaks.contains(PC) == true)
        running = false;
    }
    endTime = System.nanoTime();
```

3.3 Hlavný modul

Návrh hlavného modulu pozostával hlavne z návrhu textového editora a grafickej interakcii používateľa s emulátorom. Modul obsahuje jedno hlavné grafické okno, ktoré sa delí na dva prepínateľné panely. Tento modul som implementoval do balíka s názvom *emu8*. Je rozdelený do niekoľkých tried (ktorých UML diagramy kvôli rozsahu tejto dokumentácie neuvádzam): *debugTableModel*, *frmMain*, *frmAbout* a *memoryTableModel*. Trieda *frmMain* má ešte niekoľko vnorených tried na implementáciu modelov, resp. zobrazovania niektorých ovládacích prvkov okna.

3.3.1 Panel zdrojového kódu

Tento panel dáva používateľovi všetky možnosti fázy vytvárania programu - ide hlavne o textový editor a odkaz na kompilovanie napísaného zdrojového kódu. Textový editor podporuje zobrazovanie čísel riadkov, čo určite pomôži pri rýchlejšom nájdení prípadnej chyby v texte.

3.3.2 Panel emulátora

Tu je zabezpečená grafická interakcia používateľa s emulátorom. Používateľ tu môže vidieť:

hodnoty operačnej pamäte , zobrazené v tabuľke implementovanej ako *JTable*, ktorá používa vlastný model, triedu *memoryTableModel*. Tento model dynamicky vyberá a priraďuje hodnoty operačnej pamäte. Zobrazenie operačnej pamäte je rozdelené na stránky[19], aby bolo šetrené miestom a takisto to zvyšuje prehľadnosť. Ak používateľ hľadá konkrétnu adresu, stačí, ak klikne na tlačidlo „Nájsť adresu“, ktoré spôsobí nastavenie takej stránky operačnej pamäte, ktorá obsahuje danú adresu.

ladiace okno , v ktorom sú prehľadne v stĺpcoch zobrazené adresy operačnej pamäte a k nim prislúchajúce inštrukcie v mnemonickom tvare plus ich operačné kódy.

[19]Pozor, nejedná sa o implementovanie stránkovania operačnej pamäte

časť vnútornej štruktúry CPU , ako sú registre (vrátane registrových párov), príznaky a hodnota vybranej bunky operačnej pamäti v dekadickom, binárnom, osmičkovom, hexadecimálnom a v priamom grafickom zobrazení podľa ASCII kódovej tabuľky. Registre sa dajú explicitne zmeniť jednoduchým prepísaním ich hodnoty a potvrdením klávesou ENTER. Počas súvislej emulácie nie je možné meniť žiadne hodnoty registrov, ani operačnej pamäte.

časy vykonaného programu, resp. inštrukcie ako skutočné, tak aj teoretické vrátane počtu vykonaných operačných krokov.

4 Zhodnotenie a plány do budúcna

Literatúra

[1] *How To Write a Computer Emulator*, http://fms.komkon.org/EMUL8/HOWTO.html

[2] *Pac-Man*, http://en.wikipedia.org/wiki/Pac-Man

[3] KOLLÁR, J., HAVLICE Z.: 2001, *Technológia jazykových systémov.* Košice: elfa, s.r.o, 183 s. ISBN 80-89066-12-7

[4] *8086 Microprocessor Emulator*, http://www.emu8086.com/

[5] *8085 Simulator*, http://www.programmersheaven.com/download/47231/download.aspx

[6] HUDÁK, Š.: 2003, *Strojovo orientované jazyky.* Košice: Fakulta elektrotechniky a informatiky, TU v Košiciach, 218 s. ISBN 80-969071-3-1

[7] Intel corp.: 1975, *Intel 8080 Microcomputer Systems User's Manual*, 240 s.

[8] *UML Modeling: Reverse Engineering Java Applications*, http://www.netbeans.org/kb/55/uml-re.html

[9] HUDÁK, Š.: 2002, *Teoretická Informatika.* Košice, dostupné na internete: http://hornad.fei.tuke.sk/predmety/tispz/ti.ps

[10] TESLA: *Katalóg*