PŘÍRODOVĚDECKÁ FAKULTA UNIVERZITY PALACKÉHO
KATEDRA INFORMATIKY

DIPLOMOVÁ PRÁCE

Počítače s von Neumannovou architekturou

2012 Pavla Bláhová

Anotace

Tato diplomová se zabývá návrhem a simulací činnosti moderního 32-bitového superskalárního procesoru s instrukční sadou typu RISC.

Děkuji svému vedoucímu práce RNDr. Arnoštu Večerkovi za čas a cenné rady, které mi věnoval při konzultacích.

Obsah

Seznam obrázků

1. Úvod

1.1. Požadavky

Cílem diplomové práce je modelování činnosti moderního superskalárního procesoru. Práce by měla obsahovat:

- Návrh jednoduššího superskalárního procesoru. Procesor může být architektury CISC nebo RISC, postačující je 16-bitový.
- Návrh instrukčního souboru pro daný procesor.
- Sestavení jazyka symbolických adres (assembler) pro daný instrukční soubor a sestavení překladače z jazyka symbolických adres do strojového jazyka.
- Aplikaci, která modeluje základní rysy činnosti superskalárního procesoru (spekulativní výběr prováděných instrukcí, předvídání větvení atd.).

1.2. Popis řešení

Zadání jsem implementovala jako simulační aplikaci pro Windows nazvanou `Simulace CPU`. Do jejího hlavního okna lze zadat program napsaný v jazyce symbolických adres a krok po kroku demonstrovat jeho provádění. Pro každý výpočetní krok je k dispozici náhled na obsah jednotlivých registrů a zásobníku, výpis používané paměti a zobrazená konzole pro případný textový výstup prováděného programu. Kvůli srovnání výkonu aplikace `Simulace CPU` nabízí možnost volby vykonání programu jednoduchým 32-bitovým procesorem, nebo jeho superskalární variantou.

Ve svém textu nejdříve čtenáře seznamuji se základními pojmy, které mu usnadní orientaci v problematice návrhu procesoru. Dále postupně představuji svůj návrh 32-bitového superskalárního procesoru s instrukční sadou typu RISC. Text obsahuje podrobný popis tohoto procesoru spolu s vysvětlením a popsáním fází překladu tak, aby po jeho přečtení byl čtenář schopen použít aplikaci `Simulace CPU` – napsat program v navrženém jazyce symbolických adres a zorientovat se v jednotlivých demonstrovaných krocích výpočtu tohoto programu.

2. Architektura procesorů

2.1. Von Neumannova architektura

Von Neumannova architektura popisuje počítač se společnou pamětí pro instrukce i data. To znamená, že zpracování je sekvenční oproti například harvardské architektuře, která je typickým představitelem paralelního zpracování.

Procesor počítače se skládá z řídicí a výkonné (aritmeticko-logické) jednotky. Řídicí jednotka zpracovává jednotlivé instrukce uložené v paměti, přičemž jejich vlastní provádění nad daty má na starosti aritmeticko-logická jednotka. Vstup a výstup dat zajišťují vstupní a výstupní jednotky (zdroj [1]).

2.2. CISC vs. RISC

Procesory jsou podle návrhu instrukční sady rozděleny do dvou kategorií – RISC (redukovaná instrukční sada) a CISC (komplexní instrukční sada). V 70. letech byla rozšířenější architektura CISC, protože v této době byla operační paměť drahá a malá (16 KB za cenu kolem $500). Bylo zapotřebí zmenšit množství paměti potřebné pro uložení samotného programu, tedy každá instrukce musela umět co nejvíce věcí – vedlo to ke zesložitění instukcí. Nejen paměť, ale samotný procesor byl drahou záležitostí, a proto se obě tyto části musely využívat je co nejefektivněji.

Potřebu složitějšího (dražšího) hardwaru pro tyto složitější instrukce, vyřešil návrh mikroprogramu – interpretu, který z komplexní instrukce vygeneruje posloupnost jednodušších instrukcí, které jsou vykonány hardwarem. To mělo za následek, že se návrháři snažili navrhovat architektury CISC, které se sémanticky moc nelišily od vyšších programovacích jazyků.

Zastánci architektury RISC postupovali empiricky a snažili se ve svém návrhu zohlednit fakt, že procesor vykonává nejčastěji jen omezený počet instrukcí - instrukce skoku a přiřazení. Díky jednodušší vnitřní logice je možné zkrátit dobu zpracování jednotlivých instrukcí a tím zvýšit výkonnost systému.

2.3. Assembler – jazyk symbolických adres

Assembler je nízkoúrovňový programovací jazyk závislý na instrukční sadě a architektuře procesoru. Pro kvalitní programování je důležité znát vnitřní detaily procesoru jako např. architekturu, organizaci paměti, atd. Instrukce assembleru přímo odpovídají instrukcím strojového jazyka. To je jediný jazyk, kterému rozumí vlastní procesor (jeho instrukce sestávají pouze z řetězců jedniček a nul).

Instrukce architektury RISC mají pevnou délku a většinou 2 operandy (zdroj, cíl) uloženy buď přímo v paměti, nebo registrech. Pro programy napsané ve vyšším programovacím jazyce platí, že jsou kratší a srozumitelnější, což pro progra-

mátora znamená, že jsou napsány rychleji a lépe se udržují. Další jejich výhodou je, že jsou přenositelné.

Jaké jsou tedy důvody pro používání assembleru? Hlavními důvody jsou efektivita a dosažitelnost systémového hardwaru. Programy napsané v assembleru mají oproti programům napsaných ve vyšších programovacích jazycích menší časovou i paměťovou složitost. To proto, že obsahují pouze kód, který je nezbytný pro chod programu, a tedy i výsledná velikost přeložitelného programu je menší.

2.4. Návrh procesoru

2.4.1. ISA (sada instrukcí)

Jednou z charakteristik architektury instrukční sady (ISA [1]) je počet adres používaných v instrukci. Většina instrukcí představuje binární či unární operace s jedním výsledkem. Výjimkou je např. operace dělení: produkuje dva výstupy (samotný výsledek a jeho zbytek).

Kvůli binárním operacím potřebujeme celkem tři adresy: pro `první operand`, `druhý operand` a `výsledek`. Jejich počet lze zredukovat na 2, když se jeden ze vstupních operandů použije pro i výstup (přepíše se). `1-adresové stroje` (akumulátory) se používaly v době, kdy byla paměť drahá a pomalá. Všechny jeho instrukce pracují nad speciálním registrem.

`0-adresové stroje` používají k výpočtu zásobník. Na vrcholu zásobníku jsou vstupní operandy a výsledek se tam uloží také. Srovnáním těchto přístupů zjistíme, že počet instrukcí roste se snižujícím se počtem použitých adres. Ovšem na druhou stranu výkonnostní metriku určují přístupy do paměti a nesmí se zapomenout ani na náročnost čtení instrukce. V praxi se proto používá kombinace těchto adresování.

RISC procesory používají speciální architekturu známou jako `load/store` architekturu. Instrukční sada obsahuje 2 speciální instrukce `load` a `store`, které slouží k přesunu dat mezi registry a pamětí. Všechny další instrukce pracují výhradně s registry. Další charakteristikou této instrukční sady je režim adresování, který spolu s počtem adres přímo ovlivňuje formát instrukcí.

Typy operandů Instrukce procesoru typicky podporují pouze základní datové typy jako znaky, celá čísla a čísla s plovoucí čárkou. Pro snadnější práci s nimi jsou čísla rozdělena na znaménková a neznaménková. V paměti adresovatelné po bytech je nejmenší adresovatelnou velikostí jeden byte. Pro větší operandy se využívá více bytů. Instrukce v procesorech typu CISC jsou přizpůsobeny, aby pracovaly s operandy různých velikostí, jejichž velikost určuje použitý registr (`EAX` 32 bitů, `AX` 16 bitů, `AL` 8 bitů).

[1] ISA - instruction set architecture

V architekturách RISC se velikost operandů načítaných z paměti určuje přímo v operacích `LOAD` a `STORE`. Jsou to jediné instrukce, které umožňují přesun mezi pamětí a registry, ostatní instrukce pracují pouze s registry nebo konstantami.

Režimy adresování operandů Režimy adresování popisují, jak jsou určeny operandy. Operandy mohou být na třech místech: v registrech, v paměti, nebo součástí instrukce jako konstanty – přímé hodnoty. Určení operandu pomocí registru se nazývá `registrový režim adresování`, podobně pro konstantu `přímý režim adresování`. Architektury CISC a RISC se liší v počtu `paměťových režimů adresování`. CISC procesory podporují mnoho paměťových režimů adresování, procesory typu RISC podporují pouze jeden nebo dva tyto režimy adresování.

Instrukce přesunující data V této kategorii rozlišujeme dva druhy instrukcí, ty, které přesunují data mezi pamětí a registry (`load`, `store`) a speciální instrukce pro práci s daty na zásobníku (`push`, `pop`) a instrukce dělající přesuny dat pouze mezi registry(`mov`).

Aritmetické a logické instrukce Aritmetické instrukce pracují s celými čísly i s čísly s plovoucí čárkou. Většina procesorů poskytuje 4 základní matematické operace: sčítání, odčítání, násobení a dělení. Díky dvojkovému komplementárnímu systému není nutné mít zvlášť instrukce pro sčítání a odčítání znaménkových a neznaménkových čísel. Násobení a dělení těchto čísel už však oddělené být musí. Některé procesory operaci dělení neposkytují vůbec, případně jen částečně (bez zbytku).

Základní logické operace `AND` a `OR` jsou podporovány všemi procesory. Většina procesorů podporuje i další logiské operace `NOT` a `XOR`.

Formát instrukcí Procesory používají 2 základní typy formátu instrukcí: `instrukce s~pevnou délkou`, kde mají všechny (nebo alespoň většina) instrukce stejnou velikost a `instrukce s~proměnnou délkou instrukce`. Velikost instrukce závisí na počtu adres, a zda tyto adresy určují registr nebo místo v paměti. CISC povolují použití adresy paměti, a proto podporují instrukce s proměnnou délkou. RISC má operandy výhradně v registrech. Pro operandy v registrech postačuje velikost pro délku instrukce 18 bitů: 8 bitů určuje kód operace (až 256 druhů instrukcí), adresa operandu je dlouhá 5 bitů (až 32 registrů). Pro pamět s délkou adresy 32 bitů je potřeba 72 bitů na zakódování celé instrukce: 8 bitů pro kód operace, a 32 bitů pro adresu operandu. Kvůli celkové velikosti se většinou povoluje pouze jeden operand s adresou paměti.

2.4.2. Řízení výpočtu

Důležitou roli při výpočtu programu hraje speciální registr `IP`, který udržuje adresu následující instrukce. Vždy po načtení instrukce změní svou hodnotu na další adresu další instrukce. Zpravidla probíhá výpočet sekvenčně, tedy hodnota `IP` se automaticky zvětšuje o délku instrukce.

Pro případy, kdy potřebujeme změnit sekvenční model, máme k dispozici speciální instrukce pro větvení programu (viz následující odstavec 2.4.3.) nebo můžeme zavolat proceduru (viz 2.4.4.).

2.4.3. Větvení programu

Větvení programu je dosaženo použitím přímých a nepřímých větvicích instrukcí. `Přímé větvicí instrukce` vyžadují adresu cíle, tedy konkrétní adresu následující instrukce. `Nepřímé větvicí instrukce` mají cílovou adresu specifikovánu hodnotou v paměti nebo registru. Nejjednodušším typem větvení je tzv. `nepodmíněné větvení` s `absolutně` nebo `IP-relativně` zadanou cílovou adresou. Pokud je adresa zadána absolutně, procesor načte tuto adresu do registru `IP` a pokračuje ve výpočtu. V případě IP-relativní adresy je tato adresa přičtena k aktuální hodnotě `IP`. Hlavní výhoda používání relativních adres je v možnosti přesunu kódu z jednoho bloku paměti do jiného bez změny cílové adresy.

Při podmíněném větvení se skok (narušení sekvenčního výpočtu) provede pouze tehdy, pokud byla splněna podmínka. Existují dva základní přístupy:

Set-Then-Jump Testování podmínky a větvení jsou samostatné instrukce. Výsledek testovací instrukce se uloží do tzv. podmínkových registrů, které udržují záznamy stavů poslední aritmetické operace. Větvicí instrukce se zařídí podle jejich hodnoty. Podmínkové registry bývají nahrazeny příznaky. Např. dvojice instrukcí `cmp r1,r2; jne skok;` porovná parametry `r1`, `r2` a pokud se nerovnají, výpočet pokračuje na návěští `skok:`.

Test-and-jump Testování a větvení je zkombinováno do jedné instrukce. Testovací instrukce bere tři parametry: dvě testované hodnoty a adresu skoku. Např. `jne r1, r2, skok;` Stejně jako v předchozím příkladu porovná parametry `r1`, `r2` a pokud se nerovnají, výpočet přejde na návěští `skok:`.

2.4.4. Volání procedur

Rozdíl mezi větvením a voláním procedury je v tom, že větvením program pouze přejde na další instrukci, odkud pokračuje ve výpočtu. Při volání procedury se výpočet přesměruje na tělo procedury a po jejím provedení se výpočet přesune na následující adresu po volání této procedury. Pro takový návrat je zapotřebí

uložit si adresu za instrukcí volání procedury (používá se zásobník, nebo speciální registr) a znát indikátor konce této procedury (instrukce `return`).

Existuje několik způsobů předávání parametrů volané proceduře. První možnost je předat parametry v registrech. Tato možnost je rychlá, ale kvůli omezenému počtu registrů nejde např. provést rekurzivní volání procedur. Pomalejší možností je předat parametry procedurám pomocí jejich uložení na zásobník.

2.4.5. Zpracování větvení

Moderní procesory jsou vysoce superskalární – používají více výpočetních jednotek a paralelně přednačítají a předzpracovávají další instrukce (`pipelining`). V takovém procesoru se musí k větvení přistupovat individuálně. Dokud se nevětví, tak je toto předzpracování efektivní. Pokud se větví, musíme zrušit všechny dosud předzpracované instrukce a efektivita `pipeliningu` i paralelního zpracování se nejen vytrácí, ale i brzdí výpočet.

Lze to vyřešit zpožděním výpočtu – pokud se narazí na větvicí instrukci, `pipeliny` se naplní až poté, co bude dokončena. Lepší řešení však je nečekat a při už zpracování instrukce pro větvení se pokusit odhadnout, zda k větvení dojde, či nikoli. Pro větvení je nutné znát cílovou adresu. Většina instrukcí používá přímé větvení, kde adresa následující instrukce je součástí větvicí instrukce. U nepřímého větvení už to tak přímočaré není, neboť adresa není součástí instrukce –např. může být uložena v registru. V takovém případě se cílová adresa se musí zkusit odhadnout.

Problémem ale je zjistit, jestli se větvení provede. Používají se 3 strategie odhadu větvení: `fixní`, `statická` a `dynamická`.

Fixní Jednoduchá na implementaci, předpokládá, že buď dojde k větvení vždy (volání a návrat procedur, smyčky) nebo nikdy. Strategie, že k větvení nikdy nedojde, poskytuje provádění instrukcí po sobě a plně využívá `pipeliningu` i paralelního zpracování. Předpoklad, že k větvení dojde vždy, zase poskytuje možnost předpřipravit instrukce cílových adres ve větvích.

Statická Používá předpoklad, že větvení se nikdy neprovede, pouze pro podmíněné větvení.

Dynamická Velmi účinný a jednoduchý algoritmus pro předpověď. Výběr větve se bere podle většiny `n` předchozích výběrů. Např. pro `n = 3`. Pokud se dvakrát po sobě nestrefí, dojde ke změně předpovědi. Při nepřímé adresaci, použijeme pro předzpracování adresu předchozího skoku.

3. Popis implementovaného procesoru

V práci jsem implementovala procesor typu RISC s jednoduchou instrukční sadou. Jedná se o 32-bitový procesor se 17 registry a lineárně adresovanou pamětí. Jeho výkon zvyšuje jednoúrovňová asociativní cache a dvou úrovňový paralelismus.

Procesor během běhu programu předpokládá, že se nebudou provádět změny programu v paměti. Důvodem je přednačítání instrukcí při superskalárním a pipeliningovém provádění. Procesor při něm nekontroluje, jestli se v paměti neobjevila jiná verze přednačtené instrukce. Toto chování by nemělo být na škodu, protože změny programu za běhu stejně nepatří k dobrým programátorským technikám.

3.1. Registry

Pro použití v programech jsou k dispozici víceúčelové registry `A`, `B`, ... `L`. Dále procesor obsahuje několik speciálních registrů. Pro práci se zásobníkem slouží registry `ESP` a `EBP` (viz níže). Obsah registru `IP` ukazuje na další prováděnou instrukci. Obsah registru `IDT` ukazuje na začátek tabulky vektorů přerušení. Registr `FLAGS` je popsán v následujícím odstavci.

3.2. Příznaky – registr FLAGS

V registru `FLAGS` jsou využity první 4 bity pro 4 příznaky procesoru. Změny příznaků jsou vedlejším efektem většiny instrukcí. Používají se při porovnávání hodnot, vícebitových operacích apod.

CF	ZF	SF	OF	nevyužito

Obrázek 1. Registr FLAGS.

CF Příznak přenosu do vyššího řádu (carry flag). Nastaví se, když při aritmetické operaci dojde k přenosu do vyššího řádu a oříznutí výsledku.

ZF Příznaky nuly (zero flag). Nastaví se, když je výsledek operace nula.

SF Příznak znaménka (sign flag). Nastaví se, je-li výsledek operace při znaménkové interpretaci záporný.

OF Příznak přetečení (overflow flag). Nastaví se, dojde-li při aritmetické operaci k přetečení mimo rozsah hodnot, který lze uchovat v registru.

3.3. Paměť

Procesor je připojen k paměti obvykle o velikosti 2 KB. Lze ji lineárně adresovat po bytech 32-bitovým ukazatelem. V paměti je umístěn prováděný program, zásobník a tabulka vektorů přerušení. Vícebytové hodnoty ukládá procesor ve stylu little-endian.

3.4. Zásobník

Procesor používá zásobník číselných hodnot. Jeho dno je v paměti na adrese určené registrem `EBP` a vrchol na adrese v registru `ESP`. Přitom platí, že `ESP` $\leq$ `EBP`. Zásobník tedy roste směrem k menším adresám.

3.5. Přerušení

Přerušení jsou metodou volání podprogramů na základě číselného označení. Procesor nedefinuje žádná hardwarová přerušení, ale je možné je definovat softwarově.

Tabulka obsluh přerušení je umístěna v paměti na adrese určené registrem `IDT`. Na adrese `IDT` je uložena adresa obsluhy přerušení 0, na adrese `IDT + 4` adresa obsluhy přerušení 1 atd. Není-li zaregistrována příslušná obsluha, není chování vyvolání přerušení definováno.

V simulaci definuje obsluhu přerušení 0x21 operační systém a používá se pro volání systémových funkcí. Více v části 5. o operačním systému.

3.6. Instrukční sada

V této části si detailně popíšeme instrukční sadu implementovaného procesoru.

Každá instrukce je definována svým operačním kódem, typem a počtem parametrů. Počet parametrů se pohybuje v rozsahu 0 až 2. Typy parametrů rozeznáváme dva:

Registr Parametrem je některý ze 17 registrů procesoru.

Přímá hodnota Parametrem je 32-bitová celočíselná konstanta.

Většina instrukcí pracuje s registry, ale některé umožňují zadat parametr i přímou hodnotou. Jedná se především o instrukce skoku.

3.6.1. Zakódování instrukce

Všechny instrukce jsou shodně zakódovány do 64-bitové hodnoty, která je rozdělena takto (hodnoty v závorkách označují délku v bitech):

Kód (6)	Registr (5)	Registr (5)	Přímá hodnota (32)	Rezerva (16)

Obrázek 2. Zakódování instrukce.

Zakódování umožňuje uvést 3 parametry (2 registry a přímou hodnotu), proto alespoň jeden z nich vždy zůstává nevyužitý. Použité parametry jsou dány kódem instrukce, zbylé obsahují nedefinované hodnoty zapsané překladačem (zde použitý překladač do nich vkládá nuly).

3.6.2. Popis instrukce

Následující část obsahuje popis všech instrukcí implementovaného procesoru. U každé instrukce jsou uvedeny tyto údaje:

- Symbolický název, který se používá v assembleru (např. `MOV`, `ADD`, ...).
- Varianty parametrů instrukce - `R` znamená registr, `I` přímou hodnotu.
- Popis její funkce slovně a pomocí pseudo-kódu.
- Seznam příznaků (flags), které instrukce ovlivňuje.

3.6.3. Seznam instrukcí

Následuje seznam instrukcí implementovaného procesoru. Instrukce jsou uvedeny v abecedním pořadí.

ADC – Součet s přenosem

Varianty `ADC R, R`

Popis Sečte cílový operand (první operand), zdrojový operand (druhý operand) a příznak přenosu (CF) a výsledek uloží do cílového operandu. Instrukce ADC nerozlišuje znaménkové a neznaménkové hodnoty. Obvykle se používá při sčítání větších než 32-bitových čísel.

Fungování `cíl := cíl + zdroj + CF`

Ovlivněné příznaky Příznaky ZF, SF, CF a OF jsou nastaveny podle výsledku.

ADD – Součet

Varianty `ADD R, R`

Popis Sečte cílový operand (první operand) a zdrojový operand (druhý operand) a výsledek uloží do cílového operandu. Instrukce ADD nerozlišuje znaménkové a neznaménkové hodnoty.

Fungování `cíl := cíl + zdroj`

Ovlivněné příznaky Příznaky ZF, SF, CF a OF jsou nastaveny podle výsledku.

AND – Bitový součin

Varianty `AND R, R`

Popis Provede bitový součin cílového operandu (první operand) a zdrojového operandu (druhý operand) a výsledek uloží do cílového operandu.

Fungování `cíl := cíl AND zdroj`

Ovlivněné příznaky Příznaky ZF a SF jsou nastaveny podle výsledku. Ostatní nejsou ovlivněny.

CALL – Volání procedury

Varianty `CALL R` nebo `CALL I`

Popis Uloží aktuální pozici v programu na zásobník a provede skok na adresu zadanou v operandu.

Fungování

```
PUSH IP
IP := operand
```

Ovlivněné příznaky Neovlivňuje příznaky.

CMP – Porovnání operandů

Varianty `CMP R, R`

Popis Porovná první a druhý operand a podle výsledku nastaví příznaky. Provede se odečtením druhého operandu od prvního; výsledek se nikam neukládá.

Fungování `temp := první - druhý`

Ovlivněné příznaky Příznaky ZF, SF, CF a OF jsou nastaveny podle výsledku.

DEC – Zmenšení o 1

Varianty `DEC R`

Popis Odečte od cílového operandu jedničku.

Fungování `operand := operand - 1`

Ovlivněné příznaky Příznaky ZF, SF, CF a OF jsou nastaveny podle výsledku.

END – Ukončení činnosti procesoru

Varianty `END`

Popis Ukončí činnost procesoru. Žádná další instrukce se neprovede.

Ovlivněné příznaky Neovlivňuje příznaky.

IDIV – Znaménkové dělení

Varianty `IDIV R`

Popis Vydělí znaménkovou hodnotu určenou dvojicí registrů A:D prvním operandem. Do registru A uloží podíl, do registru D zbytek po dělení.

Fungování

```
IF operand = 0 THEN
  chyba dělení nulou
ELSE
  podíl := A:D / operand
  zbytek := A:D mod operand
  A := podíl
  D := zbytek
END IF
```

Ovlivněné příznaky Příznaky ZF a SF jsou nastaveny podle podílu. Ostatní nejsou ovlivněny.

IMUL – Znaménkové násobení

Varianty `IMUL R`

Popis Vynásobí znaménkovou hodnotu v registru A prvním operandem. Do dvojice registrů D:A uloží součin tak, že v registru A bude spodních 32 bitů součinu a v registru D bude horních 32 bitů.

Fungování `D:A := A * operand`

Ovlivněné příznaky Příznaky ZF, SF a OF jsou nastaveny podle součinu. Ostatní nejsou ovlivněny.

INC – Zvětšení o 1

Varianty `INC R`

Popis Přičte k cílovému operandu jedničku.

Fungování `operand := operand + 1`

Ovlivněné příznaky Příznaky ZF, SF, CF a OF jsou nastaveny podle výsledku.

INT – Vyvolání přerušení

Varianty `INT I`

Popis Instrukce uloží aktuální pozici v programu na zásobník a vyvolá obsluhu přerušení zadaného operandem. Operand udává číslo vektoru přerušení začínající od 0. Vektor přerušení je index do tabulky přerušení umístěné v paměti na adrese určené registrem IDT.

Fungování

```
velikostUkazatele := 4
PUSH IP
IP := IDT + (operand * velikostUkazatele)
```

Ovlivněné příznaky Neovlivňuje příznaky.

IRET – Návrat z přerušení

Varianty `IRET`

Popis Obnoví ze zásobníku pozici v programu před voláním obsluhy přerušení.

Fungování `POP IP`

Ovlivněné příznaky Neovlivňuje příznaky.

Jcc – Skok při splnění podmínky

Varianty

`JA R` nebo `JA I` – skok, je-li větší (CF=0 a ZF=0)
`JAE R` nebo `JAE I` – skok, je-li větší nebo roven (CF=0)
`JB R` nebo `JB I` – skok, je-li menší (CF=1)
`JBE R` nebo `JBE I` – skok, je-li menší nebo roven (CF=1 nebo ZF=1)
`JE R` nebo `JE I` – skok, je-li rovno (ZF=1)
`JNA R` nebo `JNA I` – skok, není-li větší (CF=1 nebo ZF=1)
`JNAE R` nebo `JNAE I` – skok, není-li větší nebo rovno (CF=1)
`JNB R` nebo `JNB I` – skok, není-li menší (CF=0)
`JNBE R` nebo `JNBE I` – skok, není-li menší nebo rovno (CF=0 a ZF=0)
`JNE R` nebo `JNE I` – skok, není-li rovno (ZF=0)
`JNZ R` nebo `JNZ I` – skok, není-li nula (ZF=0)
`JZ R` nebo `JZ I` – skok, je-li nula (ZF=1)

Popis Pokud je splněna podmínka, provede skok na adresu zadanou v operandu. Tyto instrukce skoku reagují na neznaménkové porovnání.

Fungování

```
IF podmínka THEN
  IP := operand
END IF
```

Ovlivněné příznaky Neovlivňuje příznaky.

JMP – Skok

Varianty `JMP R` nebo `JMP I`

Popis Provede skok na adresu určenou operandem bez uložení návratové informace.

Fungování `IP := operand`

Ovlivněné příznaky Neovlivňuje příznaky.

LOAD – Čtení z paměti

Varianty `LOAD R, R`

Popis Načte z paměti z adresy určené druhým operandem 32-bitové číslo a uloží ho do registru uvedeného v prvním operandu.

Fungování `první := PřečtiInt32ZPaměti(druhý)`

Ovlivněné příznaky Příznaky ZF a SF jsou nastaveny podle načtené hodnoty. Ostatní nejsou ovlivněny.

MOV – Přesun do registru

Varianty `MOV R, R` nebo `MOV R, I`

Popis Zkopíruje hodnotu určenou zdrojovým operandem (druhým) do registru určeného cílovým (prvním) operandem.

Fungování `první := druhý`

Ovlivněné příznaky Příznaky ZF a SF jsou nastaveny podle kopírované hodnoty. Ostatní nejsou ovlivněny.

NOT – Bitová negace

Varianty `NOT R`

Popis Provede negaci bitů registru zadaného operandem.

Fungování `operand := NOT operand`

Ovlivněné příznaky Příznaky ZF a SF jsou nastaveny podle výsledku. Ostatní nejsou ovlivněny.

OR – Bitový součet

Varianty `OR R, R`

Popis Provede bitový součet cílového operandu (první operand) a zdrojového operandu (druhý operand) a výsledek uloží do cílového operandu.

Fungování `cíl := cíl OR zdroj`

Ovlivněné příznaky Příznaky ZF a SF jsou nastaveny podle výsledku. Ostatní nejsou ovlivněny.

POP – Odebrání prvku ze zásobníku

Varianty `POP R`

Popis Načte z vrcholu zásobníku jeden prvek, uloží ho do registru zadaného operandem a poté zvětší ukazatel na vrchol zásobníku (ESP).

Fungování

```
operand := NačtiInt32ZPaměti(ESP)
ESP := ESP + 4
```

Ovlivněné příznaky Neovlivňuje příznaky.

PUSH – Přidání prvku na zásobník

Varianty `PUSH R`

Popis Zmenší ukazatel na vrchol zásobníku (ESP) a poté uloží na vrchol zásobníku hodnotu registru zadaného operandem.

Fungování

```
ESP := ESP - 4
UložInt32DoPaměti(ESP, operand)
```

Ovlivněné příznaky Neovlivňuje příznaky.

RET – Návrat z procedury

Varianty `RET`

Popis Obnoví ze zásobníku pozici v programu před voláním procedury.

Fungování `POP IP`

Ovlivněné příznaky Neovlivňuje příznaky.

ROL – Rotace doleva

Varianty `ROL R, R`

Popis Zrotuje obsah registru zadaného prvním operandem doleva o počet bitů určený hodnotou registru ve druhém operandu.

Fungování

```
temp := první >> (32 - druhý)
první := první << druhý
první := první OR temp
```

Ovlivněné příznaky Příznaky ZF a SF jsou nastaveny podle výsledku. Ostatní nejsou ovlivněny.

ROR – Rotace doprava

Varianty `ROR R, R`

Popis Zrotuje obsah registru zadaného prvním operandem doprava o počet bitů určený hodnotou registru ve druhém operandu.

Fungování

```
temp := první << (32 - druhý)
první := první >> druhý
první := první OR temp
```

Ovlivněné příznaky Příznaky ZF a SF jsou nastaveny podle výsledku. Ostatní nejsou ovlivněny.

SBB – Rozdíl s přenosem

Varianty `SBB R, R`

Popis Odečte od cílového operandu (první operand) zdrojový operand (druhý operand) a příznak přenosu (CF) a výsledek uloží do cílového operandu. Instrukce SBB nerozlišuje znaménkové a neznaménkové hodnoty. Obvykle se používá při odčítání větších než 32-bitových čísel.

Fungování `cíl := cíl - zdroj - CF`

Ovlivněné příznaky Příznaky ZF, SF, CF a OF jsou nastaveny podle výsledku.

SHL – Posun doleva

Varianty `SHL R, R`

Popis Posune obsah registru zadaného prvním operandem doleva o počet bitů určený hodnotou registru ve druhém operandu. Zprava bude registr doplněn nulovými bity.

Fungování `první := první << druhý`

Ovlivněné příznaky Příznaky ZF a SF jsou nastaveny podle výsledku. Ostatní nejsou ovlivněny.

SHR – Posun doprava

Varianty `SHR R, R`

Popis Posune obsah registru zadaného prvním operandem doprava o počet bitů určený hodnotou registru ve druhém operandu. Zleva bude registr doplněn nulovými bity.

Fungování `první := první >> druhý`

Ovlivněné příznaky Příznaky ZF a SF jsou nastaveny podle výsledku. Ostatní nejsou ovlivněny.

STORE – Zápis do paměti

Varianty `STORE R, R`

Popis Uloží do paměti na adresu určenou prvním operandem 32-bitové číslo z registru uvedeného v druhém operandu.

Fungování `UložInt32DoPaměti(první, druhý)`

Ovlivněné příznaky Neovlivňuje příznaky.

SUB – Rozdíl

Varianty `SUB R, R`

Popis Odečte od cílového operandu (první operand) zdrojový operand (druhý operand) a rozdíl uloží do cílového operandu. Instrukce SUB nerozlišuje znaménkové a neznaménkové hodnoty.

Fungování `cíl := cíl - zdroj`

Ovlivněné příznaky Příznaky ZF, SF, CF a OF jsou nastaveny podle rozdílu.

XOR – Výlučný bitový součet

Varianty `XOR R, R`

Popis Provede výlučný bitový součet cílového operandu (první operand) a zdrojového operandu (druhý operand) a výsledek uloží do cílového operandu.

Fungování `cíl := cíl XOR zdroj`

Ovlivněné příznaky Příznaky ZF a SF jsou nastaveny podle výsledku. Ostatní nejsou ovlivněny.

3.7. Cache

Procesor může volitelně využívat jednoúrovňovou cache. Cache má obvykle velikost 200 bytů. Rychlost čtení z cache je srovnatelná se čtením z paměti a zápis je 2x rychlejší.

Cache funguje jako asociativní paměť – pro každou uloženou adresu uchovává hodnotu a její stav. Stav může nabývat jednu ze tří hodnot:

Prázdné Buňka cache je volná.

Aktuální V buňce je hodnota stejná jako v paměti.

Není zapsáno V buňce je změněná hodnota, která ještě nebyla uložena do paměti.

3.8. Superskalární architektura

Simulovaný procesor podporuje superskalární provádění instrukcí pomocí dvou výpočetních jednotek. To znamená, že může teoreticky provádět až dvě instrukce současně[2]. Protože je prováděný program psán s tím, že bude vykonáván postupně instrukci po instrukci, musí procesor zajistit, aby to navenek působilo, že je provádění sekvenční.

Problémem, který je potřeba vyřešit, je volba instrukcí, které lze provést paralelně. Po sobě jdoucí instrukce často pracují se stejnými daty, a proto je není možné spustit souběžně, protože mohly použít nekorektní data. Uvažujme následující příklad:

```
mov a, 0
inc a
```

[2]Tato technika se nazývá ILP (instruction level parallelism, paralelizmus na úrovni instrukcí.

Pokud bychom tyto dvě instrukce spustili souběžně, obsah registru `a` by byl po jejich provedení nedefinovaný.

Je tedy nutné sledovat vstupní i výstupní data každé instrukce a na základě jejich porovnání rozhodnout o jejich možné paralelizaci. Data mohou být dvojího druhu:

- Registry včetně `ip`, `flags` apod.
- Paměť (každý jednotlivý bajt zvlášť – v simulovaném procesoru jsem se ale rozhodla v této situaci pro zjednodušení brát paměť jako jeden celek).

Kolizi zjistíme tak, že vytvoříme průniky mezi množinami vstupních a výstupních dat pro dané dvě instrukce. Je-li některý z průniků neprázdný, mohlo dojít k těmto situacím:

RR Prostředek (registr nebo paměť) je oběma instrukcemi čten. V tom případě je lze bezpečně paralelizovat.

RW, WR Prostředek je jednou instrukcí čten, druhou zapisován. Paralelizace není možná (tato situace nastala v příkladu výše).

WW Prostředek je oběma instrukcemi zapisován. Zápis do paměti se paralelizovat nepovolí, protože se nerozlišují její jednotlivé buňky. Paralelní zapisování do jednoho registru lze povolit, pokud není důležité, která ze zapisovaných hodnot dostane přednost – tj. v situaci, kdy hodnotu poté nikdo nečte.

Je důležité si uvědomit, že registr `flags` je výstupním registrem drtivé většiny instrukcí. Proto je v simulovaném procesoru kolize typu WW častým jevem.

Případ s dvojitým zápisem si demonstrujeme na příkladech:

```
mov a, 0
mov b, 1
jz skok
```

Dvě instrukce `mov` nelze provést souběžně, protože obě zapisují do registru `flags`, jeho obsah je proto nepředvídatelný a stejně tak nepředvídatelný bude i následující skok.

Naproti tomu zde první dvě instrukce `mov` mohou proběhnout paralelně, protože obsah registru `flags` přepíše třetí instrukce:

```
mov a, 0
mov b, 1
dec c
jz skok
```

Protože ověření paralelizovatelnosti častých konfliktů WW vyžaduje analýzu následujícího kódu, rozhodla jsem se nezatěžovat tímto úkolem procesor a místo toho upravit překladač tak, aby do přeloženého programu přidal informace o možnostech paralelizace jednotlivých instrukcí. Procesor už pak závislosti mezi instrukcemi nemusí prověřovat. Popis informací generovaných překladačem je v části 4.2.

3.9. Pipelining

Další způsob zvýšení počtu instrukcí, které lze dokončit v průběhu jednoho cyklu procesoru, je rozložení vykonávání instrukce do více částí, tzv. pipelining.

Pipelining jsem implementovala nad rámec zadání práce. Se superskalárním vykonáváním jsem ho nekombinovala – procesor běží buď v superskalárním režimu, nebo s pipeliningem. Použitý pipelining rozděluje zpracování instrukce do tří částí: načtení, dekódování a provedení instrukce. Pro predikci skoků se používá dynamická strategie.

3.10. Simulace času

Abychom mohli porovnávat rychlost práce programů, počítá simulovaný procesor virtuální čas měřený v bezrozměrných jednotkách. Doby trvání jednotlivých primitivních operací jsou tyto (počet tiků se pohybuje v jednotkách):

Operace	Doba trvání
Čtení paměti	5
Zápis paměti	10
Čtení cache	4
Zápis cache	4
Čtení registru	1
Zápis registru	2
Čtení instrukce z paměti	40
Dekódování instrukce	10
Provádění instrukce	10 * počet tiků

4. Překladač

V práci jsem implementovala jednoduchý překladač z assembleru do strojového kódu. Existuje ve formě knihovny .NET Frameworku, ale snadno by jej šlo rozšířit pro použití z příkazové řádky.

Vstupem překladače je zdrojový kód programu v assembleru v kódování UTF8 a výstupem jsou (v případě úspěšného překladu) přeložený program v binární formě a ladicí informace, podle nichž lze pro každou instrukci programu dohledat řádek ve zdrojovém kódu, ze kterého vznikla.

V případě neúspěšného překladu překladač vrátí seznam chyb, na které narazil, včetně jejich umístění ve zdrojovém kódu. Překladač se nezastaví na první nalezené chybě – když nějakou nalezne, zaznamená ji a pokračuje dalším řádkem.

Současná verze překladače předpokládá, že bude program nahrán v paměti od adresy `0`. Pro nahrání na jinou adresu by bylo nutné zavést relokační tabulku.

4.1. Zápis zdrojového kódu

Ve zdrojovém kódu programu se píše každý příkaz na samostatný řádek. Kromě instrukcí assembleru se v programu mohou objevit komentáře, návěští, definice symbolů pro čísla a řetězce a také samozřejmě prázdné řádky.

4.1.1. Komentáře

Komentáře jsou řádky začínající středníkem.

```
; příklad komentáře
```

Každý komentář musí být na samostatném řádku; na konec řádku je vkládat nelze.

```
mov a, 1 ; takto nelze
```

4.1.2. Instrukce

Instrukce assembleru se zapisují ve standardním tvaru:

```
název_instrukce [parametr1[, parametr2]]
```

Řádek začíná symbolickým názvem (zkratkou) instrukce, např. `mov`, `idiv`, `jmp` atd. Po mezeře následují parametry oddělené čárkou (1, 2, nebo žádný). Úplný popis všech instrukcí včetně jejich parametrů naleznete v části 3.6.3.

Je-li parametrem registr, uvádí se jeho označení – např. `a`, `b`, `esp` atd.

Přímou hodnotu lze zadat desítkovým nebo šestnáctkovým číslem:

```
; Desítková hodnota
mov a, 255
; Ekvivalentní hexadecimální
mov a, 0xFF
```

Na místo přímé hodnoty lze také napsat název návěští nebo symbolu (viz následující části).

4.1.3. Návěští

Návěští se používají pro větvení, cykly a pro definici podprogramů. Umožňují označit místo v programu, na které lze skočit pomocí instrukce skoku. Definují se zapsáním názvu následovaného dvojtečkou. Odkaz se provádí názvem bez dvojtečky.

```
mov c, 10
cyklus:
dec c
jnz cyklus
```

Při překladu programu jsou všechny výskyty odkazu na návěstí nahrazeny adresou instrukce následující po jeho definici. Na návěští se lze odkázat všude tam, kde má být přímá hodnota. Zde bude například po provedení instrukce `mov` registr `a` obsahovat adresu instrukce.

```
návěští:
mov a, návěstí
```

4.1.4. Symboly

Symboly umožňují definovat globální proměnné uložené v paměti. Zadávají se pomocí příkazu `define`:

```
define počet 20
define název "Žluťoučký kůň"
```

Hodnotou může být číslo (zadané desítkově, nebo hexadecimálně), nebo řetězec. Při definici symbolu se ve výsledném programu vyhradí místo pro uložení hodnoty. Všechny odkazy na něj se při překladu nahradí adresami tohoto místa. Jedná se tedy o ukazatel na hodnotu.

Příklad zvětšení hodnoty symbolu `číslo` z nuly na jedničku. Všimněte si, že instrukce `load` a `store` nepodporují zadání adresy přímou hodnotou. Ukazatel na `číslo` je proto zkopírován do registru `b`.

```
define číslo 0
mov b, číslo
load a, b
inc a
store b, a
```

Je-li hodnotou řetězec, uloží se do paměti po jednotlivých bytech v kódování UTF8. Za jeho konec se vloží nulový byte. Jednotlivé znaky lze v programu měnit, řetězec však nelze prodloužit – mohl by kolidovat s daty uloženými v paměti za ním.

Následuje příklad, který definovaný ASCII řetězec převede z velkých písmen na malá. Protože z paměti lze číst pouze celá 32-bitová čísla, tato implementace vždy „vyřízne“ z čísla potřebný byte. Druhá možnost (efektivnější co do přístupů do paměti, ale složitější) by byla zpracovávat řetězec po 4 znacích.

```
1  define str "TEXT VELKYMY PISMENY"
2
3  ; znak konce řetězce
4  mov e, 0
5  ; znak mezery
6  mov i, 32
7
8  ; maska pro získání bytu ze začátku intu
9  mov g, 0x000000FF
10 ; maska pro získání okolí bytu
11 mov h, g
12 not h
13 ; rozdíl velkých a malých písmen v ˜ASCII
14 mov c, 32
15
16 ; adresa začátku řetězce
17 mov b, str
18
19 smycka:
20   ; načteme int
21   load a, b
22   ; zkopírujem a oříznem okolí
23   mov d, a
24   and d, h
25   ; ořízneme byte
26   and a, g
27
28   ; pokud je 0, konec
29   cmp a, e
```

```
30    je konec
31    ; pokud je mezera, přeskočit
32    cmp a, i
33    je preskocit
34
35    ; převedem na malé
36    add a, c
37
38    ; přidáme okolí
39    or a, d
40    ; zapíšem zpátky
41    store b, a
42
43    preskocit:
44    inc b
45 jmp smycka
46
47 konec:
48 end
```

4.2. Generování tabulky pro paralelizaci

Překladač assembleru se významně podílí na schopnosti procesoru automaticky paralelizovat instrukce. Využívá většího množství času, který má při překladu, než procesor při vykonávání programu, a připravuje tzv. paralelizační tabulku. Pro každou instrukci v programu lze v této tabulce najít blok sousedních instrukcí, které s ní jdou provést paralelně. Je-li instrukce v bloku délky 1, znamená to, že musí být provedena samostatně.

S instrukcemi podmíněného skoku na přímou hodnotu se v tabulce zachází speciálním způsobem. Jsou to totiž instrukce, u kterých lze provést predikci skoku, a procesor v superskalárním režimu spekulativně provede tu větev, která se mu jeví jako pravděpodobnější. Z důvodu této dvojí možné cesty ukládá překladač pro podmíněné skoky na přímou hodnotu bloky dva – první je pro případ, že proběhne skok (blok obsahuje instrukci skoku a následující instrukci na cílové adrese, je-li se skokem paralelizovatelná), druhá pro případ neprovedení skoku.

Na obrázku 3. vidíme paralelizační tabulky následujícího programu:

```
0: mov a, 0
1: mov b, 0
2: mov c, 5
doWhile:
3: push c
;while(c > 0)
```

```
4: dec c
5: jnz doWhile
6: end
```

Instrukce	Délka bloku	Vysvětlení
0	1	Pouze samostatně
1	2	Paralelně s instrukcí 2
3	1	Pouze samostatně
4	1	Pouze samostatně
5	2	Při skoku paralelně s instrukcí 3
5	1	Bez skoku pouze samostatně
6	1	Pouze samostatně

Obrázek 3. Paralelizační tabulka

Překladač vygeneruje tabulku za konec programu. Na začátek programu pak vloží speciální instrukci `ilptable imm`, v jejímž parametru je adresa tabulky.

5. Konzola a operační systém

Do simulátoru jsem implementovala „operační systém“, který poskytuje služby pro výpis do okna textové konzoly. Služby operačního systému se po vzoru systému MS-DOS volají pomocí přerušení 0x21.

Poskytované funkce jsou tři:

Výpis řetězce `f = 1`, `b = adresa řetězce`. Funkce přečte z paměti na adrese uložené v registru `b` řetězec ukončený nulou a vypíše ho do konzoly.

Výpis čísla `f = 2`, `b = číslo`, `h = 0/1`. Funkce vypíše číslo uložené v registru `b`. Je-li hodnota registru `h` nulová, vypíše číslo desítkově, jinak ho vypíše šestnáctkově.

Nový řádek Zapíše do konzoly znak pro nový řádek.

6. Uživatelská příručka

Tato část textu obsahuje popis a práci s aplikací modelující výpočet navrženého procesoru. V aplikaci lze napsat zdrojový kód programu pomocí jazyka assembler a pak nechat procesor, aby tento program provedl. Krok po kroku lze sledovat, výpočet procesoru a v každém kroku si prohlédnout dobu trvání dosavadního výpočtu, obsahy registrů, stav zásobníku a paměti. Pro srovnání lze zvolit, na jakém typu procesoru má být výpočet proveden. V nabídce je jednoduchý procesor, jednoduchý procesor s tříúrovňovým pipelingem nebo superskalární procesor se dvěma výpočetními jednotkami. Pro všechny uvedené typy lze využit cache.

6.1. Instalace a odinstalace

Instalace aplikace se spustí souborem `setup.exe`. Aplikace se automaticky nainstaluje a spustí. Odinstalace probíhá klasicky přes ovládací panely Windows.

6.2. Spuštění aplikace

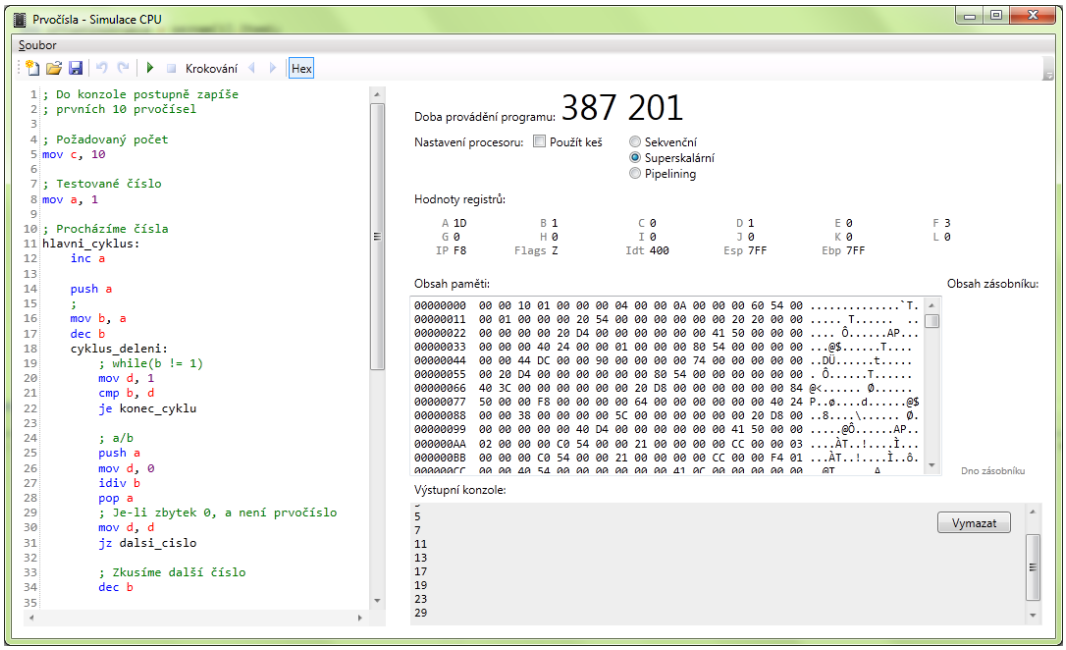
Kliknutím na zástupce `Pavla Bláhová\Simulátor CPU` v nabídce Start se aplikace spustí a otevře se prázdné okno. V levé části pod panelem nástrojů se nachází editor pro psaní programu v assembleru. V pravé části jsou informace o průběhu překladu programu – doba trvání programu, volby pro nastavení procesoru, obsah paměti, obsah zásobníku a výstupní konzole.

Na panelu nástrojů se nachází příkazy na práci se souborem – vytvoření nového souboru, otevření a uložení stávajícího souboru s programem. Dále pak příkazy zpět a vpřed, příkazy pro spuštění výpočtu a pro ovládání krokování a příkaz `Hex`, který provádí převod hodnot v registrech mezi desítkovou a hexadecimální soustavou.

6.3. Editor pro psaní programu v assembleru

Editor na psaní překládaného programu pro lepší orientaci programátora používá barevné odlišení řetězců – zelená barva je vyhrazena pro komentáře, modrá značí instrukci a červená, že se jedná o registr. Každá instrukce patří na nový řádek.

Spuštění, nebo zahájení krokování napsaného programu spustí překlad. Pokud při překladu dojde k chybě, vyvolá se okno `Chyby překladu` (umístěné pod editorem), ve kterém jsou popsány vzniklé chybné instrukce včetně řádku, na kterém se nacházejí. Výběrem chyby (dvojklikem na ni) se v editoru modře zvýrazní chybný řádek. Při zahájení krokování bezchybného programu se v každém kroku překladu žlutě zvýrazní aktuálně načtené instrukce připravené k provedení.



Obrázek 4. Hlavní okno aplikace

6.4. Nastavení procesoru

V části `Nastavení procesoru` lze pomocí přepínače vybrat, jaký typ procesoru bude program provádět. Pokud nic nevybereme, program bude proveden jednoduchým 32-bitový procesorem. Ten můžeme vylepšit na procesor pracující s cache zaškrtnutím volby `Použít keš`, případně vybráním volby `Pipelining`. Nebo se nemusíme jednoduchým procesorem zabývat a program provádět superskalárním procesorem s dvojúrovňovým paralelismem pod volbou `Superskalární`, ten může taktéž využít použití cache volbou `Použít keš`.

6.5. Výpočet a krokování

V panelu nástrojů jsou příkazy pro spuštění a krokování překladu programu. Tlačítko spustit spustí překlad. Pokud proběhne překlad v pořádku, aktualizují se údaje o prováděném programu: `Doba provádění překladu`, `Hodnoty registrů`, `Obsah paměti`, `Obsah zásobníku` a `Výstupní konzole`. Pokud dojde k chybě, vyvolá se okno `Chyby překladu`, které uživatele navede k opravení chyb tak, aby další překlad byl už bezchybný. Příkazem `Krokování` se spustí provedení prvního kroku programu (zpracuje se jedna nebo více instrukcí paralelně – podle typu procesoru) a aktualizují se `údaje o prováděném programu`. V editoru se žlutě zvýrazní instrukce, které jsou načtené a dekódované a budou se provádět v následujícím kroku. Pomocí tlačítek `Krok zpět` a `Krok vpřed` lze listovat jednotlivými kroky výpočtu.

6.6. Popis výstupů

Doba provádění programu Zaznamenává celkový čas dosud zpracované části programu. V případě krokování je to součet časů zpracování všech instrukcí předcházejících žlutě zvýrazněným instrukcím (ty budou provedeny v dalším kroku) a času pro načtení a dekódování těchto žlutě zvýrazněných instrukcí.

Hodnoty registrů Zobrazují aktuální hodnoty všech registrů. Příkaz `Hex` z panelu nástrojů zajišťuje převod hodnot registrů mezi desítkovou a šestnáckovou soustavou.

Obsah paměti Zobrazuje obsah paměti, kterou program využívá.

Obsah zásobníku Zobrazuje obsah zásobníku.

Výstupní konzole Zobrazuje výstup, který je možné díky operačnímu systému zapsat do konzole (texty a čísla). Obasah konzole se vyprazdňuje tlačítkem `Vymazat`.

7. Programátorská dokumentace

Překladač i simulační aplikace jsou napsány v jazyce C# pomocí prostředí Visual Studio 2010. Implementaci jsem rozdělila na 4 logické celky, které jsou v samostatných projektech:

Definice Součásti projektu popisují binární zakódování programů a společné struktury, které používají překladač i simulátor.

Assembler Projekt implementující překladač assembleru.

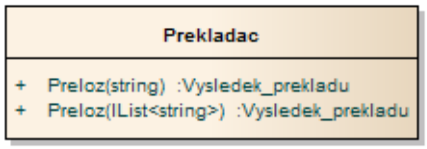
Cpu Projekt simulátoru procesoru a jeho GUI.

Testy Projekt obsahující testy překladače, procesoru a zakódování.

V následujících podkapitolách popíšu projekty *Assembler*, *Cpu* a *Testy*.

7.1. Projekt Assembler

Jádrem projektu jsou třída `Prekladac` a její metody `Preloz`, které na vstupu dostanou zdrojový kód a na výstupu předají přeložený program s ladicími informacemi, jak jsem popsala v předchozích částech.



Obrázek 5. Třída Prekladac

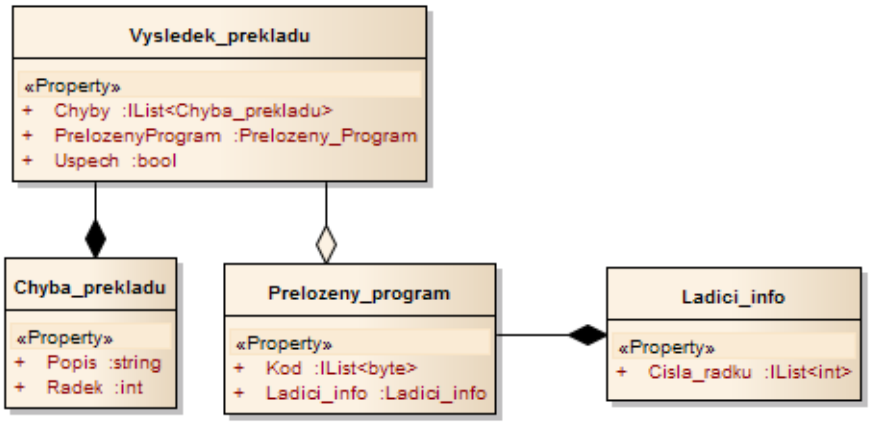
7.1.1. Průběh překladu

Rozdíl mezi zdrojovým kódem assembleru a výslednou posloupností strojových instrukcí není velký – obě posloupnosti příkazů si odpovídají téměř 1:1. Syntaxe assembleru je velmi jednoduchá a není třeba při jeho překladu vytvářet nějaké sémantické výrazové stromy.

Zásadní výhoda assembleru je v tom, že lze zpracovávat každý řádek zvlášť. Vím, že když vezmu jeden řádek zdrojového kódu, bude na něm ucelená informace, kterou mohu zpracovat. U vyšších programovacích jazyků to neplatí.

Stejně tak zjednodušuje řádkové zpracování i zotavení z chyb překladu. Narazí-li překladač vyššího programovacího jazyka na chybu a chce-li přesto pokračovat v překladu, musí nalézt „bezpečné“ místo, ze kterého pokračovat lze. V případě mého překladače je tím místem další řádek.

Zjednodušeně průběh překladu vypadá takto:



Obrázek 6. Třídy překladače

1. Vezmi další řádek zdrojového kódu.
2. Je-li to prázdný řádek nebo komentář, přeskoč.
3. Je-li to návěští nebo deklarace, zapamatuj si je.
4. Jinak je to instrukce; přečti název a parametry.
5. Je-li počet a druh parametrů správný, zakóduj instrukci do výsledku.

Dále popíšu implementaci součástí překladu, které si zaslouží pozornost.

Syntaktická analýza Pro syntaktickou analýzu řádku zdrojového kódu používám analyzátor regulárních výrazů z knihovny .NET Frameworku. Návěští, deklarace, instrukce a jednotlivé druhy parametrů instrukce (číslo, symbol, registr) mají regulární výrazy, které je dovedou rozpoznat.

Zpracování návěští Narazím-li ve zdrojovém kódu na návěští, mohu si uložit jeho offset, abych ho potom mohla doplnit na místo, kde se na něj odkazuje. Může ale dojít k situaci, kdy se na návěští odkazuje dříve, než je definováno. Toto překladač řeší tak, že pro každé návěští udržuje seznam adres, na které je potřeba uložit výsledný offset, až bude znám. Když pak překladač narazí na definici návěští, doplní na požadovaná místa aktuální offset. Je-li seznam na konci překladu neprázdný, některé návěští není definováno.

U deklarací toto není třeba provádět – symbol musí být vždy deklarován před použitím.

7.2. Projekt Cpu

Projekt *Cpu* obsahuje implementaci simulátoru procesoru a paměti, a také GUI aplikace.

7.2.1. Simulace CPU

Simulátor počítače připomíná skládačku. Jádrem simulace je třída *Procesor*. Abych mohla podle potřeby volit komponenty, které k procesoru připojím, použila jsem návrhový vzor strategie a jednotlivé komponenty oddělila pomocí rozhraní:

- `IStrategiePameti` zastřešuje přímý přístup do paměti a kešování.
- `IStrategieProvadeni` označuje komponenty, které řídí způsob provádění instrukcí. Jsou to sekvenční provádění, superskalární provádění a pipeliningové provádění.
- `IIStrategie_odhadu_dalsi_instrukce` slouží superskalárnímu provádění a pipeliningu pro předvídání skoků. Implementace jsou 3: fixní, statická a dynamická strategie.

Třída *Procesor* při svém vytvoření očekává čítač virtuálního času a strategii přístupu do paměti. Abychom jím mohli spustit program, musíme provést následující:

1. Vytvořit čítač.
2. Vytvořit dostatečně velkou paměť (simulace používá výchozí 2KB).
3. Zvolit strategii přístupu do paměti.
4. Vytvořit instanci procesoru.
5. Nastavit umístění tabulky přerušení, pokud ji chceme používat (registr `idt`).
6. Nahrát do paměti program na adresu 0.
7. Spustit procesor – zahájí se provádění programu.

Na obrázku 7. vidíme rozhraní strategie provádění instrukcí.

Pět vlastností z tohoto rozhraní, které jsou typu funkce nebo metody, naplní procesor pomocí svých funkcí. Metodu `Reset()` volá procesor při svém restartu. Metodu `Priprav` volá při zahájení krokování. Superskalární provádění tak může okamžitě přednačíst další instrukce, které mají být provedeny. Metoda `Proved_dalsi_instrukci()` slouží k vlastnímu provádění programu instrukci po instrukci.

Obrázek 7. Strategie provádění

7.2.2. Stavy CPU a paměti

Pro podporu funkce vracení se o krok zpět jsem zavedla mechanismus pro získání snímku celého stavu počítače, který zahrnuje hodnoty registrů a kompletní obsah paměti. Obsah stavu je z vnějšku skrytý, reprezentuje ho pouze prázdné značkovací rozhraní `IStav_pameti` nebo `IStav_procesoru`.

Zaměřme se například na procesor. Ten dále obsahuje metodu `Vrat_stav()`, která zachytí `IStav_procesoru` a vrátí ho. Druhou část procesu tvoří metoda `Obnov_stav(IStav_procesoru stav)`, která převezme zachycený stav a obnoví ho.

7.2.3. GUI aplikace

Grafické uživatelské rozhraní aplikace bylo vytvořeno pomocí knihovny *WPF*. Rozhraní se skládá z jediného okna, které slouží jak pro psaní programu, tak pro prohlížení výsledků činnosti procesoru.

Pro editaci zdrojového kódu aplikace používá volně dostupnou komponentu `Avalon.Edit` vyvinutou v rámci projektu SharpDevelop [2], která podporuje barvení syntaxe kódu a zvýrazňování řádků, které používám pro označení další instrukce během krokování.

7.3. Projekt Testy

V tomto projektu je shromážděno množství unit testů, které ověřují korektnost překladače a procesoru. Každý druh instrukce je zahrnut alespoň do jednoho testu.

Závěr

Hlavním cílem diplomové práce byl návrh a realizace simulování činnosti moderního superskalárního procesoru. K tomutu účelu byla vytvořena aplikace `Simulace CPU`, do které lze zapsat program a krok po kroku demonstrovat jeho provádění. Jak dlouho zpracování programu trvalo a jak přesně se při výpočtu postupovalo, lze vyčíst z doprovodných informacích (obsah registrů, zásobníku, operační paměti...) při krokování. Kvůli srovnání výkonu aplikace `Simulace CPU` nabízí možnost volby vykonání programu jednoduchým 32-bitovým procesorem, nebo jeho superskalární variantou.

Jedním znakem počítačů s Von Neumannovou architekturou je nemožnost paralelního zpracování instrukcí. Výhodou tohoto sekvenčního přístupu je jednodušší, a tedy i rychlejší, napsání programu, než při psaní programů s paralením výpočtem. Nevýhodou je, že program je prováděn vždy jen jednou výpočetní jednotkou a další zůstávají nevyužity. Při návrhu paralelního výpočtu superskalárního procesoru, proto bylo potřeba zařídit, aby nijak nenarušil toto sekvenční provádění. Toho bylo dosaženo spekulativním prováděním instrukcí, při kterém se testuje, jestli při paralelním zpracování více instrukcí najednou nedojde ke změnám výsledku.

Dalším faktorem snižujícím výkon procesoru je rychlost práce s operačním pamětí – je dost pomalá. Ve své práci jsem ji vyřešila možností použití cache. Tato rychlá mezipamět přednačítá data a instrukce, aby se procesor nemusel zdržovat čtením a zápisem z operační paměti. Ne však provádění všech typů programů použití cache zrychlí. Je prospěšná hlavně v programech s často se opakujícími cykly, kde těží z opakovaného zpracování načtených bloků instrukcí.

Hlavním cílem diplomové práce byl návrh a realizace simulování činnosti moderního superskalárního procesoru. K tomutu účelu byla vytvořena aplikace `Simulace CPU`, do které lze zapsat program a krok po kroku demonstrovat jeho provádění. Jak dlouho zpracování programu trvalo a jak přesně se při výpočtu postupovalo, lze vyčíst z doprovodných informacích (obsah registrů, zásobníku, operační paměti...) při krokování. Kvůli srovnání výkonu aplikace `Simulace CPU` nabízí možnost volby vykonání programu jednoduchým 32-bitovým procesorem, nebo jeho superskalární variantou.

Jedním znakem počítačů s Von Neumannovou architekturou je nemožnost paralelního zpracování instrukcí. Výhodou tohoto sekvenčního přístupu je jednodušší, a tedy i rychlejší, napsání programu, než při psaní programů s paralením výpočtem. Nevýhodou je, že program je prováděn vždy jen jednou výpočetní jednotkou a další zůstávají nevyužity. Při návrhu paralelního výpočtu superskalárního procesoru, proto bylo potřeba zařídit, aby nijak nenarušil toto sekvenční provádění. Toho bylo dosaženo spekulativním prováděním instrukcí, při kterém se testuje, jestli při paralelním zpracování více instrukcí najednou nedojde ke změnám výsledku.

Dalším faktorem snižujícím výkon procesoru je rychlost práce s operačním pamětí – je dost pomalá. Ve své práci jsem ji vyřešila možností použití cache. Tato rychlá mezipamět přednačítá data a instrukce, aby se procesor nemusel zdržovat čtením a zápisem z operační paměti. Ne však provádění všech typů programů použití cache zrychlí. Je prospěšná hlavně v programech s často se opakujícími cykly, kde těží z opakovaného zpracování načtených bloků instrukcí.

Conclusions

The main objective of this thesis was a design and an implementation of process simulation for a modern superscalar processor. For this purpose there was created an application called `Simulace CPU` that affords to write a program and to demostrate its execution step by step. How long does the execution take and what si exactly a procedure of its execution, is able to see in attendant informations (values in registrers, content of the stack, content of the memory ...) for each step. The aplication `Simulace CPU` offers an option to choose between standard 32-bits processor and superscalar one because of comparing performances.

One characteristic of computers with Von Neumannovou architekture is nonparallel computation. The advantage of this sequential execution is that the writing a program is simplier and quicker. The disadvantage of this is that the program can be computed only by one unit in time and the others are disused.

In an implemantation of parallel superscalar procesor must be holt sequential computation. It was done because of speculative execution, that tests all instructions to avoid changing the result. The other thing that declines the performance of procesor is a communication with memory – it is slow. We can use cache to improve it. This quick type of memory fetches data and instructions and procesor does not have to slowing down because of reading and writing memory. But it does not hepl us always. That is great for programs full of cycles particularly. It takes advantages of fetching blocks of instructions.

Reference

[1] Von Neumannova architektura [online], dostupné z `http://cs.wikipedia.org/wiki/Von_Neumannova_architektura`. [citováno 2012-30-03 16:28]

[2] Knihovna volně dostupná online `http://wiki.sharpdevelop.net/AvalonEdit.ashx`

[3] Sivarama P. Dandamudi : Guide to RISC Processors For Programmers and Engineers, Springer, 2005, ISBN 0-387-21017-2.

[4] McFarland Grant : Microprocessor Design - A Practical Guide from Design Planning to Manufacturing, The McGraw-Hill Publishing Companies, Inc., 2006, ISBN 0-07-145951-0.

[5] Bláha, Josef. Poskytnutá třída *FileDocument*.

A. Obsah přiloženého CD

V samotném závěru práce je uveden stručný popis obsahu přiloženého CD/DVD, tj. závazné adresářové struktury, důležitých souborů apod.

`bin/`
: Instalátor SETUP.EXE programu.

`doc/`
: Dokumentace práce ve formátu PDF, vytvořená dle závazného stylu KI PřF pro diplomové práce, včetně všech příloh, a všechny soubory nutné pro bezproblémové vygenerování PDF souboru dokumentace (v ZIP archivu), tj. zdrojový text dokumentace, vložené obrázky, apod.

`src/`
: Kompletní zdrojové kódy aplikace Simulace CPU.

`readme.txt`
: Instrukce pro instalaci a spuštění programu Simulace CPU, včetně požadavků pro jeho provoz.

Navíc CD/DVD obsahuje:

`data/`
: Ukázková a testovací data použitá v práci a pro potřeby obhajoby práce.

U veškerých odjinud převzatých materiálů obsažených na CD/DVD jejich zahrnutí dovolují podmínky pro jejich šíření nebo přiložený souhlas držitele copyrightu. Pro materiály, u kterých toto není splněno, je uveden jejich zdroj (webová adresa) v textu dokumentace práce nebo v souboru `readme.txt`.