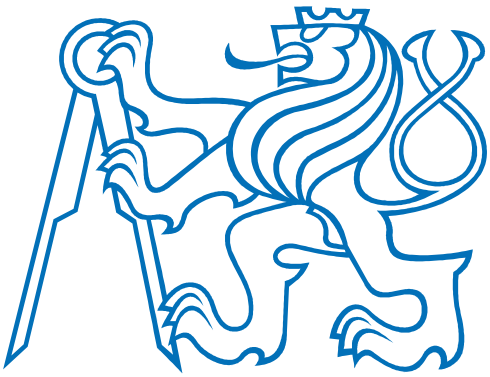
České vysoké učení technické v Praze
Fakulta elektrotechnická
Katedra řídicí techniky



Návrh zásobníkového procesoru pro vestavěné systémy a implementace podpůrných nástrojů

DIPLOMOVÁ PRÁCE
Vypracoval: Jan Procházka
Vedoucí práce: Ing. Pavel Píša PhD.
Rok: 2012

zadani

Prohlášení

Prohlašuji, že jsem svou diplomovou práci vypracoval samostatně a použil jsem pouze podklady (literaturu, projekty, SW atd.) uvedené v přiloženém seznamu.

V Praze dne ..... ….....

Jan Procházka

Poděkování

Na tomto místě bych chtěl poděkovat Ing. Pavlu Píšovi Ph.D. za vedení mé práce, za poskytnutí podkladů k vypracování, a za cenné rady. Také bych chtěl poděkovat rodině za podporu v době psaní této práce.

Jan Procházka

Abstrakt

Obsahem této práce je v teoretické části analýza virtuálních strojů založených jak na registrové, tak na zásobníkové technologii. Hlavním cílem bude navrhnout interpret typu Forth a zásobníkovou mikroprocesorovou architekturu vhodnou pro vestavěné aplikace řídicích systémů jako řízení, například číslicových systémů, s požadavkem na jednoduchost nejen celé architektury, ale i diagnostiky systému. Následně bude vytvořen interpret portovatelný na cizí platformy. Nakonec bude provedena diskuse aplikace synchronní a asynchronní architektury při návrhu mikroprocesoru.

Abstract

Content of this thesis is, in the first, theoretical part, virtual machine analysis based on register as well as stack technology. Main goal will be to design Forth style interpreter and stack microprocessor architecture which would be suitable for embedded applications of control systems for instance digital systems, with architecture simplicity and simple diagnostics requirement. Interpreter, which can be easily ported on other platforms, will be implemented. At the end an application of synchronous and asynchronous architecture for microprocessor will be discussed.

Obsah

Seznam zkratek

ABI – Aplication Binary Interface
API – Aplication Programming Interface
AST – Abstract Syntax Tree
BIOS – Basic Input Output System
CAS – Compare and Swap
CISC – Complex Instruction Set Computer
CFG – Control Flow Graph
CLDC – Connected Limited Device Configuration
CIL – Common Intermediate Language
CLI – Common Language Environment
CMS – Concurrent Mark-sweep
DVM – Dalvik Virtual Machine
EEPROM - Electrically Erasable Programmable Read Only Memory
EFI – Extensible Firmware Interface
GC – garbage collector
HIR – High-level Intermediate Representation
IR – Intermediate Representation
JIT – Just In Time (v souvislosti s kompilací mezikódu)
JVM – Java Virtual Machine
KVM – K Virtual Machine
LIR – Low-level Intermediate Representation
LLJVM – Low Level Java Virtual Machine
LLVM – Low Level Virtual Machine
LZW – Lempel-Ziv-Welch
MTF - Move-to-front
NVM – Non-volatile Memory
PC – program counter
PE - Portable Executable
RAM – Random Access Memory
RPN – reverzní polská notace, Reverse Polish Notation
ROM – Read Only Memory
SIMD – Single Instruction Multiple Data
SJLJ – setjump-longjump
SSA – Static Single Assignment
TLAB – Thread-Local Allocation Buffer
VES – Virtual Execution System
VM – Virtual Machine

Seznam obrázků

Seznam úryvků kódu

Seznam tabulek

Seznam příloh

1 Úvod

Bez digitálních elektronických systémů si lze moderní civilizaci jen těžko představit. Vlastně si dokonce těžko představit celé široké spektrum číslicových systémů, které chod naší civilizace pohánějí. Ať už se jedná o převodníky z / do analogových systémů, jednoduchá logická hradla, klopné obvody, logické obvody, filtry, jednoduché mikročipy sloužící ke zpracovávání vstupů z čidel nebo výstupů do akčních členů, mikroprocesory, programovatelná hradlová pole a celé soustavy těchto prvků propojených mezi sebou. Principů a technologií implementujících všechny ty systémy jsou nepřeberná množství. Postupem času, jak se technologie vyvíjely, některé staré zanikaly, popřípadě se přidávaly k novým kvůli zachování kompatibility ostatních částí systému. Typickým příkladem z prostředí mikroprocesorů je architektura Intel x86. Pokud nepočítáme první mikroprocesory Intel 8086 z roku 1978, potom jako předchůdce současné architektury x86 můžeme označit mikroprocesor Intel 386 vyrobený v roce 1985. V současnosti jeden z nejnovější procesorů od stejné firmy Intel Core i7 stále podporuje technologie 27 let staré, vyvinuté od roku 1985 až do dneška. Z ekonomického hlediska, je podpora na první pohled výhodná, není potřeba přepracovávat obrovské množství systémů, které na těchto technologiích závisí. Na druhou stranu je nutno všechny tyto technologie udržovat, což spotřebovává spoustu zdrojů od energetických až po lidské. Ku příkladu vnější specifikaci procesoru [01] tvoří manuál o 700 stránkách formátu A4 a specifikace instrukční sady [02] obsahuje dalších přes 4100 stránek. Takto obrovské množství informací o součásti je jen velmi těžko udržitelné. Tento příklad byl zvolen, protože je opravdu extrémní, ale podobnosti lze najít téměř všude jinde. Z tohoto problému vychází jedny z požadavků na cíle této práce.

Vestavěných mikročipů existuje nepřeberné množství, důvodem proč navrhovat další architektury je vyzkoušení aplikace technologií virtuálních strojů přímo na hardwarových platformách. Výhodou tohoto přístupu může být snazší diagnostika celého systému, která je nutná v případech, kdy je jedním z hlavních požadavků spolehlivost zařízení, nebo kdy zařízení není fyzicky dostupné pro servis. Pokud bude takováto architektura dobře navržena, může oproti ostatním usnadňovat její programování a tím zvyšovat produktivitu.

1.1 Cíle práce

Základním požadavkem je oprostit se od současných majoritních technologií a navrhnout jednoduchou řídící jednotku, která by ovšem poskytovala větší komfort při programování, a zároveň zachovat celý systém co nejjednodušší, aby se co nejsnáze začleňoval do větších celků, popřípadě z návrhu umožňoval rychlou a snadnou diagnostiku svého vlastního stavu.

Cílem této práce je tedy navrhnout portovatelný interpret a architekturu mikroprocesoru, který by byl vhodný pro vestavěné systémy a použitelný jako řídící jednotka vhodná pro implementaci mechanismů jako číslicová regulace zpracovávající data z jednoduchých čidel nebo ze vstupů, u kterých je potřeba implementovat složitější protokol. Hlavními

požadavky na architekturu jsou

- jednoduchost
- spolehlivost
- flexibilita
- nízká spotřeba
- rychlost

1.2 Struktura práce

Druhá část teoretické kapitoly je věnována virtuálním strojům, protože technologie využívané v těchto aplikacích by měly sloužit k usnadnění programování daných platforem a zároveň poskytují větší flexibilitu oproti hardwarovým implementacím mikroprocesorů.

Ve třetí části se práce bude zabývat popisem návrhu a implementace systému. V poslední kapitole budou shrnuty výsledky a zhodnocen výsledek implementace.

2 Analýza

2.1 Virtuální stroje

V současné době naprostá většina moderních programovacích jazyků využívá pro běh svých aplikací virtuální stroje (VM). Buď implementují vlastní nebo využijí již existující, který se hodí pro vykonávání instrukcí daného jazyka. Dá se říci, že VM je emulátorem hardwarové platformy (která ve skutečnosti nemusí být vůbec hardwarově implementovaná). Z toho vyplývá několik výhod a zároveň několik nedostatků.

Mezi hlavní výhody VM patří snadná přenositelnost aplikací běžící nad virtuálními stroji díky jednotnému spouštěcímu modelu na různých hardwarových platformách. Ideálně by tedy stačilo implementovat VM na novém zařízení a všechen kód pro daný virtuální stroj by se choval stejně jako na předchozí platformě. Využívá se skutečnosti, že reimplementace virtuálního stroje je jednoduchá v porovnání s přepisováním množstvím aplikací, který na něm běží. Další výhodou je přizpůsobení spouštěcího modelu vlastnostem, pro které je virtuální stroj navrhován. Příkladem může být Java Virtual Machine, která je mimo jiné navržena pro práci s objekty a s ohledem na bezpečné provádění programového kódu. Před tím, než je jakýkoliv kód spuštěn, provede se ověření chování aplikace. Obecně vývoj VM je jednoduší než vývoj hardwarové platformy. Jedním důvodem je vrstva, na které zařízení běží. Klasická hardwarová platforma závisí na detekci, vysílání a přepínání napěťových úrovní, které podléhají složitým fyzikálním zákonů, na které se musí brát ohled. Na druhou stranu virtuální stroje fungují na zjednodušeném modelu většinou operačního systému, kde jsou již napěťové úrovně interpretovány většinou binárním kódem a dalšími vyššími abstrakcemi a je zde odstíněno mnoho detailů, na které by se jinak musel brát zřetel. Při nalezení chyby v zařízení lze daleko jednodušeji distribuovat opravenou verzi VM než nový kus mikročipu. V neposlední řadě všechny tyto výhody znamenají ekonomické úspory při vývoji a spravování nejen samotné platformy, ale i aplikací, které na ní běží.

Nevýhodou VM se v určitých situacích stává simulace spouštěcího modelu, která zapříčiňuje pomalejší běh aplikací. Tento nedostatek se mnoho virtuálních strojů snaží maskovat Just-in-time (JIT) kompilací, která za běhu překládá bytekód, popřípadě jinou formu mezikódu, do instrukcí procesoru, na které VM běží. V určitých situacích JIT kompilace a následné optimalizace umožňují rychlejší běh VM aplikace než nativního programu. Tyto metody ovšem značně zvyšují složitost samotného virtuálního stroje a prostředky na jeho implementaci na různých platformách. V souvislosti s rychlostí běhu aplikací na VM je potřeba zmínit i to, že rozdílné hardwarové platformy mohou mít rozdílné spouštěcí modely, takže se implementace VM musí přizpůsobovat danému spouštěcímu modelu, což může mít za následek další snížení výkonu.

2.1.1 Java Virtual Machine

Jedním z nejvyužívanějších virtuálních strojů se stal Java Virtual Machine (JVM). Jedná se o specifikaci instrukcí a rozhraní pro práci a samotného vykonávání Java bytekódu [03]. Primárním a původním programovacím jazykem, který se kompiluje do bytekódu JVM je Java. Existuje spousta dalších jazyků primárně vyvíjených pro JVM:

- Scala [04] – zobecněný objektový model, funkcionální programování
- Clojure [05] – repl, funkcionální programování, lisp syntax, systém agentů pro programování distribuovaných systémů
- Mirah [06] – metaprogramování, ruby syntax
- a další

Mnoho programovacích jazyků bylo na JVM portováno:

- Python – Jython [07]
- Ruby – JRuby[08]
- Erlang – Erjang [09]
- Scheme – JScheme [10]
- Lua – Luaj [11], Jill [12]
- C – NestedVM [13], LLJVM [14]

Jak již bylo zmíněno, specifikace JVM řeší instrukce, chování a rozhraní virtuálního stroje. Dále jsou v práci popsány dvě nejrozšířenější implementace JVM – HotSpot Java Virtual Machine, Dalvik a virtuální stroje pro vestavěné systémy KVM a Squawk. Ani Dalvik ani Squawk technicky vlastně nejsou implementacemi JVM, ale byl zařazeny do této sekce, protože jsou určeny a původně vyvíjeny pro spouštění transformovaného Java bytekódu.

Spouštěcí model

JVM načítá aplikační kód z class souborů, které obsahují tabulky konstant, metod, členských proměnných, jejich atributů a samotný bytekód metod. Běh aplikace obstarává jedno nebo více vláken, která jsou dvojího typu

- standardní
- daemon vlákna

Vlákno je označeno jako daemon, když se předpokládá, že bude stále spuštěno, je aktivní nebo čeká na dokončení vstupů, výstupů nebo vypršení intervalu časovače. Vykonávání instrukcí je ukončeno v momentě, kdy v JVM nezůstane ani jedno standardní vlákno nebo pokud se zavolá metoda `System.exit`.

Z pohledu vlákna se instrukce spouštějí klasicky sekvenčně. Smyčka interpretu by se dala zjednodušeně zapsat následovně:

```
while opcodeIsAvailable(jvmstate)
      opcode = loadOpcode(jvmstate)
      if needsParams opcode
            params = loadParams(jvmstate, opcode)
      else
            params = empty
      jvmstate = exectuteOpcode(jvmstate, opcode, params)
repeat
```

kód 2.1: Zjednodušená smyčka interpretu JVM

Po pozitivní kontrole, zda je k dispozici další instrukce, se instrukce načte. Pokud již žádná další neexistuje, vlákno se ukončí. Všechny instrukce JVM mají délku jednoho bajtu, tímto je také limitován jejich počet. Každá instrukce může, ale také nemusí vyžadovat parametry, které za ní hned následují. Toto rozhodnutí, místo jednotné délky instrukce + operandů, je zdůvodňováno kompaktností takovéhoto uspořádání. Všechny instrukce nebo instrukční slova jsou zarovnána na 1 bajt. Struktury typu `tableswitch` (tabulka skoků) nebo `lookupswitch` (pole skoků) mohou být zarovnány na 4 bajty. Po načtení instrukce a jejích parametrů se instrukce vykoná a začne další iterace cyklu. Mimo samotné vykonání instrukce se zároveň změní registr ukazující do sekvence bytekódu (program counter, PC). Typicky se jeho hodnota zvýší o velikost instrukčního slova. V případě návratů z metod, přímých nebo podmíněných skoků, popřípadě vyhazování nebo zachytávání výjimek se PC mění dle situace.

Jak již bylo zmíněno, JVM podporuje systém výjimek. Po vyvolání výjimky se vyhledá příslušný obslužný kód. Pokud je nalezen, spustí se a následně se provede dokončující část, jestli existuje. Pokud obslužná rutina není nalezena, provede se dokončující část, jestli existuje a metoda se ukončí. Po ukončení metody se výjimka znovu vyvolá a hledání obslužné rutiny pokračuje dále stejným způsobem.

Datové typy

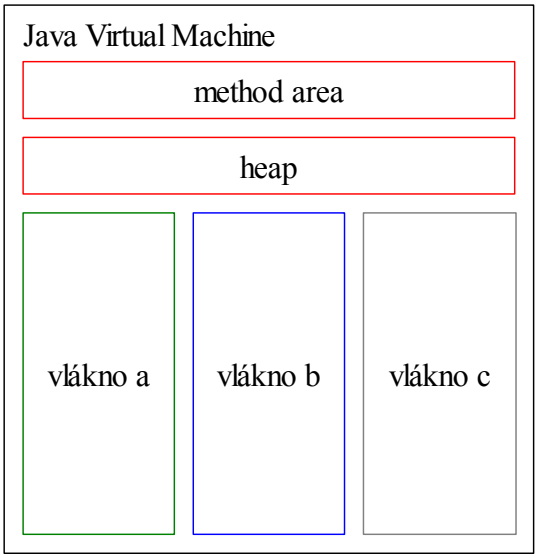
JVM pracuje s 9 základními datovými typy. Většina z nich jsou velmi podobné klasickým datovým typům jaké má například jazyk C.

- `byte` – znaménkové 8 bitové číslo
- `char` – bezznaménkové 16 bitové číslo reprezentující znak v Unicode
- `short` – znaménkové 16 bitové číslo
- `int` – znaménkové 32 bitové číslo
- `float` – desetinné 32 bitové číslo standardu IEEE 754 [15]
- `long` – 64 bitové znaménkové číslo
- `double` – desetinné 64 bitové číslo standardu IEEE 754

- `reference` – ukazatel na objekty, pole nebo hodnota `null` reprezentující prázdný ukazatel
- `return address` – ukazatele na sekvenci JVM bytekódu, obvykle s ním pracuje pouze samotný virtuální stroj a ne programovací jazyk

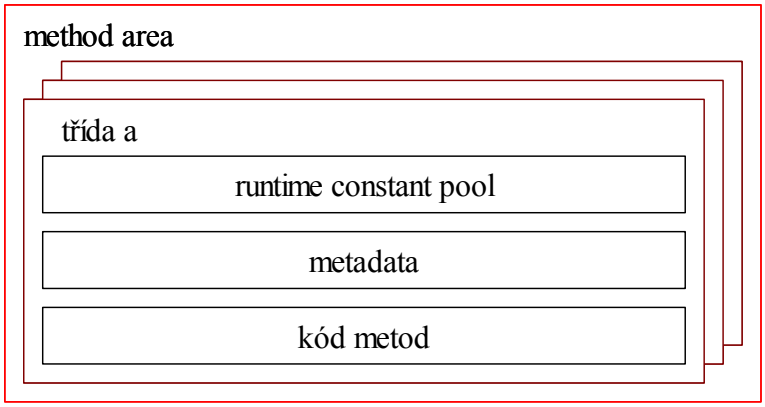
Datové typy jsou většinou staticky kontrolované při kompilaci do bytekódu. Mimo jiné, je to nutné ke kompilaci příslušných instrukcí, které jsou citlivé na datové typy.

Paměťová struktura



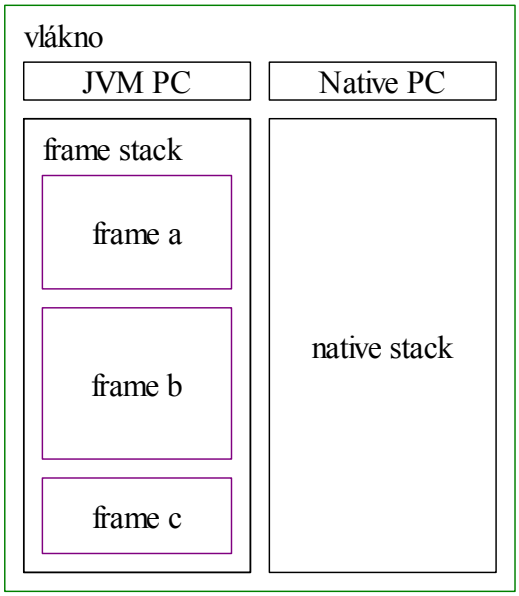
obr. 2.1: Struktura JVM

Paměťovou strukturu JVM lze rozdělit na sdílenou část a nesdílenou. Sdílená je společná pro všechna vlákna běžící ve virtuálním stroji, nesdílená je naopak přiřazena k určitému vláknu a ostatní vlákna k této části nemají přístup. Jednou ze základních částí je tzv. method area, do které jsou ukládány třídy tvořící samotnou aplikaci. Ty obsahují metody, deklarace členských proměnných a definice konstant v tzv. runtime constant pool.



obr. 2.2: Method area

K této paměťové části mají přístup všechna vlákna. Další oblastí je halda, která je opět sdílená mezi vlákny. Na haldě se vyskytují všechny dynamicky alokované struktury jako instance tříd, pole atd. O mazání položek z haldy se stará garbage collector (GC). Pokud na objekt neodkazuje žádná reference nebo je součástí cyklu referencí, na který nic neodkazuje, je označen ke smazání a popřípadě smazán.



obr. 2.3: Vlákno JVM

Mezi nesdílené oblasti patří zásobník JVM. Pro každé vlákno je vytvořen jeden. Do tohoto zásobníku se ukládají stack frames. Zásobníkový rám je vytvořen při volání metody a obsahuje její parametry, lokální proměnné a návratovou hodnotu metody. Voláním instrukce `return` je tento rámec ze zásobníku smazán a dalším instrukcím je k dispozici pouze návratová hodnota. Dalším zásobníkem v JVM je Native method stack. Tento zásobník se využívá pro volání nativních funkcí (funkcí platformy, na které virtuální stroj běží) mimo JVM.

Instrukční soubor

Jak již bylo uvedeno, instrukce JVM mají velikost jeden bajt, takže jejich počet je omezen 256. V současnosti se jich využívá 201 + 3 pro použití virtuálního stroje (debugování a pod.). Přestože je celá specifikace JVM stále ve vývoji, instrukční soubor se kvůli kompatibilitě měnil pouze minimálně. S rozvojem dynamických jazyků, stoupá potřeba efektivního interpretu pro tyto jazyky, proto vznikl Da Vinci Machine Project [16], který je zaměřen na rozšíření instrukčního souboru a ostatních částí JVM právě pro uspokojení výše uvedené poptávky.

JVM je virtuální stroj zásobníkového typu, takže instrukce ukládají a vyzvedávají všechny mezivýsledky do a ze zásobníku. Za instrukcemi může následovat více parametrů, jako například index do tabulky metod, index lokální proměnné a pod. Každá instrukce může

vybrat ze zásobníku pouze určitý typ hodnot. Takže pro sčítání pro typ `int` existuje instrukce `iadd`, pro typ float `fadd`, pro nahrání objektu z lokální proměnné na zásobník `aload` a pod. V následujících ukázkách je pro přehlednost vždy na začátku komentáře uveden v závorkách výpis zásobníku po vykonání instrukce.

```
0 iconst_3  // (3) uloží číslo 3 typu int na zásobník
1 iload_2   // (3 #2) uloží hodnotu lokální proměnné typu int
            // s indexem #2
2 iadd      // (3+#2) vyzvedne ze zásobníku poslední #2 čísla,
            // sečte je a výsledek uloží zpět
```

kód 2.2: Bytekód pro sčítání dvou čísel typu int

Mezi základní instrukce JVM patří práce ze zásobníkem, lokální proměnné, aritmetické a bitové instrukce. Podrobný popis všech instrukcí lze nalézt v [03].

```
0 dload_1          // (#1) vloží do zásobníku hodnotu proměnné #1 typu double
1 iload #15        // (#1 #15) vloží do zásobníku hodnotu  proměnné #15 typu int
2 i2d              // (#1 #15) převede hodnotu z vrcholu zásobníku z int na double
3 dadd             // (#1+#15) vyzvedne ze zásobníku poslední 2 čísla, sečte je
                   // a výsledek uloží zpět
```

kód 2.3: Bytekód pro sčítání dvou čísel různého typu

Očekává se, že každá metoda je zapouzdřena ve své třídě. Při volání metody je tedy nutné uložit na zásobník referenci na instanci třídy a poté parametry volané metody. Po zavolání metody je první lokální proměnná #0 nastavena na objekt, ve kterém je metoda zapouzdřena, a v dalších proměnných jsou uloženy předané parametry.

```
int add(int b, int b)
{
    return a + b;
}

// odpovídající JVM bytekód
 0 iload_1              // (a) vloží hodnotu  proměnné #1 do zásobníku
 1 iload_2              // (a b) vloží hodnotu proměnné #1 do zásobníku
 2 iadd                 // (a+b) sečte
 3 ireturn              // () a vrátí výsledek
```

kód 2.4: Bytekód pro metodu sčítání

Další kategorií instrukcí, které JVM zpracovává jsou řídící instrukce pro přímé, nepodmíněné skoky, volání statických, virtuálních metod a s tím související návraty z metod, porovnávání a instrukce `tableswitch` nebo `lookupswitch`.

```
int n = 0;
while (n < 100)
      n++;

// odpovídající JVM bytekód
 0 iconst_0          // (0) vloží 0 na zásobník
 1 istore_1          // () uloží vrchol zásobníku do lokální proměnné #1
 2 iload_1           // (#1) vloží lokální proměnnou na zásobník
 3 bipush 100        // (#1 100) vloží 100 na zásobník jako int
 5 if_icmpge #14     // () porovná poslední dvě položky na zásobníku,
                     // smaže je a v případě, že je první větší nebo rovna
                     // druhé, skočí na instrukci za cyklem
 8 iinc #1 1         // () zvýší lokální proměnnou #1 o 1
11 goto #2           // () skočí na začátek cyklu
```

kód 2.5: Bytekód pro cyklus

```
obj = new AnObject();
obj.var++;
obj.method();

// odpovídající JVM bytekód
 0 new #1            // (obj) vytvoří a uloží se na zásobník nový objekt
 3 dup               // (obj obj)
 4 invokespecial #4 // (obj obj) zavolá se konstruktor objektu
 7 astore_0          // (obj) uloží se vrchol zásobníku do proměnné #0
 8 aload_0           // (obj obj) na vrchol zásobníku se vloží proměnná #0
 9 dup               // (obj obj obj)
10 getfield #2       // (obj obj var) vezme ze zásobníku objekt a vloží jeho
                     // členskou proměnnou obj.var na zásobník
13 iconst_1          // (obj obj.var 1)
14 iadd              // (obj obj.var+1)
15 putfield #2       // () uloží vrchol zásobníku od obj.var
18 aload_0           // (obj)
19 invokevirtual #5 // () zavolá funkci method
```

kód 2.6: Bytekód pro práci s objekty

Tableswitch a **lookupswitch** jsou instrukce pro efektivní porovnávání proměnné s konstantami. V JVM jsou podporovány pouze konstanty typu **byte**, **int** a **short**. **Tableswitch** je určen pro porovnávání s čísly, která jsou velmi blízko sebe. Interpret tak může jednoduše v konstantním čase (stejně jako když se přistupuje k položce v jednorozměrném poli) zjistit, která větev odpovídá porovnávané proměnné. **Lookupswich** se využívá v případech, kdy hodnoty k porovnání mají od sebe větší vzdálenost, takže by velikost tabulky přesáhla rozumnou mez. Při kompilování by měl kompilátor tabulku instrukce **lookupswitch** setřídit vzestupně. Takže interpret může proměnnou porovnat s danými konstantami v logaritmickém čase, například binárním prohledáváním.

```
switch(n)
{
      case 0:  ...
      case 1:  ...
      case 2:  ...
      default: ...
}

// odpovídající JVM bytekód
 0 iload_1                // (#1)
 1 tableswitch 0 to 2     // () je povoleno porovnávání od 0 do 2
     0: 28                // skočí na adresu 28
     1: 30                // skočí na adresu 30
     2: 32                // skočí na adresu 32
     default: 34          // skočí na adresu 34
```

kód 2.7: Bytekód pro tableswitch

```
switch(n)
{
      case -100:  ...
      case 0:  ...
      case 100:  ...
      default: ...
}

// odpovídající JVM bytekód
 0 iload_1                // (#1)
 1 lookupswitch 3         // () velikost tabulky je 3
     -100: 36             // skočí na adresu 36
     0:    38             // skočí na adresu 38
     100:  40             // skočí na adresu 40
     default: 120         // skočí na adresu 42
```

kód 2.8: Bytekód pro lookupswitch

V JVM lze využít systému zachytávání a vyhazování výjimek. Každá metoda, která chce využít tohoto systému, musí mít přidělenu tabulku výjimek. Tabulka určuje, které oblasti jsou hlídané a v případě výskytu výjimky, na jaké adrese je obsluha ošetření výjimky. Pokud je výjimka vyhozena mimo hlídané oblasti, metoda se ukončí a znovu se vyhodí výjimka v nadřazené volající metodě, kde se postup opakuje.

```
try
{
      this.divZero();
}
catch (Exception e)
{
      this.handleException(e);
}

// odpovídající JVM bytekód
 0 aload_0                // (this) začátek try bloku
 1 invokevirtual #2       // () volání metody this.divZero
 4 goto 13                // () skok za try - catch blok
 5 astore_1               // () začátek catch bloku, na zásobník je
                          // před vykonáním této instrukce uložen vyhozený
                          // objekt, uloží vyhozenou výjimku do proměnné #1
 6 aload_0                // (this)
 7 aload_1                // (this #1)
 8 invokevirtual #4       // () zavolá se obsluha #5, this.handleException
11 return                 // ()
// tabulka výjimek
   From      To     Target       Type
     0        4       7          class Exception
```

kód 2.9: Bytekód pro zachytávání výjimek

Souběžné spouštění úloh a ochrana jejich sdílených prostředků je řešena systémem monitorů. Mezi instrukcemi `monitorenter` a `monitorexit` je zaručeno, že nebude docházet k chybám souběhu pro monitorovaný objekt. Oblast mezi těmito instrukcemi je nutné v případě možného výskytu výjimky ošetřit.

```
void mutateSharedObject (SharedObject shared)
{
       synchronized(shared)
       {
              shared.mutate();
       }
}

// odpovídající JVM bytekód
 0 aload_1                 // (shared)
 1 dup                     // (shared shared)
 2 astore_2                // (shared)
 3 monitorenter            // () uzamkne přístup k objektu na vrcholu zásobníku
 4 aload_1                 // (shared)
 5 invokevirtual #2        // () zavolá shared.mutate
 8 aload_2                 // (shared)
 9 monitorexit             // () odemkne přístup objektu na vrcholu zásobníku
10 goto 18                 // () přeskočí catch blok
13 astore_3                // () uloží výjimku do #3
14 aload_2                 // (shared)
15 monitorexit             // () odemkne přístup k objektu na vrcholu zásobníku
16 aload_3                 // (exception)
17 athrow                  // () znovu vyhodí výjimku, aby ji mohla volající
                           // metoda zpracovat
18 return                  // konec metody
// tabulka výjimek
   from   to  target type
     4    10    13   any
    13    16    13   any
```

kód 2.10: Bytekód pro synchronizaci v JVM

Naprostá většina instrukcí JVM se chová hladově - tedy, že hodnoty, které ze zásobníku přečtou, jsou odstraněny a zpět na zásobník je uložen výsledek vykonávání instrukce. Pro uchování hodnoty na zásobníku delší dobu se využívá instrukcí typu `dup` nebo dočasného úložiště v podobě lokálních proměnných.

Ověřování kódu

Před spuštěním jakéhokoliv kódu v JVM je nutné zkoumaný kód nejdříve ověřit. O ověřování se stará samotný virtuální stroj a má za úkol zajistit základní bezpečnost a konzistenci vykonávané aplikace. Proces probíhá ve 4 fázích.

V první fázi se aplikace nahraje z class souboru. Ověří se, že soubor odpovídá danému formátu a všechny sekce mají přesně odpovídající velikost.

Druhá fáze je zaměřena na kontrolu tříd a runtime constant poolu. Pro každou třídu se ověří, že má přiřazenou nadtřídu (kromě základní třídy Object), že nadtřída není `final` (nesmí se dědit a měnit), a že `final` metody nejsou přepsány v odvozených třídách. V runtime constant poolu je kontrolována správná forma všech položek a platnost jejich jmen.

Nejobtížnější a většinou i časově nejnáročnější částí kontroly je třetí fáze, kdy je prováděna data-flow analýza pro každou metodu. Je zajištěno, že za každou instrukcí následuje správný počet parametrů, a že každá instrukce nalezne odpovídající počet položek v zásobníku s odpovídajícími typy. Do toho lze zahrnout i to, že volané funkce mohou očekávat správné parametry na zásobníku. Ověřují se všechny přesuny kontroly kódy v rámci metody jako jsou skoky a tabulka výjimek. Každá adresa musí ukazovat na začátek instrukce a musí ukazovat do ověřované metody. Pro každou instrukci se ověří, že každá cesta, která k ní vede, zanechá před instrukcí stejný stav zásobníku – tedy stejnou velikost a stejné typy položek. Musí platit, že pokud se přistupuje k jakékoliv lokální proměnné, nesmí být její číslo vyšší než alokovaná velikost pole lokálních proměnných, a v případě čtení, že již bylo do lokální proměnné alespoň jednou zapisováno a nečtou se neinicializovaná data.

Poslední, čtvrtá, část ověřovacího procesu se provádí během prvního spuštění daného kódu metody. Ověří se, zda volaná metoda skutečně existuje. V některých případech je toto nutné provést až za běhu aplikace, protože JVM podporuje dynamické nahrávání kódu a zároveň umožňuje měnit chování kódu za běhu. Dále se ze stejného důvodu ověřuje i oprávněnost volání metod, které mohou být veřejné, soukromé (volat je může pouze zapouzdřující třída a její podtřídy) a chráněné (volat je může pouze zapouzdřující třída).

HotSpot Java Virtual Machine

HotSpot JVM [17] je oficiální implemetací specifikace JVM. Proto a z důvodu největšího rozšíření je uveden v této práci. HotSpot patří do kategorie serverových a desktopových aplikací. Obsahuje JIT kompilátory a několik odlišných typů GC (viz níže). Tento virtuální stroj lze rozdělit na části

- jádro VM
- serverová část – nahrávač, JIT kompilátor
- klientská část – nahrávač, JIT kompilátor
- garbage kolektor(y)

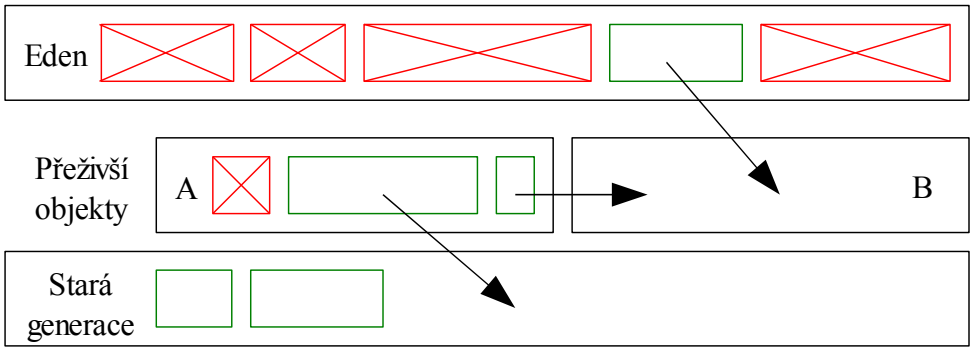
Jádro VM starající se o vykonávání bytekódu je společné pro každý běh virtuálního stroje. HotSpot ovšem provádí rozdílné optimalizace v závislosti, v jakém módu běží. Lze vybrat serverový nebo klientský. V serverovém módu se předpokládá, že virtuální stroj a interpretované aplikace budou běžet delší čas. Proto se provádí daleko agresivnější optimalizace, které se projeví na delším spouštění aplikace, ale na druhou stranu její běh bude díky její hlubší analýze rychlejší. Naopak v klientském módu je preferován rychlý start a prováděny jsou pouze méně časově náročné optimalizace. S různým využitím VM souvisí i různá výchozí nastavení, jako velikost haldy, typ GC a pod.

Spuštění JIT kompilace se řídí časem stráveným v dané metodě. Pokud je metoda volána vícekrát nebo obsahuje dlouhý cyklus, je označena jako kandidát na optimalizaci. Protože JVM podporuje dynamické nahrávání kódu, je nutné buď vyhnout se některým agresivnějším optimalizacím, nebo implementovat zpětnou transformaci. HotSpot využívá

druhého způsobu a při dynamickém nahrání kódu původní aplikaci deoptimalizuje a novou, rozšířenou, transformuje do efektivnějšího nativního kódu. Klientský JIT kompilátor provádí převod bytekódu do High-level Intermediate Representation (HIR) v podobě Static Single Asignment (SSA), který je platformě nezávislý. V této fázi se uskutečňuje vkládání metod. Následně se HIR transformuje do Low-level Intermediate Representation (LIR), která už je platformě závislá, a na které se alokují registry procesoru. Nakonec proběhne peephole optimalizace a výsledkem je strojový kód. V serverové verzi JIT kompilátoru lze nalézt podobný proces jenom s tím rozdílem, že v HIR jsou kromě vkládání metod další optimalizace. Mezi optimalizace v HIR patří hledání a eliminace invariantů smyček, odstranění stejných podvýrazů, propagace konstant, číslování globálních proměnných [18], které doplňuje eliminaci podvýrazů, rozvinutí cyklu, a optimalizace typické pro objektová prostředí jako kontrola nulové reference a kontrola rozsahu. V LIR se navíc provádí rozvrhování instrukcí pro daný procesor.

HotSpot nabízí 4 druhy garbage kolektorů [19]. Všechny se vyznačují tím, že jsou generační a přesné. Tzn. když je objekt již nedosažitelný, je garantováno, že bude hned nebo za delší časový úsek smazán.

Jedním z garbage kolektorů je sekvenční generační kopírující kolektor. Stejně jako všechny generační kolektory využívá faktu, že alokace a uvolňování paměti (objektů) není nahodilé. Většina objektů nepřežije několik málo návratů z metod a je možné je téměř hned odstranit. Schéma kolektoru je načrtnuto na obr. 2.4.



obr. 2.4: Běh GC pro mladou generaci

Všechny objekty jsou ze začátku umístěny do oblasti zvané *Eden*, která když přeteče, provede se úklid nedosažitelných objektů, kterých je v ní většina. Ostatní se překopírují do oblasti přeživších objektů *B*. Ty, které se do *B* nevejdou, putují rovnou do staré generace. Objekty v *A*, které jsou živé a relativně mladé se kopírují do oblasti *B*. Ty, které se nevejdou nebo jsou relativně staré, opět putují přímo do staré generace. Nedostupné objekty v *Edenu* a *A* se smažou. Nakonec se prohodí *A* s *B*. Alokace objektů je velmi rychlá protože se nové objekty jednoduše přidávají na zásobník, takže stačí udržovat ukazatel na volné místo na zásobníku. Paralelní alokace je řešena tím, že si každé vlákno rozdělí oblast *Edenu* do Thread-Local Allocation Buffers (TLAB), a tím se alokace provádí většinou bez potřeby zamykání sdílené paměti.

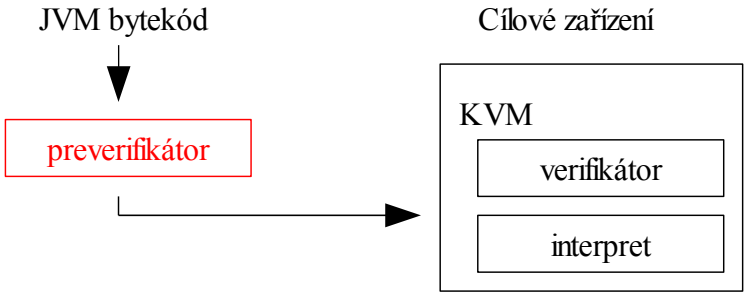
Pro uvolňování objektů staré generace slouží sekvenční algoritmus, typu mark-compact. Je aktivován v případě, že dojde dostupná paměť tím, že objekty ve staré generaci neustále přibývají. Po spuštění algoritmu se projde celý graf objektů, odstraní se nedosažitelné objekty a případně zbylé objekty sesune k sobě, aby se zabránilo fragmentaci paměti. Druhým typem GC je paralelní kolektor. Pracuje na stejném principu jako předchozí, jen s tím rozdílem, že mladé a přeživší generace se zpracovávají paralelně, stará generace se provádí stejným sekvenčím algoritmem. Dalším GC je paralelní kolektor staré generace, který nahrazuje sériový, pokud je v systému, na kterém JVM běží více procesorových jader. Stará generace je rozdělena na části přiřazené jednotlivým vláknům GC. V každé části se provede podobný algoritmus jako u sekvenčního GC staré generace. Následně je pro každou část vypočítán koeficient fragmentace a v poslední, tentokrát sériové fázi, se všechny části s dostatečně velkým koeficientem fragmentace sesunou k sobě. Posledním typem GC je souběžný mark-sweep kolektor (concurrent mark-sweep - CMS). Nevýhodou předchozích typů byla relativně dlouhá pauza, při které se musela interpretovaná aplikace zastavit, a musely se projít všechny staré objekty. Tento GC je zaměřen pro použití v případech, kdy je důležitější nízká odezva více než paměťová propustnost sběru. Dalšími nevýhodami jsou větší nároky a velikost haldy a pomalejší alokace objektů. Mladá generace je zpracovávána stejně jako u druhého typ GC, tedy paralelně. Čištění staré generace je prováděno pokud možno paralelně s během aplikace a inkrementálně, takže jedna dlouhá pauza je rozdělena do více menších. Tento GC nesesouvá fragmentované části paměti k sobě, takže alokaci nelze provádět jednoduchým způsobem, pouze ukazatelem na volné místo, ale je nutné udržovat seznam volných částí paměti. Tam, kde je to možné, se fragmentaci snaží zabránit spojováním položek v seznamu volné paměti a tím, že sleduje nejčastější velikosti objektů.

K Virtual Machine

Druhou oficiální implementací JVM je Kilobyte Virtual Machine (KVM) [20]. KVM se zaměřuje na běh aplikací na vestavěných systémech s omezenými hardwarovými možnostmi. Uvádí se [20], že minimální požadavky pro provoz virtuálního stroje jsou 128 kB paměti pro samotnou VM a dalších 128 kB pro haldu využívanou knihovnami a aplikacemi nad KVM. Virtuální stroj má podobnou architekturu jako HotSpot VM, takže se práce dále zabývá pouze odlištnostmi, konkrétně preverifikací (viz níže). V oficiální dokumentaci je řečeno, že KVM je implementací kompletní. Není zřejmé, co je pod termínem kompletní myšleno, ale kompletní implementací specifikace JVM není, jak je dále v [20] naznačeno. KVM se řídí specifikací Connected Limited Device Configuration (CLDC) [22], která se odlišuje od JVM v těchto bodech:

- není nutná podpora aritmetiky v plovoucí desetinné čárce
- není podpora pro daemon vlákna
- zavádí se krok preverifikace, a díky tomu je zjednodušena samotná verifikace při nahrávání tříd do JVM
- a další

Pro téma této diplomové práce je zajímavý pouze třetí bod – preverifikace – znázorněný na obr. 2.5. Jak bylo popsáno v kapitole 2.1.1 nejnáročnější fáze ověřování je třetí fáze, kde se provádí analýza datového toku. V systémech s omezenými prostředky je tento krok nežádoucí kvůli nárokům na výpočetní a paměťové zdroje. CLDC zavádí pro každou metodu, která obsahuje alespoň jednu řídící instrukci, *StackMap*, která je vypočtena v průběhu preverifikace [21].



obr. 2.5: Proces nahrávání třídy v KVM

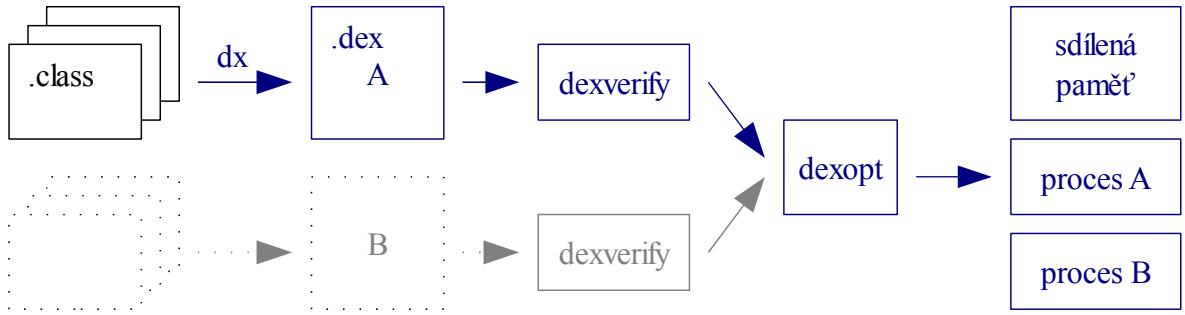
StackMap je pole, do kterého se uloží informace o datovém typu možného budoucího stavu zásobníku pro každou instrukci metody. Při procesu ověřování v KVM pak stačí pouze lineárně projít instrukce metody a u každé ověřit, jestli je na zásobníku odpovídající počet a typ operandů.

Jako správce paměti slouží jednoduchý *mark-and-sweep* GC, který nesesouvá přeživší objekty, což má za následek fragmentaci paměti.

Dalvik VM

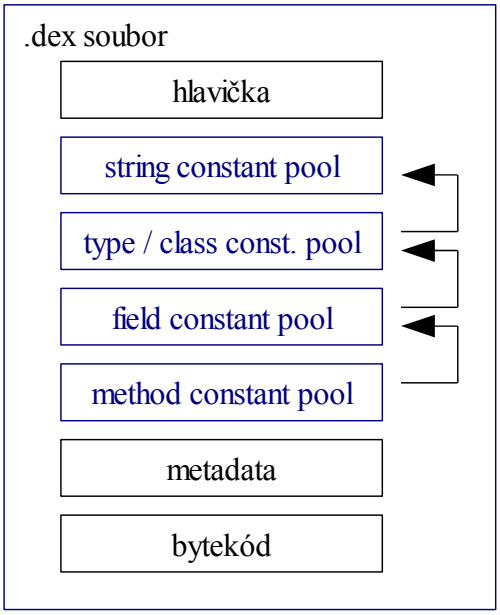
Dalvik (DVM) [23] je virtuální stroj vyvíjený společností Google Inc. Cílem implementace je poskytnout alternativu pro Java aplikace na mobilních zařízení. Tomuto cíli byly podřízeny požadavky na systém a z nich vyplývající omezení. Cílovým zařízením je tedy přístroj s operační pamětí minimálně 64MB operační paměti RAM bez možnosti swapování, 256MB permanentní paměti a napájení baterkou.

Většina mobilních zařízení je poháněna registrovými procesory, proto byl Dalvik navržen jako registrový virtuální stroj, aby interpret více korespondoval s hardwarovou architekturou. Je tedy nutné zásobníkový Java bytekód transformovat do registrového kódu. Při té příležitosti provádí Dalvik mnoho optimalizací. Schéma obr. 2.6 shrnuje přehled transformace.



obr. 2.6: Schéma transformace Dalvik VM

Nejprve se všechny *.class* soubory aplikace převedou do jednoho souboru v *.dex* formátu, který je kompaktnější a obsahuje už převedený registrový bytekód. Následně nahrávaný kód a data do paměti rozdělí na sdílenou část a nesdílenou, která běží ve svém procesu. Každý takovýto proces obsahuje i instanci samotného virtuálního stroje.



obr. 2.7: Struktura .dex souboru

Struktura *.dex* souboru znázorněná na obr. 2.7 se liší od *.class* formátu tím, že kombinuje data a kód všech tříd aplikace do jednoho balíku. Dále je původní *runtime constant pool* rozdělen na více částí. První z nich, *string constant pool*, obsahuje všechny řetězce, na které se pak odkazují následující části jako *type / class constant pool* ve formě indexů. Těmito dvěma fakty se významně redukuje místo, potřebné k uložení všech potřebných informací. Podle [24] je průměrně velikost zkomprimovaných tříd aplikace obdobná jako velikost nekomprimovaného *.dex* souboru.

Proces spouštění transformovaných aplikací je následující. Při spuštění operačního systému se do paměti zavede proces zvaný Zygote, který nahraje standardní knihovnu a další, u kterých je pravděpodobné, že většina aplikací bude využívat. Následně při spouštění dalších procesů uživatelských programu se provede *fork* ze Zygote a do nového procesu se nahraje

kód spouštěné aplikace případně s dalšími potřebnými knihovnami. Pokud nový proces potřebuje zapisovat do sdílené paměti, oblast se ze sdílené části překopíruje do paměti procesu. Využívá se tedy *copy-on-write* principu.

Zajímavě je řešena alokace a uvolňování nepotřebné paměti. Protože Dalvik obsahuje sdílené části, je nutné, aby garbage collector měl přehled o referencích na sdílené objekty z různých procesů. V celém prostředí tedy běží pouze jedna instance GC, která ukládá *mark* bity označující objekty k odstranění do sdílené paměti mimo samotné instance.

```
macro dispatch
      op = read code[pc]
      op_address = address_of op_a + 16 * op
      pc = pc + 1
      jump op_address

op_a:
      // 16 bytová implementace instrukce op_a
      dispatch
op_b:
      // 16 bajtová implementace instrukce op_b - 1. část
      jump op_b_rest:
op_c:
      // 16 bajtová implementace instrukce op_c
      dispatch
ob_b_rest:
      // 2. část instrukce op_b
      dispatch
```

kód 2.11: Smyčka interpretu DVM

Smyčka interpretu virtuálního stroje je optimalizována tak, aby při spuštění každé instrukce bylo čteno z paměti co nejméně (přístup do necachované paměti je většinou samozřejmě řádově pomalejší než například jednotaktové aritmetické instrukce).

Ověřování a optimalizace samotného bytekódu probíhá obdobně jako u HotSpot VM s tím rozdílem, že jsou procesy přizpůsobeny registrové architektuře a některé výsledky verifikace a optimalizace jsou cachovány na permanentním úložišti.

Samotný bytekód a volací konvence se snaží co nejvíce přiblížit spouštěcímu modelu klasických platforem jako ARM nebo x86. Toto rozhodnutí je odůvodňováno lepším mapováním instrukcí na cílovou platformu a jednodušší implementací JIT kompilátoru na těchto strojích. Při volání metody se na zásobníku vytvoří rámec obsahující parametry metody a všechny lokální proměnné. Velikost instrukce DVM bytekódu je 16 bitů. Každá může adresovat 16 registrů, u určitých typů i 256. Dále existují speciální instrukce (například pro optimalizovaný přístup k členským proměnným třídy nebo pro volání metod), které se vyskytují pouze v optimalizovaném bytekódu, který není platformě nezávislý. V několika následujících úryvcích jsou uvedeny příklady disassemblovaného bajtkódu. Pro porovnání se jedná o stejné kódy jako v kapitole popisující JVM instrukce.

```
int add(int b, int b)
{
    return a + b;
}

      // v1: this
      // v2: param[0]
      // v3: param[1]
      add-int       v0,v2,v3      // součet v2 a v3 uloží do v0
      return        v0            // vrátí v0
```

kód 2.12: DVM bytekód pro scíčtání dvou čísel typu int

```
int n = 0;
while (n < 100)
      n++;

      const/4       v0,0          // uloží do v0 0
label1:

      const/16      v1,100        // uloží do v1 100
      if-ge         v0,v1,label2  // pokud je v0 větší nebo rovno
                                  // v1, skočí na konec cyklu
      add-int/lit8  v0,v0,1       // přičte 1 k v0 a uloží
                                  // nazpět do v0
      goto          label1        // skok na začátek cyklu
label2:
```

kód 2.13: DVM bytekód smyčky

```
obj = new AnObject();
obj.var++;
obj.method();

      // v2: this
      new-instance        v0, AnObject              // vytvoří nový objekt
                                                    // a uloží ho do v0
      invoke-direct       {v0}, AnObject/<init>     // zavolá konstruktor obj
      iget                v1,v0, AnObject.var I     // uloží obj.var do v1
      add-int/lit8        v1,v1,1                   // přičte 1 do v1
      iput                v1,v0,AnObject.var I      // uloží obj.var do v1
      invoke-virtual      {v0},AnObject/method      // zavolá obj.method
```

kód 2.14: DVM bytekód pro práci s objekty

```
try
{
	this.divZero();
}
catch (Exception e)
{
	this.handleException(e);
}

	// v1: this
	// tabulka výjimek:
	from   to     using  type
	label1 label2 label3 java/lang/Exception
label1:
	invoke-virtual	{v1}, divZero		// zavolá this.divZero
label2:
	return-void				// vyskočí z metody
label3:
	move-exception	v0			// uloží vyvolanou výjimku do v0
	invoke-virtual	{v1,v0}, handleException
					// zavolá this.handleException(v0)
	goto		label2			// skočí na konec metody
```

kód 2.15: DVM bytekód pro zachytávání výjimek

```
void mutateSharedObject (SharedObject shared)
{
	synchronized(shared)
	{
		shared.mutate();
	}
}

	// v1: this
	// v2: param[0]
	// tabulka výjimek
	from   to     using  type
	label1 label3 label2 java/lang/Exception
	monitor-enter	v2		// uzamkne objekt v2
label1:
	invoke-virtual	{v2}, mutate	// zavolá v2.mutate()
	monitor-exit	v2		// odemkne objekt v2
	return-void			// vyskočí z funkce
label2:
	move-exception	v0		// uloží zachycenou výjimku
					// do v0
	monitor-exit	v2		// odemkne v2
label3:
	throw		v0		// znovu vyhodí výjimku v0
```

kód 2.16: DVM bytekód pro vícevláknovou synchronizaci

Ukázky **tableswitch** a **lookupswitch** v DVM byly vynechány, protože se jedná o velmi

podobné principy jen s rozdílem v názvu instrukcí – **packed-switch** a **sparse-switch**. Zcela novým typem instrukcí jsou instrukce pro inicializaci polí. Pokud je v JVM potřeba inicializovat pole, musí JVM pole vytvořit a pak postupně každou buňku zaplnit pomocí instrukcí (**a|b|c|d|f|i|l|s**)**astore**, což neúměrně zvyšuje velikost výsledného bytekódu. DVM využívá instrukci **fill-array-data**, která spolu s daty zabírá pouze minimální prostor.

```
public int array()
{
	int[] data = {3, 1, 4, 1, 5, 9};
	return data[3];
}

	// v2: this
	const/4			v0,6		// uloží velikost pole do v0
	new-array		v0,v0,[I	// vytvoří pole a uloží ho do v0
	fill-array-data		v0,label1	// naplní pole daty
	const/4			v1,3		// uloží do v1 3
	aget			v0,v0,v1	// nahraje do v0 data z pole na indexu v1
	return			v0		// vrátí v0 z metody
label1:
	data-array
		0x03, 0x00, 0x00, 0x00	// #0
		0x01, 0x00, 0x00, 0x00	// #1
		0x04, 0x00, 0x00, 0x00	// #2
		0x01, 0x00, 0x00, 0x00	// #3
		0x05, 0x00, 0x00, 0x00	// #4
		0x09, 0x00, 0x00, 0x00	// #5
	end data-array
```

kód 2.17: Instrukce fill-array-data v DVM

Squawk

Cílem vývoje Squawk [25] byl běh JVM aplikací na systémech s 32 bitovými procesory s 8kB RAM, 32kB EEPROM nebo Flash pamětí a 160kB ROM. Je celý implementován v Javě, pouze kritické součásti se překládají do C a následně do strojového kódu.

Obdobně jako KVM, Squawk využívá translátoru, který přeloží *class* soubory do tzv. *suite* archivu, který se následně nahrává na cílové zařízení, kde běží interpret. Aby autoři projektu dosáhli stanovených minimálních požadavků, bylo třeba přidat omezení při generování bytekódu.

- lokální proměnné nemohou být využity pro operandy různých typů, což zjednodušuje ověřování bytekódu a GC může jednodušeji nalézt ukazatele při uvolňování paměti
- zásobník metody musí být na konci bloku prázdný, čímž odpadá analýza spojování zásobníků a tím se opět zjednodušuje verifikace

Z předchozích omezení vyplývá, že se množství bytekódu po kompilaci zvětší, protože se

s přísnějším ukládáním lokálních proměnných zvětší i zásobník metody a přibudou `load` / `store` instrukce pro přenos mezi zásobníkem a lokálními proměnnými. Ovšem díky agresivnějším optimalizacím při kompilaci a pozměněným instrukcím (viz níže) bylo naměřeno navýšení velikostí bytekódu pouze o 6% při překladu JavaCard 2.0 Core API oproti KVM.

V samotném instrukčním souboru bylo provedeno několik změn

- podmínky byly zkráceny na 2 bajty, kde první určuje typ podmínky a druhý 8 bitový ofset
- zavedly se prefixy rozšiřující parametry instrukcí, které lze využít právě u podmínek
- `load` / `store` instrukce byly nahrazeny beztypovými a uvolněné místo se zaplnilo `load` / `store` instrukcemi s indexem lokální proměnné přímo v instrukčním bajtu
- byly přidány instrukce `clinit` a `invokeinit`, které se starají o tvorbu objektů a jejich inicializaci namísto původních `new` a `invokespecial`. Tato změna má za cíl zjednodušit ověřování bytekódu tak, aby při vytváření objektů nezůstala na zásobníku neinicializovaná instance.

Jak bylo výše naznačeno, uspořádání *class* souborů (*suite*), bylo pozměněno. Byl stanoven maximální limit na počet statických členských proměnných, symbolické reference jsou vyhledány translátorem, tudíž jsou na cílové zařízení uloženy pouze nalezené indexy a všechna metadata se uloží na začátek archivu, tak že při verifikaci jednotlivých tříd jsou již všechna potřebná data k dispozici.

Proces ověřování mohl být díky restrikcím bytekódu redukován do dvou fází, kde se v první kontroluje platnost instrukcí, indexy z přeložených symbolických odkazů, jestli ukazují na správná místa a proběhne typová kontrola. Ve druhé fázi se ověřují řídicí instrukce jako například podmínky, zda neodkazují doprostřed instrukcí nebo mimo danou metodu. Tímto způsobem je možné při verifikaci mít najednou v RAM pouze pár bytů pro každou metodu.

Při návrhu Squawk se počítalo s odlišnými charakteristikami dostupných paměťových úložišť. Lze využít paměti typu RAM, NVM (například flash, EEPROM) a ROM. Platí, že reference uložené v dočasnější paměti mohou odkazovat na objekty v trvalejší, ale ne naopak. Reference z RAM mohou odkazovat kamkoliv, ukazatelé NVM mohou odkazovat pouze do NVM a ROM a refrence z ROM mohou odkazovat pouze do ROM. Toho se využívá při rozdělování *suite* do jednotlivých částí paměti, kdy neměnná metadata a bytekód mohou být uloženy v NVM nebo ROM, ostatní části jako zásobníky a instanční proměnné musí využívat pouze paměť typu RAM.

Zásobník vlákna je uložen po kouscích, ve kterých je na konec umístěn ukazatel na následující část zásobníku. Každý takovýto kus je objektem pro garbage kolekci. Každé vlákno si také uchovává ukazatel na nejmladší kus zásobníku, který zároveň obsahuje

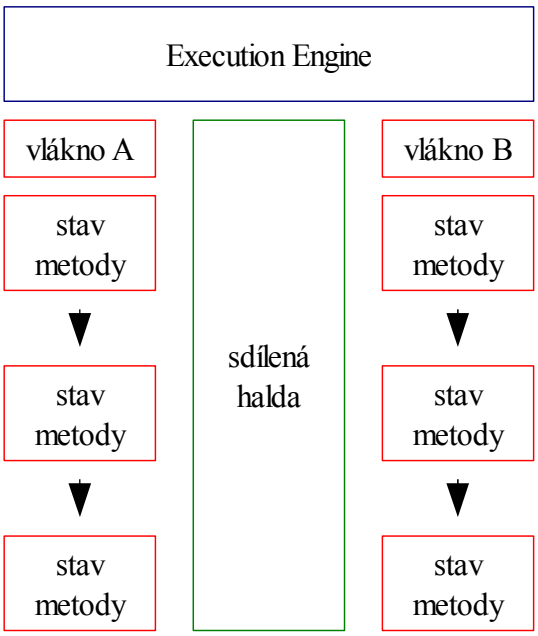
referenci na nejmladší aktivační záznam. Tyto záznamy nepodléhají GC, protože se často mění, a tím by byl systém zbytečně zatěžován. Samotný GC je spuštěn pouze v případě nedostatku paměti pro alokaci nového objektu nebo kusu zásobníku nebo při volání `System.gc()`.

2.1.2 Common Intermediate Language

Common Intermediate Language (CIL) je bytekód využívaný pro Common Language Infractructure (CLI), která je jádrem pro aplikace a celou platformu .NET. Celá architektura je postavena tak, aby CIL bytekód bylo možno využít pro více programovacích jazyků. Každý takovýto jazyk musí odpovídat standardu Common Language Specification [26]. Pro tuto práci a její rozsah je důležitý pouze CIL bytekód a jeho virtuální stroj. Obě dvě součásti jsou popsány v následujících odstavcích.

Spouštěcí model

Virtuální stroj (VES – Virtual Execution System) je zásobníkového typu, podobně jako HotSpot JVM. Jako úložiště pro aplikační kód slouží spouštěcí soubor typu Portable Exectuable (PE), běžně využívaný pro běh nativních programů pod operačním systémem Microsoft Windows.



obr. 2.8: Spouštěcí model VES interpretu

Spouštěcí model virtuálního stroje je klasický, každé vlákno využívá vlastní zásobník a využívá sdílenou haldu. Zásobníky vláken jsou uloženy v paměti jednoho procesu, tedy také sdílené. Po nahrání kódu do VES, probíhá verifikace, po které následuje samotné vykonávání instrukcí. Virtuální mašina pracuje s následujícími datovými typy:

- `int8`, `unsigned int8` – zarovnává se na nativní počet bitů architektury

- `int16`, `unsigned int16` – zarovnává se na nativní počet bitu architektury
- `int32`, `unsigned int32` – klasický 32 bitový typ
- `int64`, `unsigned int64` – klasický 32 bitový typ
- `float32`, `float64` – typy pro číslo s polovoucí desetinou čárkou
- `native int`, `native unsigned int` – velikosti těchto typů závisí na architektuře, kde aplikace běží. Pokud je nutné znát velikosti už při kompilaci, použije se opatrný odhad, tedy 64 bitů.
- `F` – nativní číslo s plovoucí desetinou čárkou, je využíváno pouze samotným VES
- `O` – nativní odkaz na objekt podléhající garbage kolektoru – musí být zajištěno, že se s odkazem nebude pracovat po vykonání GC, protože GC může objekt překopírovat do jiné části paměti
- `&` - ukazatel na paměťový blok, který může nebo nemusí podléhat GC, musí splňovat stejnou podmínku jako `O`

Instrukční soubor

V následujících částech jsou uvedeny příklady kódu zkompilovaného do CIL bytekódu. Operace nad zásobníkem mají podobný charakter jako zásobníkové operace JVM, tedy spotřebovávají operandy, které využívají pro svou činnost.

```
int add(int a, int b)
{
    return a + b;
}

      ldarg.1             // vloží parametr 1 na zásobník
      ldarg.2             // vloží 2. parametr na zásobník
      add                 // sebere a sečte poslední dvě položky na
                          // zásobníku a výsledek vloží zpět na zásobník
      ret                 // návrat z metody
```

kód 2.18: Sčítání dvou čísel v CIL

Následující smyčka cyklu není plně optimalizovaná z důvodu kvality Mono C# kompilátoru, který byl využit pro úryvky bytekódu CIL.

```
int n = 0;
while (n < 100)
      n++;

      ldc.i4.0                  // nahraje hodnotu 0 na zásobník
      stloc.0                   // odebere hodnotu z vrcholku zásobníku do
                                // lokální proměnné p0
      br          label2        // skok na label2

label1:
      ldloc.0                   // vloží lokální proměnnou p0 na zásobník
      ldc.i4.1                  // vloží hodnotu 1 na vrchol zásobníku
      add                       // sečte poslední dvě buňky na zásobníku
      stloc.0                   // výsledek uloží do p0
label2:
      ldloc.0                   // vloží p0 na zásobník
      ldc.i4.s    100           // vloží hodnotu 100 na zásobní
      blt         label1        // porovná poslední dvě hodnoty na zásobníku
                                // a skočí na label1, pokud je druhá hodnota
                                // menší než první
```

kód 2.19: Demonstrace smyčky v CIL

Obdobně jako Dalvik VM, VES umožňuje nahrávat obsah pole známý při kompilaci. Využívá se k tomu tokenu, který odkazuje na metadata, sekci uloženou ve spustitelném souboru obsahující právě například inicializační data pro pole.

```
public int array()
{
      int[] data = {3, 1, 4, 1, 5, 9}
      return data[3];
}

      ldc.i4.6                        // vloží hodnotu 6 na zásobík
      newarr      System.Int32        // vytvoří pole o velikosti hodnoty
                                      // na zásobníku a odebere ji
      dup                             // zdvojí vrchol zásobníku
                                      // (referenci na pole)
      ldtoken     field valuetype 'data-token'
      call        void class System.Runtime.CompilerServices.RuntimeHelpers::
                  InitializeArray(class System.Array,
                  valuetype System.RuntimeFieldHandle)
                                      // zavolá funkci pro zkopírování dat
                                      // odkazovaných tokenem na vrcholu zásobníku
      stloc.0                         // odebere vrchol zásobníku do p0
      ldloc.0                         // nahraje p0 na zásobník
      ldc.i4.3                        // vloží hodnotu 3 na zásobník
      ldelem.i4                       // vloží hodnotu buňky v poli na zásobník
      ret                             // návrat z metody
```

kód 2.20: Přístup k poli v CIL

Pro kompletní vytvoření nového objektu postačí VES, na rozdíl od HotSpot JVM, jediná instrukce, která jednak alokuje paměť pro objekt a jednak ho inicializuje a zavolá jeho konstruktor.

```
obj = new AnObject();
obj.var++;
obj.method();

      newobj       instance void class AnObject::'.ctor'() // vytvoří nový
                                       // objekt a uloží ho na zásobník
      stloc.0                          // vloží vrchol zásobníku do lokální
                                       // proměnné p0
      ldloc.0                          // odebere vrchol zásobníku do p0
      dup                              // dvojí vrchol zásobníku
      ldfld        int32 AnObject::var // vloží členskou proměnnou do zásobníku
      ldc.i4.1                         // vloží hodnotu 1 do zásobníku
      add                              // sečte proměnné na vrcholu zásobniku
      stfld        int32 AnObject::var // výsledek z vrcholu zásobníku odebere
                                       // zpět do členské proměnné objektu
      ldloc.0                          // vloží lokální p0 na zásobník
      callvirt     instance void class AnObject::'method'()
                                       // zavolá metod objektu
```

kód 2.21: Práce s objekty v CIL

VES implementuje systém výjimek, podporující několik typů ošetřovacích rutin.

- *fault* rutina – kód, který se provede při vyhození jakékoliv výjimky v ošetřovaném bloku
- *catch* rutina – spustí se pouze v případě, že typ zachytávané výjimky odpovídá typu vyhozené výjimky.
- *finally* rutina – využívá se pro dokončení operací, jako uvolnění zdrojů a pod., provádí se vždy na konci ošetřovaného bloku kódu, ať je vyhozena výjimka nebo není
- *filter* runtina – pokud výjimka projde tímto blokem kódu, provede se příslušná *catch* rutina, v opačném případě ne a je vyhozena do nadřazeného chráněného bloku, pokud existuje

```
try
{
      this.divZero();
}
catch (Exception e)
{
      this.handleException(e);
}

from        to          type
label1      label2      System.Exception

label1:
      ldarg.0           // vloží aktuální objekt na zásobník
      call instance void class exception::divZero()
                        // volání členské metody, která může vyhodit výjimku
      leave label3      // vyskočí z chráněné ho bloku kódu na label3
label2:
      stloc.0           // odebere vrchol zásobníku do p0
      ldarg.0           // vloží na zásobník aktuální objekt
      ldloc.0           // vloží p0 na zásobník
      call instance void class exception::handleException(class System.Exception)
                        // zavolá metodu ošetřující výjimku
      leave label3      // vyskočí z chráněné ho bloku kódu na label3
label3:
      ret               // návrat z metody
```

kód 2.22: Ošetřování výjimek v CIL

CIL využívá konstrukce *switch* pouze v případě, že testované hodnoty jsou blízko u sebe – u JVM by to odpovídalo instrukci **`tableswitch`**. Pokud ovšem testované hodnoty jsou řídké, VES musí hodnoty porovnávat klasickou sekvencí podmínek, protože CIL bytekód nemá zabudovánu obdobu instrukce **`lookupswitch`**.

```
public int test(n)
{
	switch(n)
	{
		case -100:  ...
		case 0:  ...
		case 100:  ...
		default: ...
	}
}

	ldarg.1			// vloží 1. parametr na zásobník
	stloc.0			// odebere vrchol zásobníku do p0
	ldloc.0			// vloží p0 na vrchol zásobníku
	ldc.i4.s	-100	// vloží hodnotu -100 na zásobník
	beq		switch1	// skočí na switch1 v případě rovností dvou
				// posledních položek na zásobníku
	ldloc.0			// vloží p0 na zásobní
	brfalse		switch2	// pokud je vrchol zásobníku false (hodnota 0)
				// skok na switch2
	ldloc.0			// vloží p0 na zásobník
	ldc.i4.s	100	// vloží hodnotu 100 na zásobník
	beq		switch3	// skok na switch3 při rovnosti 2 posledních
				// položek na zásobníku
	br		label1	// skočí na label1
switch1:
	...
switch2:
	...
switch3
	...
label1:
	br		label2	// skočí na label2

	ldc.i4.3
label2:
	ret			// návrat z metody
```

kód 2.23: Příklad struktury switch s řídkými hodnotami v CIL

Přístup ke sdíleným prostředkům vícevláknové aplikace a jejich synchronizace je možno řešit více možnostmi. První z nich uzamkne objekt při jakémkoliv volání *synchronizované* metody. Tato vlastnost metody se ukládá do oblasti metadat. Je možno explicitně si vyžádat kritickou sekci použitím volání systémové funkce virtuálního stroje `System.Threading.Monitor`, viz příklad. Dále lze před instrukci vložit prefix **`volatile`**, který serializuje čtení nebo zápis do paměti následující instrukce přes všechna vlákna procesu. Konečně jsou zde i možnosti využít atomické operace typu Compare and Swap (CAS) a pod. Synchronizaci v samotném bytekódu je možné provádět pouze za pomocí **`volatile`** prefixu.

```
public void mutateSharedObject(shared obj)
{
       System.Threading.Monitor.Enter(obj);
       obj.mutate();
       System.Threading.Monitor.Exit(obj);
}

       ldarg.1                          // vloží obj na zásobník
       call         void class System.Threading.Monitor::Enter(object)
                                        // uzamkne objekt na vrcholu zásobníku
       ldarg.1                          // vloží obj na zásobník
       callvirt     instance void class shared::mutate()
       ldarg.1                          // vloží obj na zásobník
       call         void class System.Threading.Monitor::Exit(object)
                                        // odemkne objekt na vrcholu zásobníku
       ret                              // návrat z metody
```

kód 2.24: Ošetřování sdíleného stavu v CIL

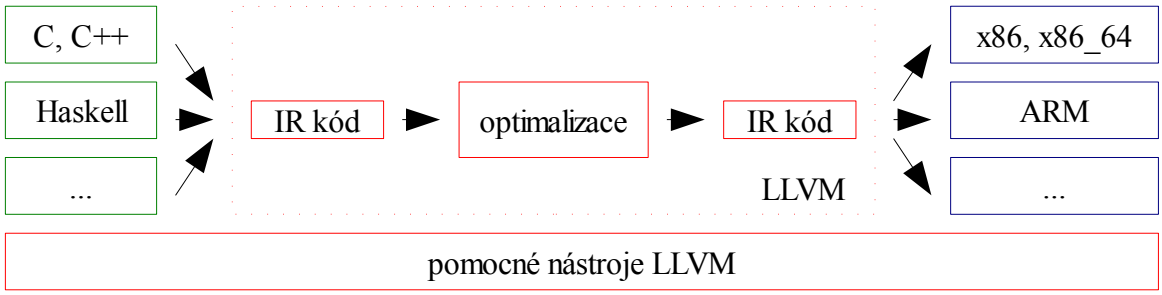
CIL obsahuje několik dalších prefixů, které ovlivňují instrukci bezprostředně následující.

- **`constrained`** – používá se před voláním virtuální metody, zabezpečí dereferenci kontextu třídy *this*, pokud je potřeba.
- **`no`** – při zpracování instrukce se neprovedou kontroly typu, mezí nebo nulového ukazatele
- **`readonly`** – lze použít při čtení buňky z pole. Vynechá se kontrola typu hodnoty z důvodu efektivity. Využívá se při čtení z pole generického typu. Instrukce vrátí *controlled-mutability pointer*, což znamená, že ukazatel na tuto buňku pole nelze využít jako argument ve volání dalších metod.
- **`tail`** – zavolá danou metodu až po odstranění stávajícího zásobníkového rámce
- **`unaligned`** – data následující instrukce nemusí být zarovnána na očekávanou hodnotu, což v určitých případech může uspořit místo bez zvýšení procesorového času potřebného pro zpracování instrukce.
- **`volatile`** – serializuje přístup k proměnné, viz výše

Pro zvýšení efektivity zapouzdřování jednoduchých typů jako **`int`**`xx` a **`float`**`xx`, jsou implementovány instrukce **`box`**, **`unbox`** a **`unbox.any`**.

2.1.3 Low Level Virtual Machine

Low Level Virtual Machine (LLVM) [27] je zastřešující název pro sadu programů usnadňující kompilaci, popřípadě interpretaci různých programovacích jazyků na různé hardwarové architektury. Princip činnosti je načrtnut na obr. 2.9.

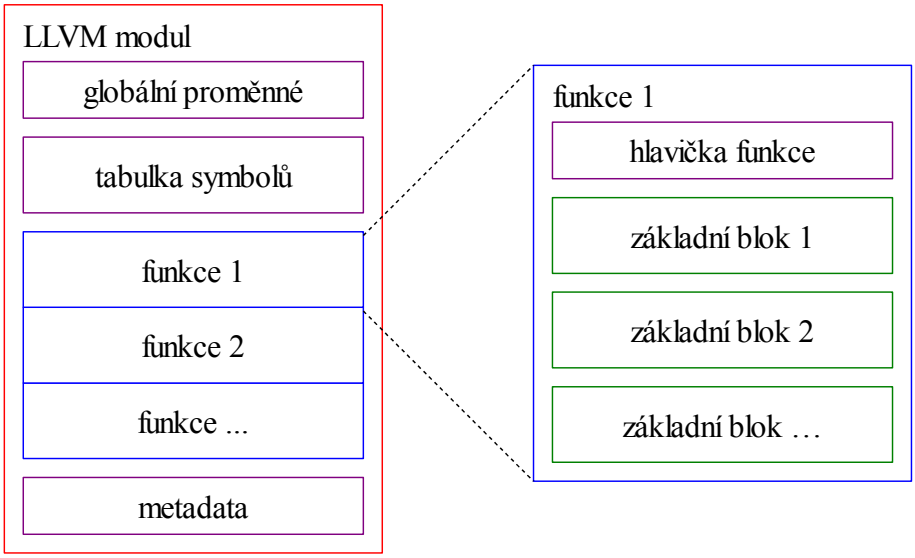


obr. 2.9: LLVM architektura

Kompilátor jazyka rozparsuje zdrojový kód, vytvoří AST stromy a ty následně převede do Intermediate Representation (IR). IR je bytekód, nad kterým se provádí všechny optimalizační fáze. Výsledkem procesu je opět IR, který je v posledním kroku převeden instrukcí požadované hardwarové platformy. Výhodou implementace takovéhoto systému je oddělení kompilace a generování kódu a nezávislost obou částí. Nutností je ovšem dostatečně obecná IR, kterou rozebereme v následující části. Ostatním funkcím LLVM se věnovat nebudeme, neboť to přesahuje rámec této práce.

Spouštěcí model

LLVM program se skládá z modulů obsahujících definice funkcí, globálních proměnných, tabulky symbolů a metadat.



obr. 2.10: LLVM modul

Definice funkce obsahuje informace o volací konvenci, parametrech, návratovém typu a dalších volitelných položkách. Tělo funkce se sestává se základních bloků - tedy bloků, které obsahují jen jednu instrukci, která ruší lineární provádění kódu (vyjma volání jednochuchých funkcí), jako skoky větvení a pod., a tato instrukce musí být na konci základního bloku. Základní bloky formují tzv. Control Flow Graph (CFG) funkce.

LLVM modul může obsahovat sekci metadat, která poskytuje extra informace pro procesy optimalizace, sestavování, generování nebo ladění kódu.

LLVM nedefinuje spouštěcí model pro souběžné vykonávání kódu, namísto toho nechává volnost pro implementace různých typů modelů pro dané platformy za podpory instrukcí pro atomickou práci s pamětí (`cmpxchg`, `atomic_load`, `atomic_store`, ...).

Typový systém

Všechny instrukce pracující s daty podléhají typovému systému LLVM. Byl zaveden kvůli snazší analýze kódu a větším možnostem optimalizace. Datové typy můžeme rozdělit do následujících kategorií

- celočíselné typy – `ixx`, kde `xx` je počet bitů – například `i1`, `i20`, `i32`, `i64`, ...
- typy s plovoucí desetinou čárkou – 32, 64, 80, 128 bitové
- pointer – udává adresu paměti proměnné, ukazatel na návěští se zapisuje jako `i8*`
- `label` – reprezentuje návěští kódu
- `metadata` – obsahuje metadata v modulu
- `void` – typ, který nereprezentuje žádnou hodnotu a nemá žádnou velikost
- function – typ funkce
- `opaque` – reprezentuje struktury, které nemají definované tělo
- vector – odvozený datový typ, používá se pro instrukce SIMD (Signle Instruction Multiple Data)
- structure – odvozený datový typ, kolekce dalších datových typů
- array – odvozený datový typ, pole datových typů

Každá instrukce pracuje jak na vstupu, tak na výstupu pouze s určeným datovým typem.

IR bytekód

Instrukce bytekódu lze zařadit do několika kategorií

- ukončující instrukce – zakončují základní bloky funkce – mezi ně patří návraty z funkce, skoky, větvení a volání dalších funkcí, které se nemusí vrátit na místo původního volání (`ret`, `br`, `switch`, `invoke`, ...)
- binární instrukce – zpracovávají klasické matematické operace, podporují skalární i vektorové vstupy (`add`, `sub`, `div`, ...)
- bitové instrukce – bitové operace, vždy vrací hodnoty stejného typu jako vstupy (`and`, `or`, `xor`, ...)

- paměťové instrukce – instrukce pro čtení a zápis z / do paměti nebo zásobníkových rámců (`alloca`, `load`, `store`)
- vektorové instrukce – pro práci s vektory (`extractelement`, `insertelement`, `shufflevector`)
- typové instrukce – přetypovávají proměnné (`bitcast`, `trunc`, …)
- instrukce pro práci s odvozenými typy - `insertvalue`, `extractvalue`
- ostatní instrukce – porovnávání, volání jednoduché funkce nebo instrukce pro zpracování výjimek (`icmp`, `call`, `landingpad`, ...)

Následující ukázky byly sestaveny pro vytvoření ucelené představy LLVM bytekódu.

```
int add(int a, int b)
{
      return a + b;
}

define i32 @_Z3addii(i32 %a, i32 %b) nounwind {
entry:
  %1 = add nsw i32 %a, %b        // součet argumentů a, b se uloží
                                 // do lokální proměnné
  ret i32 %1                     // návrat z funkce
}
```

kód 2.25: Součet čísel v LLVM IR

Instrukce `phi` vybírá hodntou proměnné podle toho, z jakého uzlu CFG byla přivedena kontrola do aktuálního základního bloku.

```
while(n < 100)
      n++;

entry:
 br label %while.body                                  // skok do těla cyklu

while.body:
  %n.03 = phi i32 [ 1, %entry ],                       // přiřazení v SSA formě
             [ %add, %while.body ]                     // podle předchozí control flow
  %add = add nsw i32 %n.03, 1                          // přičte 1 a vloží do add
  %cmp = icmp slt i32 %add, 100                        // porovnání je menší než 100
  br i1 %cmp, label %while.body, label %while.end      // skok na začátek cyklu,
                                                       // pokud je registr cmp 1,
                                                       // jinak skočí na konec cyklu
while.end:
  ret i32 %add                                         // návrat z funkce
```

kód 2.26: Smyčka v LLVM IR

Pro přístup k buňkám pole se nejprve musí spočítat adresa buňky pomocí instrukce

`getelementptr` a až poté se může načíst samotná hodnota.

```
int array()
{
      int[] data = {3, 1, 4, 1, 5, 9}
      return data[3];
}

entry:
  %arrayidx = getelementptr inbounds [6 x i32]* @_Z5arrayvE4data,
                  i64 0, i64 3                    // vrátí ukazatel buňky
  %0 = load i32* %arrayidx, align 4, !tbaa !0     // načte buňku z pole
  return %0                                       // návrat z funkce
```

kód 2.27: Práce s poli v LLVM IR

LLVM samozřejmě podporuje zachytávání výjimek. Nejčastěji se používají dva mechanismy ošetřování a vyhazování výjimek. První, jednodušší, využívá funkce standardní knihovny C `setjmp` a `longjmp` (SJLJ).

```
push(zasobnikVyjimek, kontext)

vyjimka = setjmp(kontext)
if vyjimka == 0                  // try blok
      ...
      longjmp(kontext, vyjimka)  // throw
      ...
elseif vyjimka == 1              // catch blok
      ...
elseif vyjimka == 2              // catch blok
      ...

pop(zasobnikVyjimek)
```

kód 2.28: Pseudokód SJLJ zachytávání výjimek

Funkce `setjmp` zkopíruje aktuální kontext procesoru včetně instrukčního ukazatele do paměti a vrací 0. Po zavolání `longjmp` s ukazatelem na kontext se stav procesou obnoví a provádění kódu pokračuje podobně jako po prvním volání `setjmp`, jen s tím, že návratová hodnota je jiná. Tento princip využívá systém SJLJ výjimek. Před vstupem do **`try`** bloku se na zásobník uloží informace o předchozím hlídaném bloku kódu a `setjmp` uloží stav procesoru. Při vyhození výjimky se zavolá `longjmp` s posledním uloženým stavem procesoru a díky jiné návratové hodnotě ze `setjmp` se provede místo původního **`try`** bloku **`catch`** blok. Nevýhodou tohoto principu je ukládání stavu procesoru. Na registrových procesorech je tato operace pomalá, díky nutnosti uložení všech registrů do paměti.

Druhý způsob se nazývá Intel Itanium ABI Exception Handling a počítá s větší spoluprací kompilátoru. Při kompilaci se v datové oblasti vytvoří tabulky, které popisují rozmezí ošetřovaných, ošetřujících a čistících bloků (**`try`**, **`catch`**, **`finally`**). Celý proces je

podrobně popsán v [28] a [29]. Zjednodušeně lze říct, že po vyhození výjimky se v první fázi v tabulkách vytvořených kompilátorem vyhledá patřičná ošetřující rutina tzv. *landingpad*, v podstatě simulující odvíjení zásobníku. Pokud se *landingpad* najde, spustí se druhá fáze, kdy se opět od začátku odvíjí zásobník, ale tentokrát se při procesu volají čistící bloky až do doby než se opět narazí na *landingpad* nalezený v první fázi. Rozdělení do dvou fází je výhodné z toho důvodu, že lze výjimečnou situaci ošetřit před spuštěním druhé fáze a pokračovat ve vykonávání kódu z místa, kde byla výjimka vyvolána. Další výhodou tohoto způsobu ošetřování výjimečných stavů je rychlejší provádění normálního kódu. Před každým `try` blokem není nutné ukládat stav procesoru. Na druhou stranu samotné ošetřování výjimek je pomalejší díky postupnému odvíjení zásobníku, ale obecně to problém není. Vychází to z podstaty výjimek, výjimečné stavy nastávají relativně méně často než běžný kód.

Protože byl tento systém navrhován s ohledem na platformní nezávislost, jedním z předpokladů konkrétní implementace je vytvoření vlastní *personality function*, která se volá v každém kroku odvíjení zásobníku a stará se o rozpoznávání typů výjimek, a rozhodování, kdy je třeba zavolat *landingpad*, provést čistící kód, nebo odvíjení přerušit.

LLVM podporuje oba typy zachytávání výjimek. Následující příklad ukazuje Intel Itanium ABI Exception Handling.

```
int exception(int a, int b)
{
      try { divZero(a, b); }
      catch(int e) { handleException(e); }
}

entry:
  %call = invoke i32 @_Z7divZerov(i32 %a, i32 %b)    // volá metodu potencionálně
          to label %try.cont unwind label %lpad      // vyhazující výjimku
lpad:
  %0 = landingpad { i8*, i32 }                 // deklarace landingpadu
             personality i8* bitcast (i32 (...)* @__gxx_personality_v0 to i8*)
             catch i8* bitcast (i8** @_ZTIi to i8*)  // zachytává výjimky typu int
  %1 = extractvalue { i8*, i32 } %0, 1         // extrahuje typ výjimky
  %2 = tail call i32 @llvm.eh.typeid.for(i8* bitcast (i8** @_ZTIi to i8*)) nounwind
                                               // vrací informace o typu výjimky
  %matches = icmp eq i32 %1, %2                // kontrola typu výjimky
  br i1 %matches, label %catch, label %eh.resume     // pokud typ souhlasí, skok
                                               // na ošetřující funkci, jinak se
                                               // pokračuje v odvíjení zásobníku
catch:
  %3 = extractvalue { i8*, i32 } %0, 0         // extrahuje samotnou výjimku
  %4 = tail call i8* @__cxa_begin_catch(i8* %3) nounwind // vrací výjimku jako C++
                                               // objekt
  %5 = bitcast i8* %4 to i32*                  // konvertuje typy
  %exn.scalar = load i32* %5, align 4          // načte hodnotu z adresy ukazatele
  %call3 = tail call i32 @_Z15handleExceptioni(i32 %exn.scalar)
                                               // rutina pro ošetření výjimky
  tail call void @__cxa_end_catch() nounwind   // uvonlní paměť po poslední
                                               // zachycené výjimce
  br label %try.cont                           // skok na konec funkce

try.cont:
  %c.0 = phi i32 [ %call3, %catch ], [ %call, %entry ]     // návratová hodnota je
                                               // závislá na předchozím control flow
  ret i32 %c.0                                 // návrat z funkce

eh.resume:
  resume { i8*, i32 } %0                       // pokračuje v propagaci výjimky
```

kód 2.29: Zachytávání výjimek v LLVM IR

Pro úplnost je níže uveden příklad větvení kódu pomocí instrukce **switch**.

```
int test(int n)
{
       case 0: return 5;
       case 1: return 8;
       case 2: return 1;
       default: break;

       return 3;
}

entry:
  switch i32 %n, label %sw.epilog [     // instrukce switch
    i32 0, label %return
    i32 1, label %sw.bb1
    i32 2, label %sw.bb2
  ]

sw.bb1:
  br label %return                      // skok na návěští
sw.bb2:
  br label %return                      // skok na návěští
sw.epilog:
  br label %return                      // skok na návěští
return:
  %retval.0 = phi i32 [ 3, %sw.epilog ],      // SSA přiřazení, uloží do registru
                      [ 1, %sw.bb2 ],         // retval hodnotu, která záleží
                      [ 8, %sw.bb1 ],         // na předchozím control flow
                      [ 5, %entry ]
```

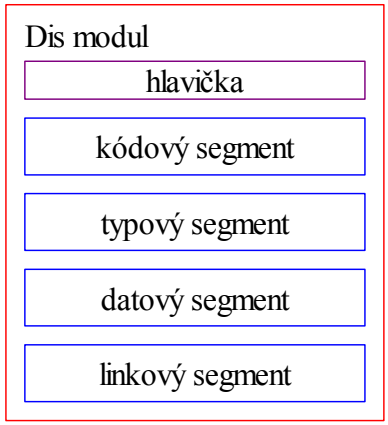
kód 2.30: Konstrukce switch v LLVM IR

2.1.4 Dis Virtual Machine

Dis Virtual Machine [31] je registrový interpret vyvinutý pro operační systém Inferno od firmy Vita Nuova a hlavní jazyk, který se kompiluje do bytekódu Dis VM se nazývá Limbo, ačkoliv existují i implementace překladačů z jiných jazyků. Interpret obsahuje JIT překladač, garbage kolektor a podporuje dynamické nahrávání modulů. Z pohledu této práce je zajímavá koncepce komunikačních kanálů a komunikace mezi procesy, které budou popsány později.

Spouštěcí model

Aplikace pro interpretaci na virtuálním stroji Dis je rozdělena do modulů [32], obdobně jako program v Javě do *class* souborů.



obr. 2.11: Dis VM modul

Moduly obsahují kódový, typový, datový a linkový segment, popřípadě ještě pak sekce s tabulkou symbolů pro ladění. Do datového segmentu se ukládají informace o typech v paměťové oblasti a v zásobníkových rámcích. Bitmapa v typové sekci obsahuje informace o paměti, se kterou bude virtuální stroj pracovat, zda se jedná o ukazatele nebo ne. Tuto informaci využívá garbage kolektor. V linkovém segmentu je úložiště exportovaných funkcí modulu.

Paměťová oblast rezervovaná pro vlákno se skládá z globální datové oblasti, zásobníkové oblasti a registrů

- PC – čítač instrukcí
- MP – memory pointer – ukazatel na oblast obsahující ukazatele
- FP – frame pointer – ukazatel na zásobníkové rámce. Pokud by zásobník přetekl, automaticky se rozšíří.

Před každým voláním funkce se musí explicitně vytvořit její zásobníkový rámec instrukcí `frame` a až poté samotnou funkci zavolat instrukcí `call`. Pokud je potřeba volat exportované funkce z jiných modulů, existují pro tento účel speciální instrukce `mframe` a `mcall`.

Virtuální stroj umí pracovat s klasickými celočíselnými typ velikosti 8, 32, 64 bitů, s čísly s plovoucí desetinou čárkou, ukazateli, řetězci v UTF-8. Navíc u každé datové buňky rozlišuje, zda se jedná o hodnoty nebo o ukazatele.

O alokaci paměti a její uvolňování se stará garbage kolektor, který je hybridního typu. Počítá reference a problém s cykly v grafu řeší real-time mark-and-sweap kolektorem. Kvůli jeho efektivitě musí být každá manipulace s ukazateli známa. Pro tento účel se využívá typový segment obsahující bitovou mapu paměti.

Kanály

Koncept komunikačních kanálů vychází z [32]. V prostředí vláken spuštěných souběžně ho lze použít jako alternativu ke klasické sdílené paměti, semaforům, kritickým sekcím nebo

monitorům. V jednovláknových procesech mohou kanály nahradit *event driven* programování, popřípadě využít pro tvorbu generátorů a korutin, což jsou formy kooperativního multitaskingu.

Kanálem lze poslat data z jednoho vlákna / procesu do druhého, popřípadě do stejného, jen na jiné místo v kódu. Komunikace probíhá obousměrně a synchronně, dokud všechny data nebudou druhou stranou přijata, instrukce odesílání zablokuje provádění kódu aktuálního vlákna, stejně tak jako instrukce pro naslouchání se zablokuje, dokud nebudou přijata jakákoliv data. V případě, že vše probíhá v jednom vláknu, plánovač po zablokování při odeslání pokračuje ve spouštění kódu v místě, kde je naslouchánо na daném kanálu. Využívá se klasického round-robin plánování. Asynchronní komunikace lze simulovat vytvořením samostatného vlákna pro odesílání dat kanálem.

Další výhodou koncepce kanálů je snadné maskování komunikačního protokolu a komunikačních rozhraní tam, kde není třeba, aby aplikace znala veškeré detaily procesu. Komunikace může probíhat mezi vlákny, procesy nebo po síti mezi počítači a vše může být transparentní za předpokladu, že jsou daná komunikační rozhraní implementována buď ve virtuálním stroji popřípadě přímo na hardwaru.

Instrukční soubor

Každá instrukce může obsahovat až tři operandy. První dva jsou zdrojové a do posledního operandu se obvykle ukládá výsledek operace. Všechny tři operandy mohou obsahovat efektivní adresu tj. v kontextu Dis VM přímá adresa, nepřímá s ofsetem nebo v případě prvního a třetího operandu dvojitě nepřímá se dvěma nepřímými ofsety.

Instrukce by se daly rozdělit do níže uvedených kategorií. Písmeno *X* znační, že se u instrukce rozlišuje typ operandů.

- matematické instrukce – **add**X, **sub**X, **mul**X, ...
- bitové instrukce – **and**X, **or**X, **xor**X, ...
- řídicí instrukce – **jmp**, **call**, **ret**, tabulka skoků **goto**, vytvoření vlákna **spawn**, ukončení vlákna **exit**, větvení, testování čísel v seznamu rozsahů **case**X, testování řetězců **casec**, …
- instrukce pro práci s objekty – vytvoření objektu / s vynulováním **new** / **newz**, vytvoření pole / s vynulováním **newa** / **newz**
- paměťové instrukce – vytvoření zásobníkového rámce **frame**, počítání efektivní adresy položky v poli **indx**, načtení prvku z pole **ind**X, kopírování proměnné / ukazatele / paměťového bloku **mov**X / **movm** / **movmp**
- řetězcové instrukce – spojování řetězců **addc**, počítání délky **lenc**, extrahování podřetězců **slicec**
- instrukce pro práci se seznamem – přidání do seznamu **cons**X, kopírování první

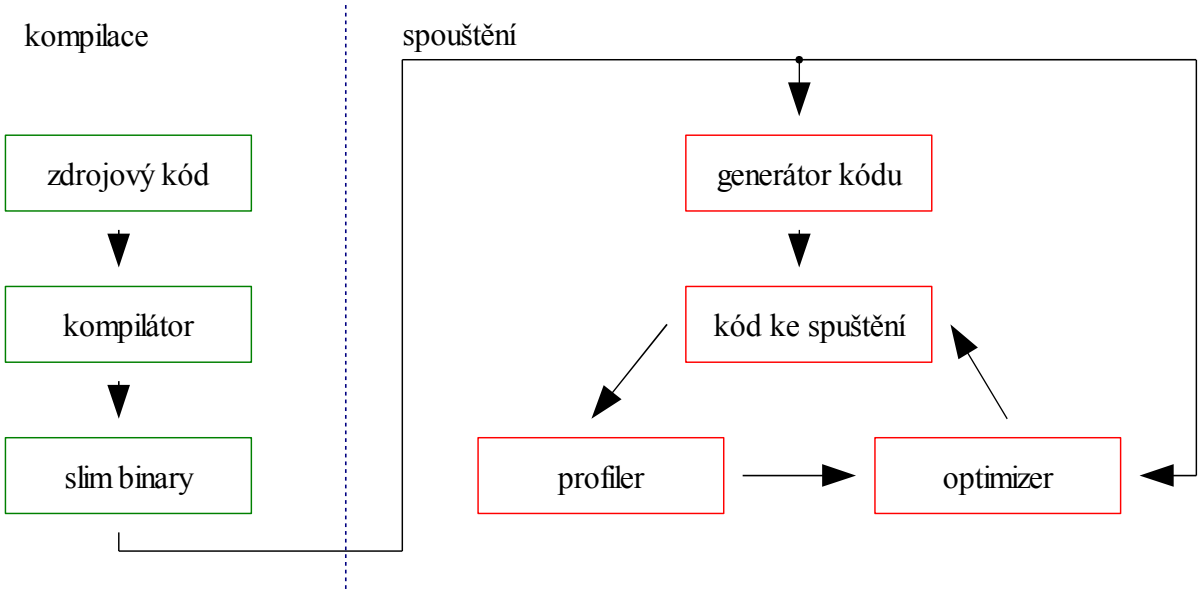
položky `headx`, poslední položky `tail` a zjištění délky seznamu `lenl`

- instrukce pro práci s kanály – vytvoření kanálu `newc`, synchronní čtení / zápis do kanálu `recv` / `send`, blokující / neblokující výběr kanálů ze seznamu, které jsou připravené k čtení popř. zápisu `alt` / `nbalt`,
- instrukce pro manipulaci s moduly – načtení modulu do paměti VM `load`, alokace zásobníkového rámce v jiném modulu `mframe`, volání funkce z jiného modulu `mcall`, alokace objektu definovaného v jiném modulu `mnewz`, vytvoření vlákna v jiném modulu `mspawn`
- instrukce pro konverzi základních typů

2.1.5 Juice

Virtuální stroj Juice byl vyvinut Michaelem Franzem a Thomasem Kistlerem jako alternativa ke klasickým VM pro interpretaci Java bytekódu (pozn.: tato sekce nepojednává o Juice JVM [34] pro real-time aplikace, jejímž hlavním autorem je Corrado Santoro, a jejímž hlavním přínosem je hard real-time garbage kolektor).

Hlavní odlišností Juice interpretu od klasických je fakt, že kód není reprezentován bytekódem, ale abstraktními syntaktickými stromy (AST). Kompilátor přeloží zdrojový kód do AST reprezentace, která lépe vystihuje strukturu aplikace. AST reprezentace obsahuje více informací, protože klasický bytekód je v podstatě linearizací AST, kde se z hlediska správné funkčnosti programu ztrácí nedůležitá data. Tato data mohou ovšem posloužit jak k rychlejšímu generování kódu na cílové architektuře, tak k jednoduší verifikační fázi, tak i ke zjednodušení optimalizací.



obr. 2.12: Princip činnosti Juice

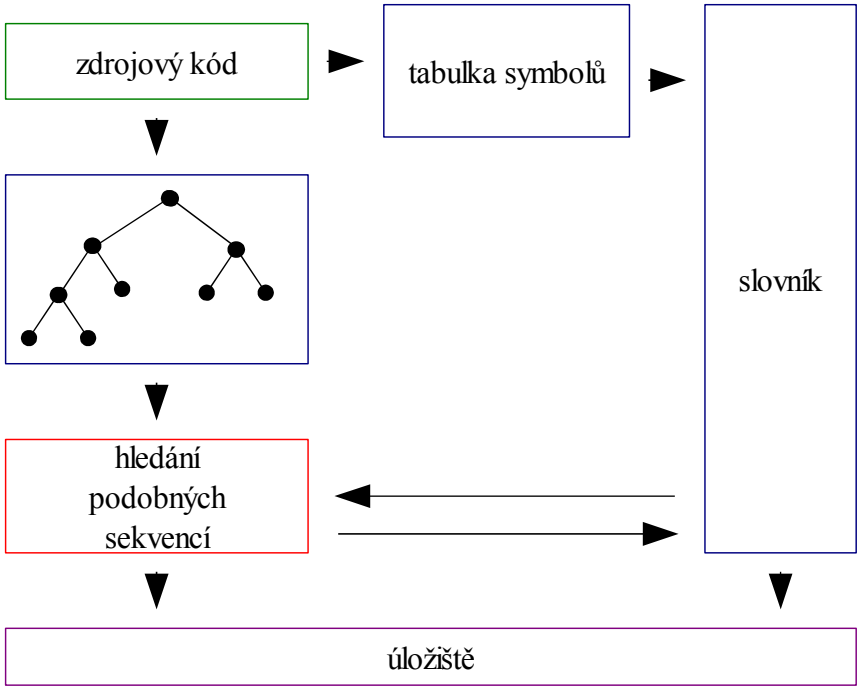
Zkompilovaná AST reprezentace se ukládá do souboru formátu *Slim Binary* [35]. Tento

formát je speciálně navržen k tomu, aby s co největší účinností bylo možno AST reprezentaci přenášet. Využívá se k tomu algoritmus podobný Lempel-Ziv-Welch (LZW) kompresi. Ve zdrojovém kódu se často objevují podobné konstrukce a výrazy,

```
a + b          =>     (a + ..) b
a + c          =>     (a + ..) c
6 symbolů      =>     4 symboly
```

kód 2.31: Princip komprese AST

které jsou kompilátorem přeloženy opět do podobných částí AST, viz obr. 2.9 . Podle [35] je úspora místa oproti zkomprimovanému standardnímu Java bytekódu více než 40% na testovaných datech.



obr. 2.13: Kompilace zdrojového kódu do Slim Binary

Ve fázi distribuce a spouštění aplikace se pracuje se soubory formátu *Slim Binary* [37]. Při startu programu se aktivuje generátor kódu pro danou platformu. AST reprezentace se přeloží do instrukční sady procesoru a následně se spustí. Za běhu je aplikace profilována a údaje o místech, která jsou potřeba optimalizovat nejvíce se postupují VM, kde je provedena analýza a optimalizace stávajícího úseku kódu. Daná část se pak za běhu nahradí optimalizovanou. Výhodou je, že optimalizace jsou efektivnější, protože VM má více informací o kódu díky AST reprezentaci a díky informacím získaných za běhu programu. Nevýhodou celého tohoto procesu může být, že počáteční prodleva při spuštění aplikace je díky překladu do strojových instrukcí velká, autoři ovšem tvrdí, že tato skutečnost je vyvážena nižší prodlevou při načítání v důsledku menší velikosti na úložišti.

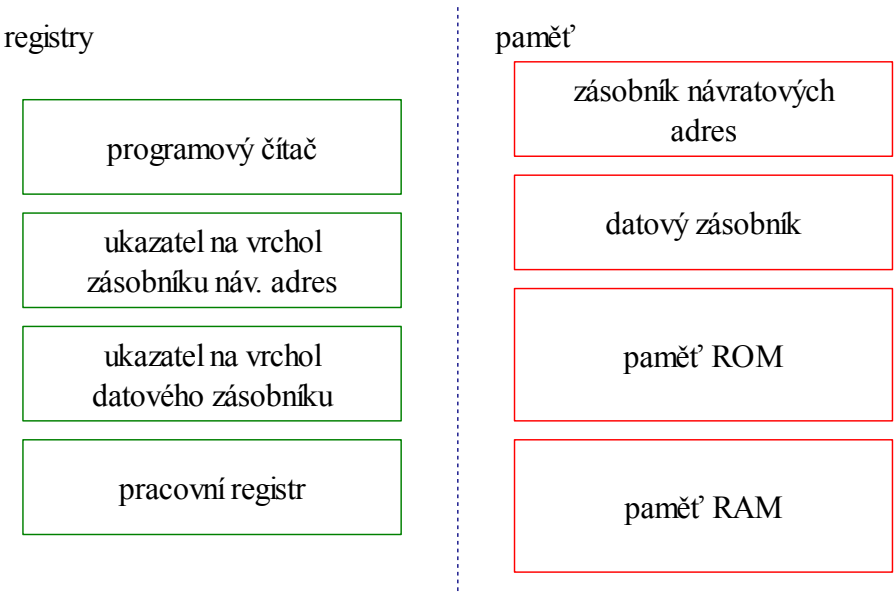
2.1.6 Forth

Virtuálních strojů typu Forth je nepřeberné množství, proto budou dále popsány pouze obecné principy a základní stavební kameny. Tato technologie je stará více jak 30 let, přesto se v různých obměnách používá dodnes. Lze uvést, že nahrazení BIOSu některých desktopových počítačů bylo realizováno technologií OpenFrimware [38], kerá je založena na Forthu, třetí fáze FreeBSD boot loaderu obsahuje interpret Forthu, nebo že americká firma Forth Inc. dodává své systémy pro součásti družic, raketoplánů a jiných aplikací, kde se klade důraz na spolehlivost [39].

V této kapitole budou načrtnuty pouze základní principy, podrobnější popis bude následovat ve třetí části, kde bude navržen a popsán konkrétní virtuální stroj typu Forth.

Spouštěcí model

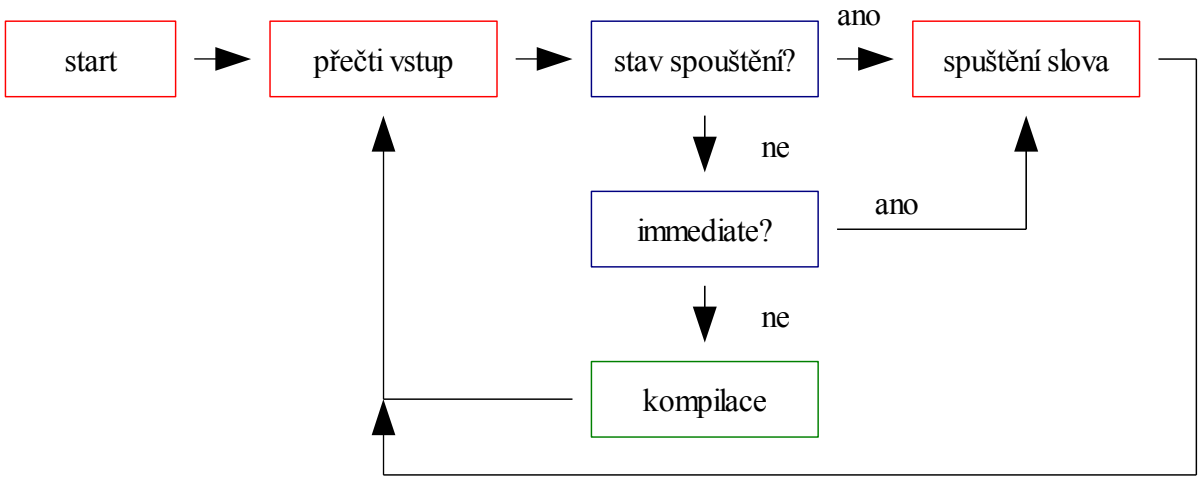
Základem Forth VM jsou dva nebo více zásobníků, které nemusí být adresovatelné, operační paměť RAM a popřípadě i paměť typu ROM. Může se tedy jednat i o Hardwardskou architekturu.



obr. 2.14: Minimální požadavky na paměťové oblasti Forth VM

Jeden zásobník slouží pro návratové adresy z volání funkcí, druhý a případně další pro data. Při volání funkce, které se říká slovo, musí být na datovém zásobníku uloženy všechny jeho parametry, na návratový zásobník se uloží instrukční ukazatel zvýšený o jedničku. Po vykonání patřičného kódu se výstup slova opět uloží na datový zásobník, odebere se poslední položka na zásobníku návratových adres a skočí se na její hodnotu, tedy na místo za voláním funkce. Teoreticky je tak slovo funkcí, která jako vstup přebírá celý zásobník a jako výstup poskytuje opět celý zásobník. Slova jsou uložena ve slovníku, který je typicky reprezentován spojovým seznamem nebo hash tabulkou, a je možné ho rozšiřovat i za běhu programu.

Existují dva základní stavy virtuálního stroje, stav kompilace a stav spouštění. Mezi těmito stavy lze libovolně přepínat. Diagram na obr. 2.15 zjednodušeně popisuje pouze typickou práci interpretu, samotný interpret lze za běhu jakkoliv přeprogramovat. Podrobnější popis vnější smyčky interpretu lze nalézt v sekci implementace na straně 62.



obr. 2.15: Zjednodušený popis běhu vnějšího interpretu typu Forth

Ve stavu kompilace se každé slovo na vstupu zakompiluje do definice předchozího slova. Naopak ve stavu spouštění jsou slova na instrukčním ukazateli prováděna. Aby bylo možné z režimu kompilace přepnout do stavu spouštění, existují tzv. *immediate* slova, která se i přes fakt, že právě probíhá kompilace, spustí. Takto lze vytvářet řídící konstrukce nebo makra za běhu programu, popřípadě kompilovat výsledky složitějších výpočtů přímo do kódu. Tento systém umožňuje obrovskou flexibilitu, jak například pro vytváření doménových jazyků, tak je možno ho využít pro JIT optimalizace.

Bytekód

Existuje několik druhů reprezentace instrukcí Forthu. Každý typ má své výhody a nevýhody. V následujícím odstavci jsou pouze načrtnuty základní typy, opět budou podrobě rozebrány a implementovány ve třetí části

- *indirect threading* – využívá dvojité přesměrování, což snižuje rychlost interpretu díky potřebě provádět dvě čtení z paměti před načtením další instrukce. Je nejlépe portovatelný.
- *direct threading* – využívá pouze jednoduché přesměrování, umožňuje rychlejší provádění kódu, díky pouze jednomu čtení z paměti před načtením instrukce, ale při implementaci interpretu je třeba využít velmi jednoduchý JIT kompilátor, což má za následek pro každou platformu použít patřičný kód.
- *subroutine threading* – kód aplikace je překládán jednoduchým JIT kompilátorem do instrukcí procesoru, takže rychlost spouštění je téměř totožná s nativní aplikací, ale opět se snižuje portovatelnost mezi platformami

a na některých se velikost reprezentace programu v paměti může zvětšit

- *token threading* – bytekód, oproti klasickému musí mít schopnost rozšiřování. Je nejpomalejší ze všech reprezentací i za použití triků z Dalvik VM (viz kód 2.11), ale na běžně využívaných platformách nejúspornější co do velikosti reprezentace.

Test rychlosti jednotlivých druhů interpretů na různých platformách lze nalézt na [42].

Zdrojový kód

Pro zápis zdrojového kódu se využívá postfixová notace, někdy nazývaná reverzní polská notace (RPN). Výhodou oproti prefixové notaci využívané například v Lispu nebo infixové je fakt, že není potřeba závorek. Naopak nevýhoda oproti infixové notaci spočívá ze začátku v tom, že si lze těžko bez znalosti zásobníkových efektů slov představit výpočet složitějšího matematického výrazu. To nutí slova ve Forthu psát krátká, což ale má za následek lepší strukturu kódu a snazší využití opakovaných sekvencí. Slova jsou od sebe oddělena bílými znaky a název slova může obsahovat všechny ostatní znaky.

Níže je uveden jednoduchý výpočet kvadratického diskriminantu. Komentáře se obvykle uvádí v závorkách a v závorkách se uvádějí zásobníkové diagramy. Zásobníkový diagram se čte zleva doprava, jako kdyby zásobník rostl zleva doprava. Před oddělovačem `--` se uvádní počáteční stav zásobníku a za ním koncový po operaci. Při vynechání oddělovače je zaznamenán pouze koncový stav. V podrobném zápisu je u každého kroku znázorněn stav zásobníku po vykonání dané instrukce.

```
: square            dup mul                          ( b -- b*b )
: discriminant      tuck square tuck 4 mul mul sub   ( a b c -- b^2-4ac )

# podrobný zápis

: square            ( b )
      dup           ( b b )
      mul           ( b*b )

: discriminant      ( a b c )
      tuck          ( c a b )
      square        ( c a b^2 )
      tuck          ( b^2 c a )
      4             ( b^2 c a 4 )
      mul           ( b^2 c 4a )
      mul           ( b^2 4ac )
      sub           ( b^2-4ac )
```

kód 2.32: Výpočet kvadratického diskriminantu ve Forthu

Zdrojový kód v 2.32 neodpovídá standardu ANS Forth [43], jeho účelem je pouze ukázka zápisu zdrojového kódu ve Forthu.

3 Implementace

V této části práce bude rozebrán návrh a implementace virtuálního stroje typu Forth. K požadavkům uvedených v první kapitole je potřeba přidat portovatelnost. Protože tento interpret bude spouštěn na platformách, které nejsou optimalizované pro běh zásobníkového virtuálního stroje, a které mají různé spouštěcí modely a charakteristiky, bude nutné zvolit kompromis mezi rychlostí vykonávání instrukcí a přenositelností. Z toho vyplývá výběr typu *threadingu*.

Naopak při návrhu procesorové architektury, která by měla usnadnit práci interpretu a dopomoci k vyšší efektivitě, je možné zcela odstranit požadavek portovatelnosti samotného interpretu a samotných strojových instrukcí, protože nikde jinde nebude daný interpret spouštěn a instrukce vykonávány, než na navržených platformách. Co by mělo zůstat co nejvíce přenositelné mezi všemi architekturami, je zdrojový kód.

3.1 Interpret

Jak již bylo řečeno, nejdříve je nutné vybrat správný spouštěcí model, který odpovídá požadavkům přenositelnosti.

3.1.1 Spouštěcí model

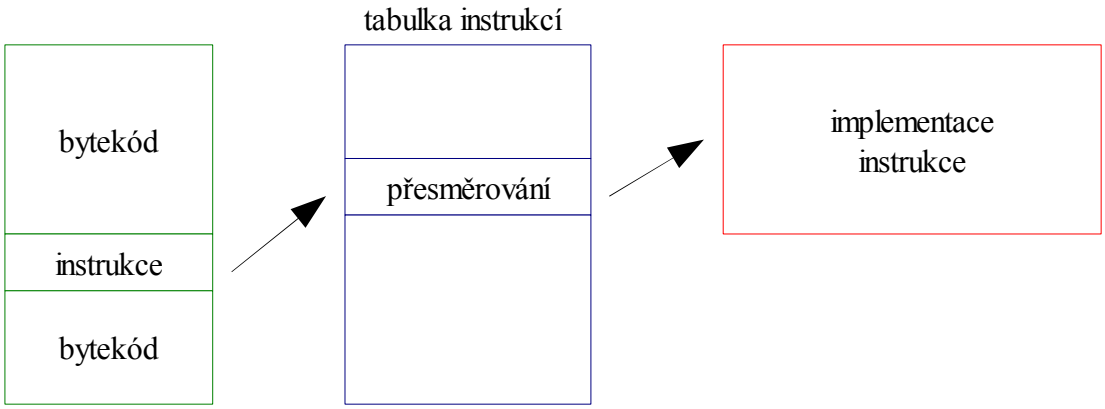
V následujícím textu je nutné rozlišit dva druhy instrukcí. Strojové instrukce, tedy ty vykonávané na samotném procesoru, a instrukce bytekódu, tedy ty, které vykonává interpret, a k jejich spouštění jsou potřeba strojové instrukce. Jedním z kritérií pro porovnávání reprezentací spustitelného kódu je spotřeba strojových instrukcí a přístupů k paměti pro vykonání jedné instrukce bytekódu. Uvažovaná porovnávání nezapočítávají v úvahu JIT kompilaci. Ta bude diskutována dále.

Největším problémem dnešních mikroprocesorů a mikročipů je rychlost a spotřeba energie při přístupu k paměti následovaným spotřebou energie hodinových tiků a spotřebou energie samotných hradel při vykonávání instrukce. Nejprve je tedy nutné zajistit co nejmenší počet přístupů do paměti při čtení a vykonávání instrukce. Na různých architekturách se počty samozřejmě budou lišit, uvažována je architektura CISC (Complex Instruction Set Computer) typu x86.

Token threading

Klasický interpret využívající bytekód je velmi dobře portovatelný a dokonce i samotný bytekód lze většinou spouštět na různých architekturách. Pokud je potřeba rozšířit funkčnost aplikace, standardní postup je vytvoření metody, tj. alokování místa pro ní, a následně zapsání kódu tím samým typem bytekódu, jako při vytváření ostatních metod. Tzn. reprezentace programu je statická, nemění se, a je tedy snadné ji optimalizovat co do velikosti. Většinou si interpret vystačí s jedním bajtem na bytekódovou instrukci. Při spouštění je ovšem nutné

načíst instrukci, dekódovat ji, vyhledat v tabulce instrukcí a následně ji vykonat, což může znamenat potřebu dekódovat další parametry. Aplikace psané ve Forthu typicky volají metody mnohem častěji než běžné programy v C, takže by to v tomto případě mělo za následek časté čtení parametru instrukce – většinou adresy skoku nebo volání metody.



obr. 3.1: Token threading

Načíst instrukci bytekódu z paměti je potřeba vždy. Vyhledání v tabulce instrukcí stojí opět dva paměťové přístupy a skok na metodu vykonání instrukce další paměťový přístup. Je možné vynechat hledání v tabulce instrukcí a spočítat adresu rutiny pro vykonání instrukce, to ale stojí další cykly načítání a vykonávání strojových instrukcí. Většinou je nutné provést operaci sčítání spolu s bitovým posunem. Ilustraci tohoto principu lze nalézt v kódu 2.11. Pokud čítač instrukcí je umístěn v procesorovém registru, tak posun čítače spotřebuje pouze jedno čtení z paměti, tedy načtení strojové instrukce inkrementace. Celkem sečteno minimálně 5 – 6 čtení z paměti slouží k posunu čítače a spuštění instrukce bytekódu. Je nutné brát v potaz i hierarchii cache pamětí, ale pak je nutné zkoumat konkrétní architekturu. Bez použití inline optimalizací v typické programu Forth interpret stráví mnoho času čtením parametrů bytekódu, z důvodu častých skoků. Což znamená další paměťové přístupy. Počet strojových instrukcí se liší podle zvolené metody, při využití vyhledávací tabulky minimálně 4 (načtení instrukce bytekódu, vypočtení ofsetu v tabulce a načtení z tabulky další, inkrementace programového čítače, skok na implementaci instrukce).

Subroutine threading

Naprosto odlišným způsobem se chová interpret postavený na *subroutine threading* modelu. Reprezentace kódu v paměti není postavena na bytekódu, ale na sekvenci strojových instrukcí, většinou skoků nebo volání podfunkcí. Jako ilustrace takové reprezentace poslouží úryvek 3.1.

```
// naivní reprezentace
methodA:
       call methodB
       call instructionA
       call instructionB
       call methodC
       return

// optimalizovaná reprezentace
optimizedA:
       call methodB
       call insructionA
       call instructionB
       jump methodC        // tail call
```

kód 3.1: Reprezentace subroutine threading modelu

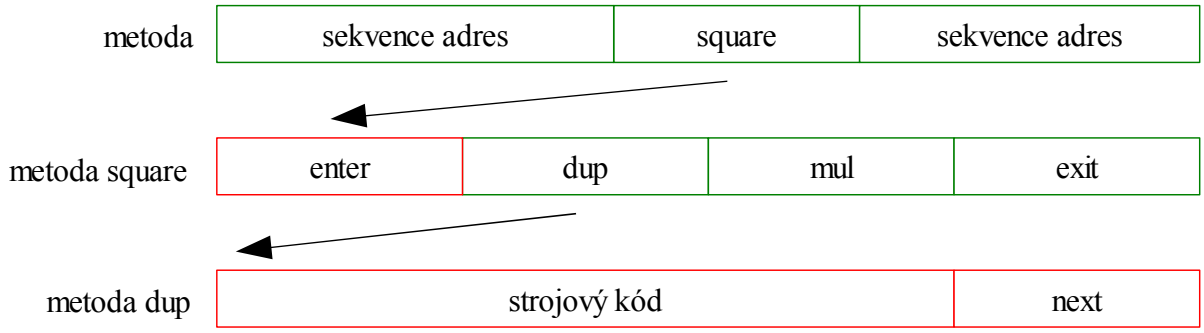
Výhoda oproti bytekódu spočívá v tom, že není nutné si explicitně udržovat čítač instrukcí, protože je využit nativní čítač na procesoru. A ani ukazatel na vrchol zásobníku návratových adres není třeba explicitně zavádět, protože se použije nativní registr používaný při volání podprocedur. V závislosti na architektuře se ovšem může velikost takovéto reprezentace zvýšit. Na architekturách typu x86 a x86_x64 zabere velikost jednoho záznamu 2 - 5 bajtů [02] při využití instrukcí `call rel32 / rel16 / rel8`. nebo na architektuře ARM 4 bajty při využití instrukce `bl rel24`. Nevýhodou takovéto reprezentace je praktická nepřenositelnost samotného bytekódu mezi platformami, náročnost implementace přenositelného interpretu se zvyšuje díky potřebě udržovat větší množství platformně závislého kódu. Dalším už méně důležitým faktem je samotná nutnost JIT kompilátoru, složitost jádra interpretu se tak zvyšuje.

K načtení a začátku vykonávání instrukce interpretu postačí dvě strojové instrukce, samotný skok na rutinu a návrat z ní. Pro přechod mezi instrukcemi je potřeba načíst tři paměťové buňky a do jedné zapsat. Je nutné načíst strojovou instrukci volání, uložit návratovou adresu na zásobník, načíst strojovou instrukci návratu a uloženou hodnotu na zásobníku. Na většině architektur je *subroutine threading* nejrychlejší formou interpretace. Pro klasické interprety je naopak obrovskou nevýhodou jak nepřenositelnost reprezentace aplikace, tak obtížná přenositelnost samotného interpretu. Pro jazyky typu Forth první problém odpadá, protože se programy většinou kompilují přímo ze zdrojového kódu. Otázkou zůstává velikost reprezentace, která je závislá na architektuře.

Direct threading

Další využívanou metodou interpretace je *direct threading*. Metoda programu je reprezentována seznamem adres, na kterých jsou implementovány ostatní metody, které se mají při průchodu volat. Protože se nejedná o strojový kód, je nutné si udržovat pracovní registr, načítat z něho adresy a následně na ně přenášet kontrolu. Aby takovýto systém mohl fungovat, na začátku metody musí být krátký úsek strojového kódu, který zajistí, že se

z následující sekvence budou adresy načítat a budou se ostatní metody volat. Na obr. 3.2 jsou adresy metod označeny zeleně a strojový kód červeně:



obr. 3.2: Direct threading

K přesunu instrukčního ukazatele jsou nutné dva registry. Jeden jako programový čítač a druhý, již zmiňovaný, pracovní registr. Programový čítač ukazuje vždy na další vykonávanou instrukci a do pracovního registru se z původní hodnoty na adrese čítače načte adresa kódu pro vykonání. Výše popsaný proces přesunu kontroly je naznačen v následujícím úryvku:

```
pcw = (pc)          // do pracovního registru se přesune hodnota na adrese
                    // programového čítače (adresa začátku volané funkce)
pc = pc + 1         // čítač se přesune na další instrukci
jump pcw            // skok na hodnotu v pracovním registru
```

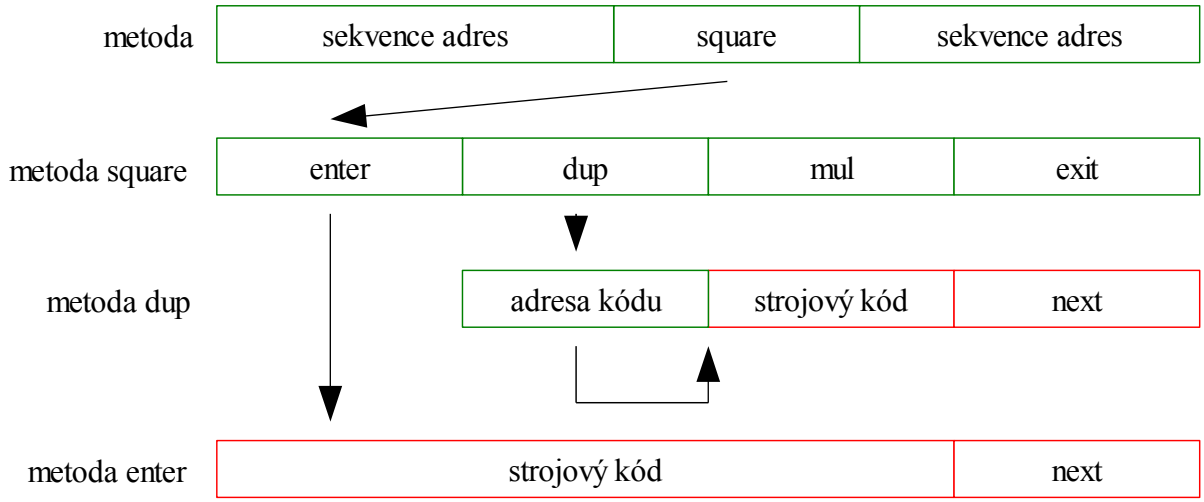
kód 3.2: Funkce `next` pro direct threading

Funkci na 3.2 se také říká vnitřní interpret, vnější interpret bude popsán dále. Tuto funkci je nutné pro každý přesun na další interpretovanou funkci. Vnitřním interpretem jsou ukončeny metody `enter`, `exit` a všechny nativně implementované instrukce. Funkce interpretované aplikace musí být ukončeny adresou na začátek metody `exit`. Význam metod `enter` a `exit` spočívá v uložení návratové adresy, neboli programového čítače, na zásobník v prvním případě a načtení návratové adresy do programového čítače při volání `exit`.

V celkovém součtu se při přesunu vykonávání kódu na další instrukci interpretu spotřebují tři strojové instrukce ve funkci `next` v úryvku 3.2 a počet paměťových přístupů se zastaví na hodnotě čtyři (jedno čtení z adresy programového čítače a tři načtení strojových instrukcí). Výhodou tohoto spouštěcího modelu je jeho relativní rychlost, naopak interpret musí implementovat jednoduchou JIT kompilaci (volání funkce `enter`), což opět znesnadňuje přenositelnost na další platformy.

Indirect threading

Spouštěcí model *indirect threadingu* vychází z modelu *direct threadingu*. Rozdílem oproti předchozímu je jedno přesměrování navíc, jedna instrukce funkce `next` navíc a pozměněná úvodní část každé metody. Mechanismus přenosu kontroly zachycuje obr. 3.3.



obr. 3.3: Indirect threading

Nyní každá metoda nezačíná strojovým kódem, ale ukazatelem na strojový kód, opět většinou na metodu `enter`. Jak již bylo řečeno, je nutné provést jedno přesměrování navíc.

```
pcw = (pc)          // do pracovního registru se přesune hodnota na adrese
                    // programového čítače (adresa začátku volané metody)
pc = pc + 1         // čítač se přesune na další instrukci
code = (pcw)        // na začátku volané metody nyní není strojový kód, ale
                    // ukazatel, takže je nutné načíst adresu strojového kódu
jump code           // skok na hodnotu v pracovním registru
```

kód 3.3: Funkce `next` pro indirect threading

To má za následek o jednu instrukci navíc při spouštění metody a o dva paměťové přístupy navíc (načtení nové instrukce a přesměrování na strojový kód). Celkem tedy přesun kontroly spotřebuje čtyři instrukce a 6 paměťových přístupů. Spolu s metodou *token threadingu* patří k nejpomalejším. Výhodou oproti ostatním metodám je snadná přenositelnost výsledného interpretu na další platformy. Pokud interpret není implementován v instrukcích assembleru, o všechen platformě závislý kód se postará překladač programovacího jazyka, který je využit pro kompilaci interpretu. Další plusem tohoto modelu je jednoduchost samotného interpretu. Není třeba JIT kompilátoru pro nově definované metody.

3.1.2 Reprezentace kódu

V předchozí podkapitole byly rozebrány nejčastěji používané metody implementací interpretu, spočítány strojové instrukce a paměťové přístupy potřebné k vykonávání jeho instrukcí. Tabulka 3.1 tyto poznatky shrnuje.

threading	strojové instrukce	přístupy do paměti
token	4	5 až 6
subroutine	2	3
direct	3	4
indirect	4	6

tab. 3.1: Porovnání spouštěcích modelů interpretů

Nejrychlejšími metodami pro implementaci interpretů se tedy zdají být *subroutine threading* a direct threading. Obě dvě varianty ovšem nesplňují podmínku snadné přenositelnosti na další platformy. Za předpokladu typické charakteristiky krátkých funkcí ve Forthu je tedy ze zbylých možností lepší varianta *indirect threadingu*, protože není vyžadováno další čtení parametru instrukce interpretu pro volání funkce aplikace. Navíc pokud by byla požadována větší rychlost, je možné interpret pro danou platformu snáze upravit do formy *direct threadingu*. Jako spouštěcí model interpretu byl tedy vybrán *indirect threading*.

Proces spouštění aplikace bude následující. Program, reprezentovaný zdrojovým kódem, interpret nahraje do paměti a začne kompilovat do vnitřní reprezentace. Nabízí se tu možnost, využít k načítání aplikace již předkompilovaného kódu pro rychlejší start, ale jednak by tím složitost interpretu vzrostla a jednak rychlost kompilace je velmi vysoká díky paměťové reprezentaci, viz dále.

Dalšími rozhodnutími by byl výběr rozložení paměti, vláknový model, přítomnost a typ garbage kolektoru. Pokud bude instrukční sada a samotný interpret navržen dostatečně flexibilně, je možné naprogramovat garbage kolektor a vláknový model v aplikacích spouštěných na virtuálním stroji. Podpora většího spektra vláknových modelů nebude v tomto interpretu implementována opět z důvodu větší složitosti takového virtuálního stroje. Pokud by byla vyžadována, nejjednodušším modelem by byla podpora zachytávání událostí externího časovače pro preemptivní multitasking. Kooperativní multitasking bude možno ve stávající implementaci doprogramovat v případě potřeby v uživatelských aplikacích.

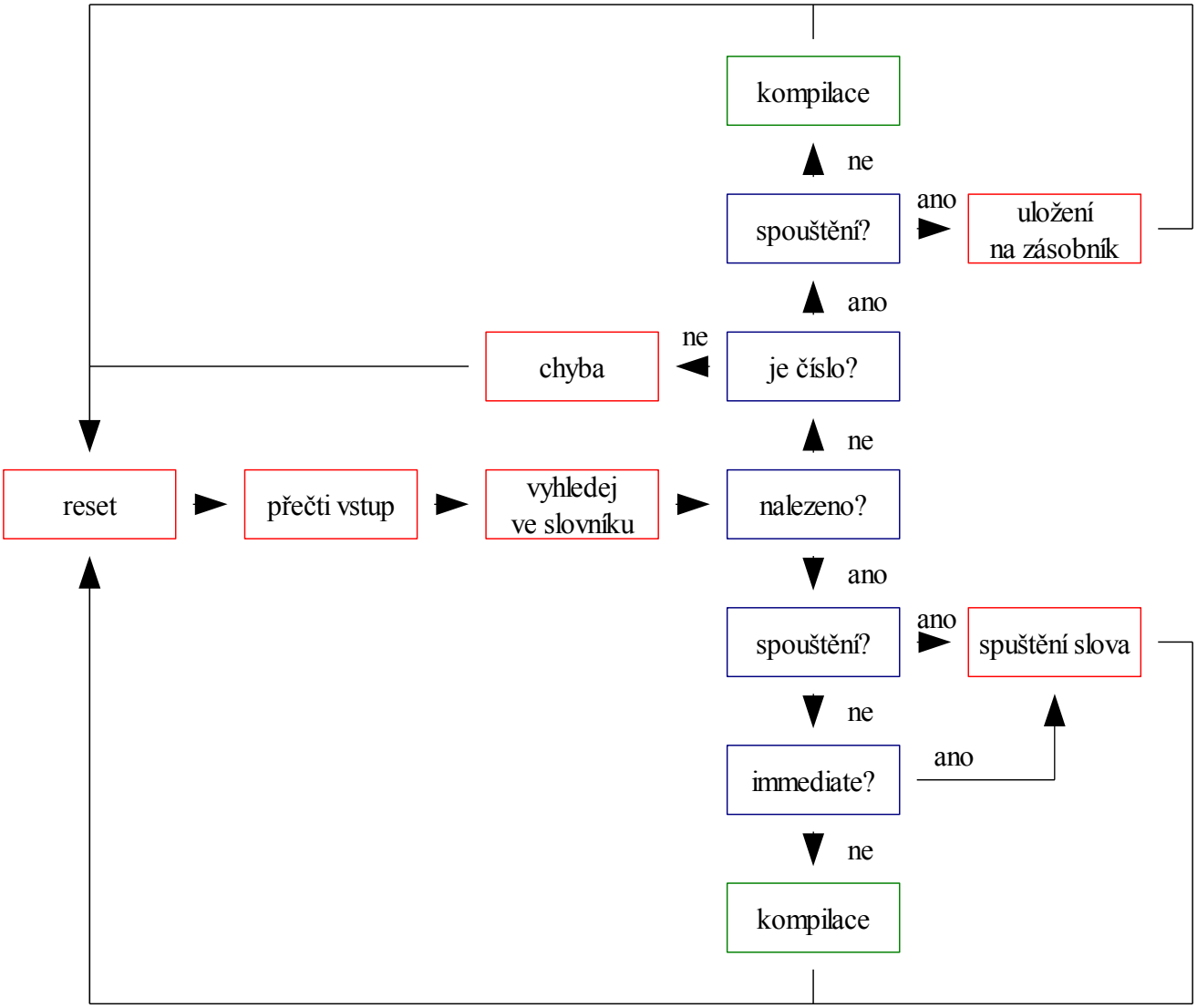
V dalším textu se bude využívat pojmu *slovo*, které v tomto kontextu znamená analogii ke klasické funkci v ostatních programovacích jazycích. Nejedná se o *slovo* procesoru, které odpovídá paměťové buňce. Dalším pojmem je slovník, kterým se označuje datová struktura obsahující všechna slova interpretu.

Smyčka interpretu

Smyčkou vnějšího interpretu je myšlen proces zpracovávání vstupních dat, tedy zdrojových kódů aplikace. Zdrojový kód bude načítán po slovech oddělených bílými znaky. Každé slovo se ihned zakompiluje nebo spustí v závislosti na stavu interpretu. Vzdáleně lze tento proces přirovnat k jednofázové kompilaci. Pokud není smyčka interpretu uživatelsky přeprogramována, data samotného zdrojového kódu jsou hned odhazována. Virtuální stroj bude nabízet prostředky zachytávání vstupního zdrojového kódu, což vede k obrovské

flexibilitě, a například je tímto způsobem možné psát dle potřeby parsery již avizovaných doménových jazyků.

Obrázek 3.4 pro názornost shrnuje smyčku vnějšího interpretu.



obr. 3.4: Smyčka vnějšího interpretu

Začíná se voláním *reset*. Tato část, resetuje zásobník návratových adres. Po přečtení vstupu a jeho vyhledání ve slovníku záleží, zda bylo slovo na vstupu nalezeno, nenalezeno, je číslo, není, jestli je kompilátor ve stavu spouštění a na příznacích nalezeného slova.

Struktura paměti

Pro virtuální stroj typu Forth jsou potřeba tři hlavní paměťové oblasti. Datový zásobník, návratový zásobník a paměť typu RAM. Dále jsou nutné registry programového čítače, pracovního registru, ukazatel na datový zásobník a ukazatel na zásobník návratových adres. Teoreticky je možné mapovat všechny tyto oblasti do paměti RAM, ale pokud to jde, budeme alokovat registry virtuálního stroje z registrů procesoru, protože se s nimi pracuje nejčastěji.

Všechno ostatní tedy bude v lineární oblasti RAM. Spolu s proměnnými interpretu je nutné vyhradit místo pro vstupní buffer pro načítání slov ze zdrojového kódu.

obr. 3.5: Mapa paměti virtuálního stroje

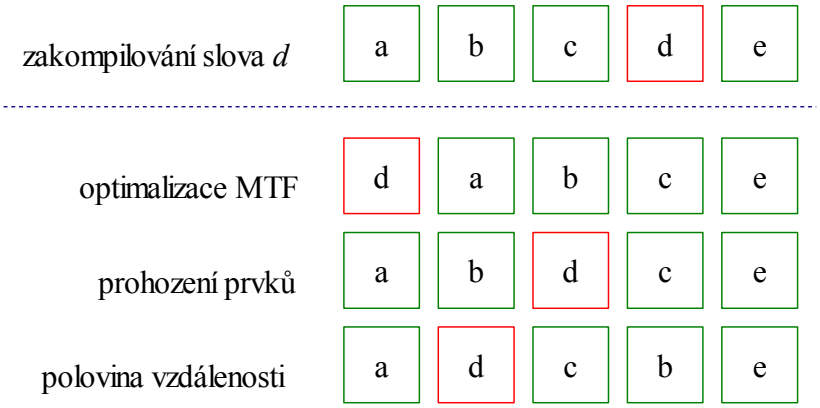
Pro běh virtuálního stroje postačí tři adresovatelné proměnné

- `state` – určuje, jestli je interpret ve stavu kompilace nebo spouštění
- `latest` – ukazatel na poslední zakompilované slovo
- `here` – ukazatel na volnou uživatelskou paměť

Pokud budeme uvažovat 32 bitovou architekturu, potom 12 bajtů zaberou proměnné interpretu, velikost vstupního bufferu se může pohybovat do 32 bajtů, z dalšího textu bude jasné proč. Vstupní buffer musí pojmout celé jedno slovo ze zdrojového kódu. Ani v ostatních jazycích programátoři obvykle nenazývají názvy funkcí nebo proměnných delší. Velikost datového a návratového zásobníku záleží na typu aplikací, které se budou provádět. Typicky pro 32 bitové architektury stačí hodnoty kolem 128 bajtů a výše pro jednoduché aplikace, řádově stovky bajtů pro složité. Ovšem za předpokladu, že aplikace nebude využívat zásobníky pro hluboké rekurze. Výchozí hodnota byla zvolena 512 bajtů. Velikost uživatelské paměti opět záleží na charakteru aplikací, které na interpretu poběží, výchozí hodnota udává velikost 32 kB.

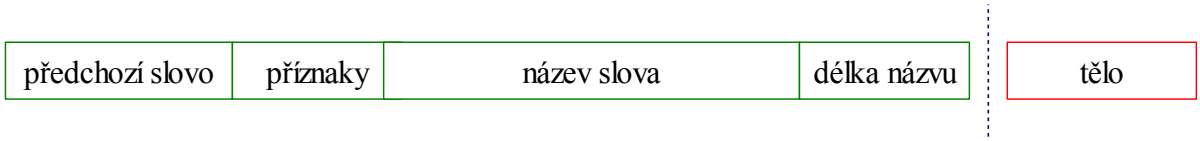
Nejen při kompilaci nových slov je potřeba alokovat paměť. O tento proces se stará jednoduchý *bump-the-pointer* alokátor. Proměnná `here` ukazuje na volnou paměť. Při požadavku na alokaci se jednoduše proměnná zvýší o velikost požadované alokace. Uvolnění paměti probíhá obdobně, pouze snížením ukazatele v `here`.

Dalším důležitým faktem je reprezentace zakompilovaných slov. Nejpřístupnější dynamickou datovou strukturou je jednoduše spojový seznam. Ten má ovšem nevýhodu v lineární časové složitosti vyhledání prvku. Tato nevýhoda se může projevit pouze v rychlosti kompilace, nikoliv spouštění, takže pro malé aplikace, které neobsahují mnoho slov není podstatná. U větších aplikací se může projevit delší čas překladu, to lze kompenzovat použitím samoorganizujících se seznamů. V úvahu přichází optimalizační metoda *move-fo-front* (MTF), prohození sousedních prvků seznamu při jeho čtení nebo prohození prvků v závislosti na vzdálenosti od počátku seznamu. V případě MTF se ukazatel na zakompilované slovo umístí na začátek seznamu, v případě prohazování prvků se při kompilování prohodí slovo se svým předchůdcem, a tak se pomalu posouvá k začátku seznamu. Prohození s prvkem, který je v polovině vzdálenosti od hlavičky seznamu je kompromisem mezi optimalizací MFT a prohozením sousedních prvků.



obr. 3.6: Optimalizace samoorganizujícím se seznamem

Pokud bude platit Paretův princip, kdy se menšina slov využívá ve většině kódu, potom mají tyto optimalizace smysl.

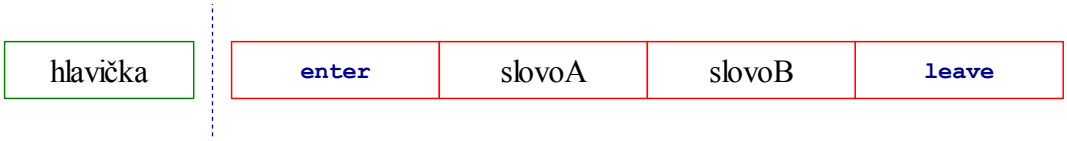


obr. 3.7: Struktura slova

Samotná struktura slova je načrtnuta na obr. 3.7. Paměťovou oblast rezervovanou pro slovo lze rozdělit na hlavičku a tělo. V hlavičce jsou obsaženy informace o názvu, příznacích slova a ukazatel na předchozí slovo. Délka názvu slova bude v hlavičce obsažena dvakrát. jednou v příznacích slova a podruhé za názvem slova. Je to z důvodu možnosti zpětného disassemblování, aby z ukazatele na tělo slova šla spočítat adresa jeho hlavičky. Na 32 bitovém systému bude tedy hlavička slova 6 bajtů dlouhá (4 bajty zabere ukazatel na předchozí slovo, pro příznaky a délku slova je potřeba rezervovat 1 bajt) a ještě se musí připočíst délka názvu slova. Pro příznaky je potřeba rezervovat tři bity, a protože pro uložení délky názvu slova zbývá 5 bitů, vyplývá z toho omezení pro maximální délku názvu slova, která je 32 bajtů. Příznaky lze využít k následujícím označení

- *immediate* bit – označuje, zda je slovo typu *immediate*, tedy slovo, které je možné spustit i v průběhu kompilace
- *hidden* bit – využívá se při kompilaci nového slova nebo k označení privátních slov. Více v podkapitole o jazykových strukturách.
- *code* bit – zaznamenává informaci o tom, jestli je slovo implementováno ve strojovém kódu, nebo se jedná o slovo složené z ostatních. Uplatňuje se pouze při disassemblování. Tento bit by bylo možné vynechat a najít pro něj jiné uplatnění, informace o tom, že je se nejedná o instrukci interpretu lze zjistit z faktu, že první paměťová buňka těla slova obsahuje ukazatel na tělo instrukce `enter`.

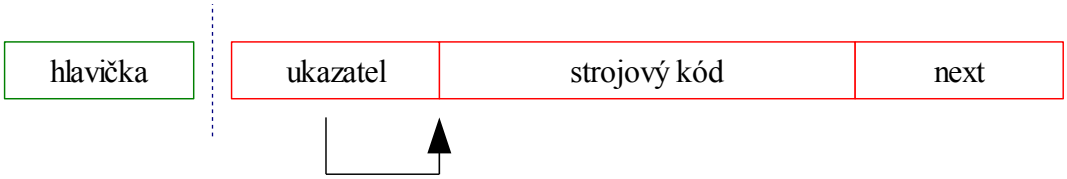
Tělo slova je pak seznamem ukazatelů na těla ostatních slov, tak jak je výše uvedeno v části popisující model *indirect threading*.



obr. 3.8: Struktura slova interpretu

Fakt, že tělo obsahuje ukazatele znamená, že slovo na 32 bitových systémech dále spotřebuje dalších minimálně 8 bajtů pro instrukce `enter` a `leave`.

Instrukce interpretu se skládá ze stejné hlavičky jako slovo, rozdílný je obsah těla. První paměťová buňka těla neobsahuje ukazatel na tělo instrukce `enter`, ale ukazatel na samotný strojový kód instrukce, jak je načrtnuto na obr. 3.9.



obr. 3.9: Struktura těla instrukce interpretu

Implementace instrukce interpretu musí být zakončena kódem vnitřního interpretu *next* nebo v případě řídících instrukcí `jump`, `branch` a `not-branch` změnou programového čítače a pracovního registru podle odpovídajícího relativního ofsetu.

V tabulce 3.2 jsou shrnuty velikosti datových reprezentací instrukcí interpretu a slov na jednotlivých architekturách.

	instrukce	slovo + `enter` + `leave`
16 bit	6 B	8 B
32 bit	10 B	14 B
64 bit	18 B	26 B

tab. 3.2: Velikosti datových reprezentací instrukcí a slov interpretu

3.1.3 Instrukční sada

Dalším faktorem pro jak paměťovou, tak časovou efektivitu celého systému je výběr instrukční sady. Obzvlášť u zásobníkových architektur to může ovlivňovat snadnost programování. Vychází to z faktu, že instrukce zásobníkových architektur mají v naprosté většině případů přístup pouze k zásobníkům, nad nimiž provádí operace. Operace nad zásobníkem mohou zvyšovat nebo snižovat velkost zásobníku. Například operace sčítání vezme první a druhý prvek na zásobníku, sečte je a pak má několik možností, jak s výsledkem

naložit. Jednou je odstranění načtených operandů ze zásobníku a vložení výsledku místo nich. Nebo hodnotou součtu může přepsat jen vrchol zásobníku, takže po operaci zůstane na dvou posledních položkách zásobníku první operand a výsledek operace. Popřípadě obě původní položky zachová a výsledek vloží na nový vrchol zásobníku.

```
žravý přístup              ( a b -- a+b )
líný přístup               ( a b -- a b a+b )
kombinovaný přístup        ( a b -- a a+b )
```

kód 3.4: Porovnání typů instrukčních sad

Běžně se využívá žravého přístupu, důvodem je abstraktní představa zásobníkových buněk jako objektů. V reálném světě se při vyžívání objektů samotný objekt neklonuje, ale přesouvá (pokud se vezme kladivo ze stolu, nezůstane na ponku, ale budeme ho mít pouze v ruce), tento princip je aplikován na operace nad zásobníkovými buňkami. Při návrhu instrukční sady se budeme držet těchto principů a budeme tedy využívat žravý přístup. Diskuse k vhodnosti výběru je provedena v závěru práce.

Úplný výčet všech instrukcí interpretu je uveden v příloze, takže zde nebudeme každou instrukci zvlášť rozebírat, ale pouze načrtneme skupiny instrukcí a popíšeme pouze vybrané. Implementovaná instrukční sada se nesnaží minimální, mnohé instrukce jsou přidány pouze z důvodu větší efektivity provádění. Celou instrukční sadu můžeme rozdělit do kategorií:

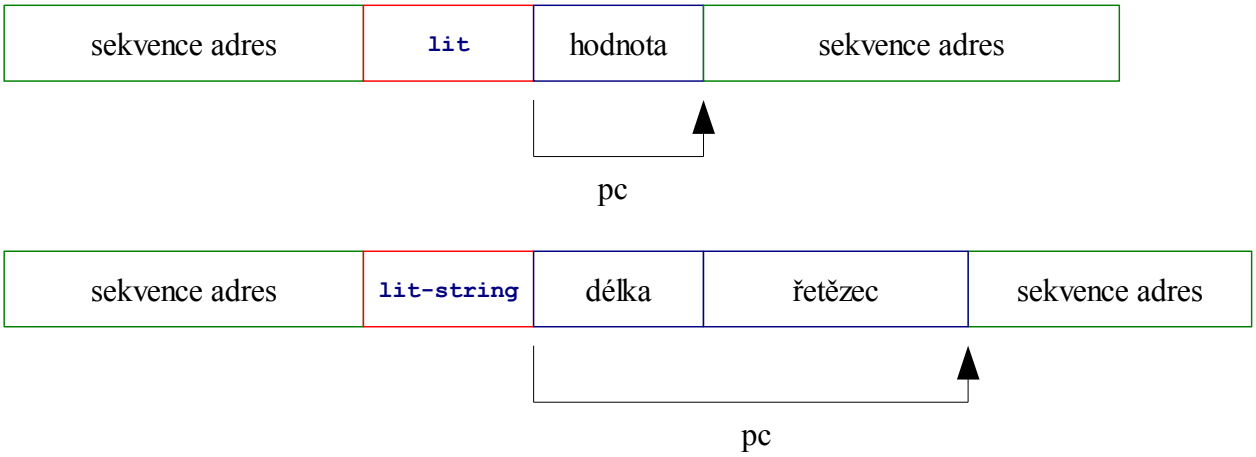
- zásobníkové instrukce – zahrnují operace nad zásobníkem od vložení čísla nebo řetězce na zásobník, až po rotace vrcholu zásobníku
- aritmetické instrukce – sčítání, odčítání, násobení a dělení. Dělící instrukce je specifická tím, že na zásobník uloží jak výsledek dělení, tak zbytek po dělení. Pokud aplikace požaduje například pouze operaci modulo, odstraní si výsledek po dělení ze zásobníku. Většinou je nutné pro spočítání modula stejně obě čísla vydělit, takže se rozdělením `divmod` nic neušetří.
- bitové instrukce – mezi ně patří běžně používané bitové operace and, or, xor, bitová inverze a posuny
- instrukce přenosu kontroly – implementovány jsou tři, nepodmíněný skok, a dva druhy podmíněných. Jeden provádí větvení, pokud na zásobníku je 0, druhý pokud na zásobníku 0 není.
- logické instrukce – byla implementována pouze operace `not`. Všechny další lze v případě potřeby sestavit z `not` a bitových operací
- porovnávací instrukce – porovnávají první a druhou položku zásobníku. Interpret obsahuje operace je rovno, je menší a je menší nebo rovno.
- instrukce pro práci z pamětí – zajišťují čtení a zapisování z / do paměti včetně operací pro čtení z pole a kopírování bloku paměti. Instrukce pracují s bajtově

adresovatelnou pamětí.

- konstanty – aby byl interpret portovatelný na jiné platformy, musí obsahovat informaci o velikosti standardní nativní buňky. K tomuto účelu slouží konstanta `cell`.
- proměnné – přístup ke 4 základním proměnným interpretu – stav interpretu, ukazatel na poslední zakompilované slovo, ukazatel na volnou paměť a ukazatel na vstupní buffer, který není nezbytně nutný, ale opět je zařazen z důvodu lepší přenositelnosti mezi platformami a větší flexibilitě při nastavování kompilačních parametrů zdrojového kódu interpretu.
- základní instrukce pro práci se vstupem a výstupem – `key` a `emit` se starají o čtení ze vstupu a zapisování do výstupu. Vždy tak činí po jednom bajtu.
- instrukce interpretu – aby bylo možno zanořovat volání slov, musí interpret obsahovat instrukce pro tuto činnost. Implementovány jsou základní dvě, `enter` a `exit`, a ještě instrukce `leave`, která je aliasem instrukce `exit`, a je to z výše uvedeného důvodu možnosti disassemblování.
- definující instrukce – je implementována pouze jedna základní a ve skutečnosti není pravou instrukcí, ale složeným slovem interpretu z ostatních instrukcí. Po přečtení slova ze vstupu zařídí vytvoření hlavičky nového slova a přepne do režimu kompilace
- ostatní – sem lze zařadit instrukce, které vykonávají činnost, kterou potřebuje vykonávat i samotný interpret ke svému běhu. Takže tím, že své vlastní funkce exportuje pro použití, navýší se jeho velikost pouze o pár bajtů. Mezi takovéto instrukce patří například převod z řetězce na číslo, porovnávání řetězců, načtení slova ze vstupu, vyhledání slova ve slovníku, vytvoření hlavičky slova, přepínání mezi stavem spouštění a kompilace, změny příznaků slova, ukončení interpretu a samotná smyčka vnějšího interpretu

Celkem je k dispozici 64 instrukcí nebo slov exportovaných interpretem pro použití. Tento počet lze ještě významně zredukovat a implementovat instrukce ne ve strojovém kódu, ale v samotném programovacím jazyku interpretu. Bude to, ale na úkor efektivity provádění, protože musí být několikrát volán vnitřní interpret *next*, aby spustil všechny instrukce složeného slova.

Mezi základní zásobníkové instrukce lze zařadit `lit` a `lit-string`. První jmenovaná slouží k uložení čísla na zásobník, druhá k uložení velikosti a ukazatele na řetězec. Protože interpret očekává tělo slova složené pouze z ukazatelů na těla dalších slov, musí se při vykonávaní instrukcí `lit` a `lit-string` posunout programový čítač o velikost hodnoty nebo řetězce, které jsou při kompilaci uloženy ihned za jmenované instrukce. Obr. 3.10 ilustruje případ spouštění instrukcí `lit` a `lit-string`.



obr. 3.10: Instrukce `lit` *a* `lit-string`

Instrukce `jump`, `branch`, `not-branch` fungují na podobném principu jako `lit`. Při spouštění načtou relativní ofset na programovém čítači, ten inkrementují a popřípadě na načtený ofset skočí.

Jak již bylo zmíněno instrukce `enter`, `leave` a `exit` slouží k zanořování volání, popřípadě k návratu z volání. `Enter` uloží aktuální programový čítač na zásobník návratových adres a zvýší jeho adresu o jedničku. Instrukce `leave` a `exit` doplňují předešlou instrukci, při jejich spuštění načtou ze zásobníku návratových adres hodnotu uloženého programového čítače a do aktuálního tuto hodnotu načtou.

Velmi využívanou instrukcí, zejména při kompilaci, je `,` (čárka). Uloží do paměti ukazatel na volnou paměť a ukazatel inkrementuje.

Užitečnou instrukcí interpretu je `execute`, která odečte ze zásobníku adresu těla slova a slovo provede. Po dokončení spouštění slova se kontrola vrací za instrukci `execute`. Takovýmto způsobem se dají jednoduše vytvářet tabulky skoků nebo další užitečné jazykové konstrukce, viz například slovo *run* v souboru *tests.forth* na přiloženém CD.

3.1.4 Proměnné, spouštění a kompilace

Interpret zpřístupňuje své stavy programovacímu jazyku přes volání svých instrukcí, nebo přes proměnné. Základními třemi jsou

- `state` – stav interpretu, indikuje zda se nachází ve stavu kompilace nebo spouštění
- `latest` – obsahuje ukazatel na naposledy zakompilované slovo
- `here` – obsahuje ukazatel na volnou paměť, do které se ukládají uživatelské proměnné nebo kompilují nová slova

Při startu celého systém se nejdříve nastaví interpret do stavu spouštění, zavolá se slovo `reset`, které resetuje zásobník návratových adres, spustí instrukci `interpret` obsahující vnější smyčku interpretu, viz obr. 3.4, a následně zavolá samo sebe. K přetečení zásobníku nedojde,

protože vždy na začátku je zásobník návratových adres vymazán. Slovo `reset` musí být složené slovo, aby vnitřní interpret mohl inkrementovat čítač instrukcí a skočit na další slovo.

V několika odstavcích popíšeme vytváření nových slov a kompilace jejich těl. Kompilace slova probíhá tak, že se nejprve na adrese volné paměti vytvoří hlavička slova a za ní se typicky uloží ukazatel na tělo `enter`. Při předefinovaném vytváření slov je možné ukládat místo instrukce `enter` jakoukoliv jinou, nově definované slovo se pak bude chovat jako alias vložené instrukce. Nesmí se ovšem jednat o ukazatel na tělo složené slova. Vnitřní interpret očekává na tomto místě ukazatel do strojového kódu, takže by pak nově vytvořené spouštěné slovo skončilo s chybou. O tvorbu popsané hlavičky se stará instrukce `create`, která navíc ještě posune ukazatel na volnou paměť o délku vytvořené hlavičky. Proto, aby začalo kompilovat tělo slova, je nutné přepnout interpret do stavu kompilace buď uložení do proměnné `state` 1 nebo zavoláním slova `]`, které provádí tutéž činnost. V další podkapitole bude jasné, proč byla zrovna zvolena reprezentace pravou hranatou závorkou. Ve stavu kompilace tělo každého nalezeného slova je uloženo na adresu ukazatele volného místa `here` a tento ukazatel se zvýší o jednu paměťovou buňku. Pokud se kompiluje číslo, vloží se na adresu volné paměti instrukce lit, do následující paměťové buňky se vloží hodnota čísla a opět se `here` posune o patřičnou velikost, tedy o velikost 2 buněk. Aby bylo možno přestat kompilovat a začít spouštět ostatní slova, je nutné přepnout interpret zpět do stavu spouštění. Aby se tak stalo, musí existovat slovo typu *immediate* a musí uložit hodnotu 0 do `state`. Interpret obsahuje instrukci `[`, která potřebné operace provede.

3.1.5 Jazykové struktury

Pro programovací jazyk typu Forth je specifické to, že samotný interpret v sobě většinou neobsahuje jazykové struktury jako například podmínky a cykly, které jsou základem v ostatních programovacích jazycích. Interpret naopak obsahuje jednoduché nástroje, se kterými potřebné jazykové struktury lze vytvořit bez většího úsilí. Nemusí zůstat pouze u základních, meze jsou omezené pouze dostupnou pamětí interpretu.

Tvorba slov

Základním stavebním kamenem je tvorba nových slov, která byla popsána v předchozí kapitole. Aby bylo možné začít vytvářet slova, je nejdříve nutné vytvořit slovo pro jejich kompilování do slovníku. Může se vytvořit za běhu interpretu, ale je výhodnější mít takové slovo umístěno přímo v interpretu, protože to je jednak jednodušší a rozhodně ne zbytečné, protože snad každý program využívá vkládání slov do slovníku. Obvykle se využívá názvu slova `:`. Po jeho spuštění se do ukazatele `latest` uloží aktuální adresa volné paměti, tedy začátek nového slova, následně se vytvoří hlavička, zakompiluje `enter` a interpret se přepne do stavu kompilace. Následující slova nebo čísla na vstupu již nebudou spouštěna, ale kompilována. Pokud chceme kompilované slovo ukončit, můžeme využít instrukce `;`, která se postará o ukončení slova instrukcí `leave` a přepne interpret zpět do stavu spouštění. Příklad definice nového slova je znázorněn v úryvku 3.5. V závorkách je uveden zásobníkový efekt

nového slova. Závorka *(* je mimochodem také funkčním slovem, které musí být vytvořeno, a pokud má fungovat jako komentář, jednoduše začne načítat vstup instrukcí `key` až do chvíle, než objeví koncovou závorku.

```
: square ( a a -- a*a ) dup mul ;
```

kód 3.5: Ukázka tvorby nového slova

V průběhu kompilace slova nebo kdykoliv před kompilací dalšího, je možné zavolat instrukci `immediate`, která nastaví typ slova na *immediate*, takže při dalším výskytu se slovo spustí i když bude interpret ve stavu kompilace.

Makra a JIT kompilace

V kódu 3.6 je znázorněno použití instrukcí `,`, `[` a `]`. Snad ospravedlňuje jejich pojmenování.

```
: sedmicka lit [ 7 , ] ;
```

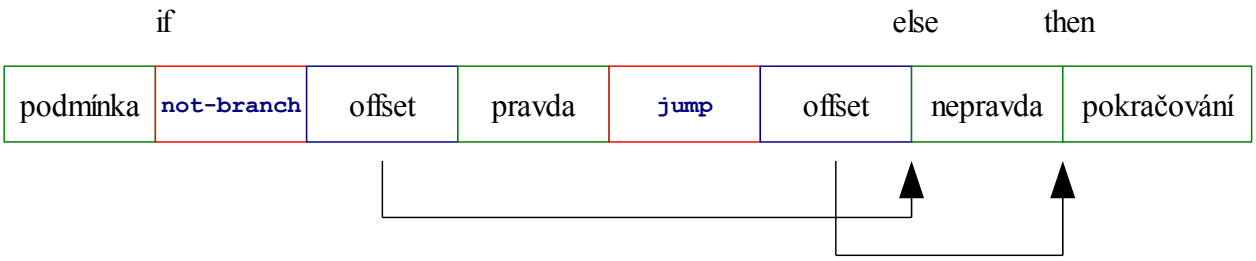
kód 3.6: Příklad JIT kompilace

Výše uvedená ukázka vytvoří slovo *sedmicka*, zakompiluje instrukci `lit` a následně se interpret přepne do stavu spouštění. V něm se uloží na zásobník číslo sedm a uloží ho do volné paměti, tedy za zakompilovanou instrukci `lit` a přepne se zpět do stavu kompilace, kde už jenom vytvoří konec slova pomocí `;`. Takto nově vytvořené slovo při spouštění pouze uloží na zásobník číslo 7, ale za pomocí tohoto principu lze vytvářet makra nebo optimalizovat program za běhu JIT kompilací.

Podmínky a ostatní jazykové struktury

Další základní jazykovou strukturou jsou podmínky. V ostatních programovacích jazycích jsou podmínky kompilovány většinou do strojového kódu nebo bytekódu ve formě skoků na patřičné adresy. V případě jazykové konstrukce *if*, když podmínka není vyhodnocena pravdivě, podmíněný skok skočí za blok kódu provádějící se, když podmínka je splněna. V případě, že je dále obsažena druhá větev *else*, na konci pravdivého bloku se umístí skok za blok nepravdivý.

Analogicky budeme postupovat při vytváření podmínek v interpretu. Výsledný kód slova obsahující podmínky by měl vypadat podobně jako na obr. 3.11.



obr. 3.11: Struktura podmínky if-else-then

Slova *if else* a *then* musí mít nastavený příznak *immediate*, protože se většinou budou vyskytovat ve stavu kompilace. Pokud ve stavu kompilace přijde řada na *if*, spustí se a na pozici `here` uloží adresu na tělo `not-branch`, za ní číslo 0 jako rezerva pro relativní ofset, který bude spočítán později, a na zásobník uloží `here`. Po vykonání *if* interpret kompiluje blok pravdivé větve až do chvíle než narazí na *else*. *Else* se spustí podobně jako *if*, zakompiluje `jump` a rezervu pro ofset. Nakonec spočítá relativní ofset díky předešlé adrese uložené na zásobníku, na onu adresu uloží spočítaný ofset a na zásobník vloží `here`, nyní ukazující na paměťovou buňku za `jump`. Interpret pak opět pokračuje v kompilaci až ke slovu *then*, které provede tytéž operace s relativními ofsety, jen už na zásobník neukládá `here`.

Analogicky se vytvářejí ostatní jazykové konstrukce jako cykly, konstrukce *switch*, konstanty, proměnné, anonymní slova, rekurze, řetězce a další. Popis všech ostatních by byl příliš zdlouhavý, jejich samotný kód lze shlédnout a vyzkoušet na přiloženém CD v souboru *extensions.forth*.

Rozšíření

Jako nástroje pro diagnostiku vlastního stavu byla implementována slova:

- . – vypíše vrchol zásobníku
- *?* – vypíše hodnotu na zadané adrese
- *stack* – vypíše obsah zásobníku
- *word-list* – vypíše seznam všech slov
- *words* – vypíše seznam slov začínající na daný řetězec
- *see* – disassembluje zadané slovo

Všechna slova lze najít opět v souboru *extension.forth*.

3.2 Mikroprocesor

V [44] je popsáno mnoho mikroprocesorových architektur, které se v minulosti vyznačovaly svými neobvyklými vlastnostmi a někdy poskytovaly zcela odlišný a přínosný pohled na procesorové architektury. Dnes běžné mikroprocesory většinou využívají

registrovou architekturu a jsou založeny na podobných principech ve smyslu klasického vykonávání strojových instrukcí. V mnoha takovýchto případech je nepohodlné programovat přímo ve strojovém kódu, a tak se volí vyšší programovací jazyk, který ovšem nevyhnutelně ztrácí kontrolu nad jednotlivými detaily architektury. V této sekci bude proveden návrh architektury a instrukční sady mikroprocesoru, který by byl vhodný pro běh interpretu implementovaného v této práci.

Nový interpret a mikroprocesor bude mířit do oblasti vestavěných systémů, a proto zvolíme 16 bitovou architekturu spíše než 32 bitovou. Pokračováním této práce by mohlo být i srovnání návrhů pro obě architektury.

3.2.1 Instrukční sada

Hlavními požadavky na navrhovanou architekturu jsou jednoduchost a co Největší usnadnění práce interpretu. Jako nejlepší spouštěcí model interpretu implementovaném na nové architektuře se jeví *subroutine threading*. Na nový interpret odpadá požadavek přenositelnosti a instrukční sada procesoru se dá navrhnout s ohledem na právě vybraný spouštěcí model, takže zůstává pouze výhoda tohoto modelu, a tou je efektivita. Navíc se nemusí zavádět extra registry pro instrukční ukazatel a ukazatele na vrcholy zásobníků, využijí se již stávající na mikroprocesoru.

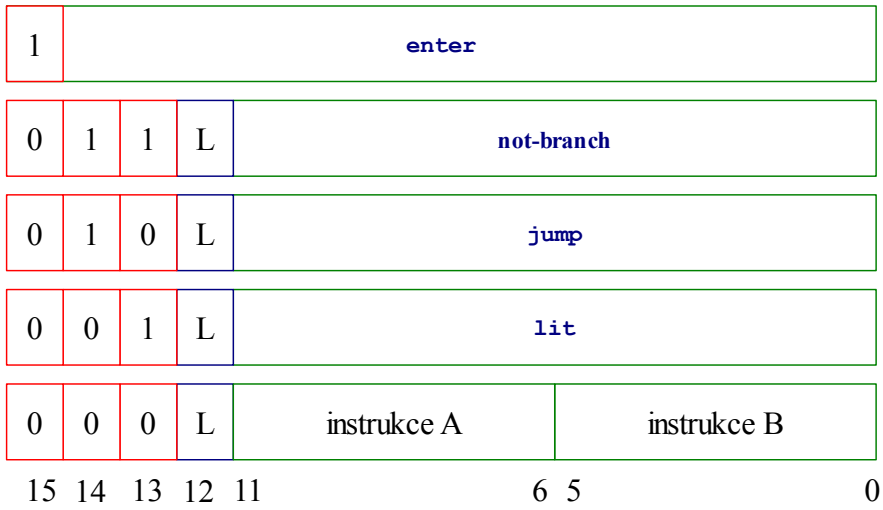
Abychom mohli navrhnout efektivní reprezentaci instrukcí procesoru v paměti a jejich efektivní vykonávání, je třeba znát alespoň přibližné frekvence výskytu instrukcí. Dynamická popisuje četnost spouštěných instrukcí, statická udává frekvenci instrukcí uložených v paměti. Podle [47] jsou četnosti vykonávaných instrukcí zásobníkového procesoru zhruba následující:

instrukce	četnosti instrukcí	
	dynamické	statické
`enter`	~12%	20-30%
`exit`	~12%	~8%
`not-branch`	~5%	~3%
`lit`	~2-3%	~9%
`fetch`	~1-3%	~4%
`dup`	~1-3%	~1-3%
`to-rs`	~1-3%	~1-3%
`from-rs`	~1-3%	~1-3%
`swap`	~1-3%	~1-3%

tab. 3.3: Četnosti instrukcí zásobníkového procesoru

Z tab. 3.3 vyplývá, že instrukční sadu je třeba navrhnout tak, aby co nejvíce usnadňovala vykonávání instrukcí `enter` a `exit` a zároveň, aby velikost jejich reprezentace byla minimální. Pod druhý požadavek na délku instrukce spadá i `lit`.

Vzhledem k těmto omezením bylo navrženo instrukční slovo zachycené na obr. 3.12.



obr. 3.12: Instrukční slovo mikroprocesoru

Instrukce se z paměti čtou po 16 bitech. Pokud horní bit bude 1, provede se instrukce `enter` s adresou uloženou v dalších 15 bitech. Stále ale bude možné využít celý 16 bitový adresový prostor, protože stanovíme podmínku, že instrukce `enter` může adresovat paměť pouze po 2 bajtech, tedy že 15 bitová adresa za prvním bitem je adresa posunutá o 1 bit doleva. Z tohoto omezení vyplývá nutnost zarovnávat počátky těl na dvoubajtové hranice, což ale vzhledem k délce celého slova nebude většinou tvořit velký problém. Dékódování dalších instrukcí probíhá tak, že pokud horní tři bity jsou nenulové, provedou se odpovídající instrukce `not-branch`, `jump` nebo `lit` s 12 bitovým parametrem. U instrukcí `not-branch` a `jump` není toto omezení problémem, protože 12 bitový parametr je relativní adresa a slova v programovacím jazyce typu Forth mívají mnohem menší délku, takže velikost parametru je více než dostatečná. V případě instrukce `lit` to znamená, že pokud chceme na zásobník vložit číslo větší než 12 bitů, je nutné využít další instrukci. V posledním případě jsou načteny dvě 6 bitové instrukce. Pokud je *instrukce A* nulová, potom se první 4 bity vrcholu zásobníku doplní spodními 4 bity *instrukce B*. Jedná se tedy o doplněk, pokud v předešlém cyklu bylo potřeba vložit na zásobník číslo přesahující 12 bitů. Příznak *L* na 12 bitu označuje, že po vykonání instrukcí `not-branch`, `jump`, `lit` nebo *instrukce B* se provede `leave`. Ze zásobníku návratových adres se tedy načte adresa a uloží se do programového čítače. Pokud by bylo potřeba vyskočit z podfunkce hned za *instrukcí A*, *instrukce B* bude obsahovat `nop`, tedy nebude se na tomto místě provádět žádná operace.

Díky tomuto návrhu jsou splněny požadavky zadané četnostmi výskytu slov. Pouze v některých případech je nutné provést 2 instrukce lit pro vložení plného 16 bitového čísla na zásobník. Navíc je možné provést rovnou 2 instrukce při jednom čtení z paměti. Oproti implementovanému interpretu běžícímu na jiných 16 bitových platformách se většina instrukcí v těle slov zkrátí o 1 bajt tedy na polovinu, a instrukce s parametrem jako lit, `jump` a `not-branch` dokonce o 2 bajty, opět na polovinu. Navíc odpadá nutnost explicitně `enter`

a leave, což znamená, že by se měla původní paměťová reprezentace aplikace zmenšit o více jak polovinu, pokud nepočítáme hlavičky definovaných slov. Rychlost interpretu bude navíc shodná s rychlostí aplikace běžící na procesoru bez interpretu. Stalo se tak díky eliminaci vnitřního interpretu (popřípadě jeho přesunu na mikroprocesor), zůstane pouze vnější pro řízení spouštění a kompilace.

Na druhou stranu bude muset být kompilátor o něco složitější, protože je potřeba instrukce správně zakódovat, ale tento fakt nebude nejspíš představovat problém, protože kompilování konkrétní instrukce není kritické místo, co se týká časové náročnosti.

Další vývoj mikroprocesorové architektury je diskutován v závěru.

3.2.2 Asynchronní architektura

Asynchronní architektura [45] vychází z toho, že jednotlivé logické celky obvodu nejsou řízeny hodinovým signálem, ale synchronizaci si celky zajišťují komunikací s obvody, na kterých jsou závislé a na kterých závisí. Až je výpočet v jedné části hotov, vyšle se na vstupy závislého obvodu spolu s příznakem, že je možné si výsledky převzít. Toto má spoustu důsledků, od snížené spotřeby, elektromagnetického vyzařování, tak až po zvětšení počtu tranzistorů, což může spotřebu naopak zvýšit.

Pro zhodnocení aplikace asynchronní architektury by bylo nejdříve potřeba dopracovat návrh mikroprocesoru do dalších stádií, aby mělo smysl provádět hlubší analýzu.

4 Závěr

V práci byla provedena analýza vybraných virtuálních strojů od VM určených pro interpretaci aplikací na serverových nebo desktopových počítačů až po VM, které jsou využívány ve vestavěných systémech.

Na základě těchto poznatků byl navržen interpret typu Forth, určený pro běh na širokém spektru architektur s ohledem na fakt, že pro podobný bude navržena instrukční sada procesoru. Interpret byl naprogramován v jazyce C, za použití standardu *stdc99 s GNU* rozšířením *computed gotos*. Měl by tedy být přenositelný na platformy, které jsou podporovány překladačem *GNU gcc*, *clang* a ostatními, které podporují výše uvedený standard a rozšíření. V průběhu implementace se musel několikrát volit způsob, jak lze inicializovat obsah paměti interpretu, protože se postupně naráželo na problémy s přenositelností. Konkrétně jak staticky a portovatelně inicializovat paměť, která obsahuje ukazatele na další paměťové oblasti. Různá řešení měla omezení ve velikosti výsledného kódu nebo v rychlosti inicializace. Nakonec se všechny problémy podařilo vyřešit vygenerováním instrukční sady a počátečního obsahu paměti za pomoci skriptovacího jazyka *Python* a takto vygenerovaný obsah se spolu s kódem interpretu zkompiloval do spustitelného souboru. Tabulka 4.1 shrnuje velikosti interpretu a velikosti obsahu jeho paměti na různých platformách. Druhý sloupec uvádí velikost paměti po startu, třetí sloupec velikost paměti po načtení souboru *extensions.forth*, který obsahuje rozšíření interpretu o podmínky, cykly, *switch* struktury, rekurze, anonymní funkce, práce s řetězci a další včetně diagnostiky vlastního stavu s disassemblerem.

	interpret	RAM jádra interpretu	RAM extensions.forth
32 bit	6188 B	1161 B	4560 B
64 bit	8686 B	1797 B	8404 B

tab. 4.1: Velikosti interpretu a obsahu jeho paměti

Dále v interpretu byly implementovány optimalizace kompilační fáze pomocí samoorganizujích se seznamů algoritmy *move-to-front*, prohozením se sousedem a prohozením s logaritmickým krokem. Ukázalo se, že na testovaných datech si nejlépe vede nejjednodušší optimalizace - algoritmus *move-to-front*. Průměrné časy jsou uvedeny v tab. 4.2. Počítač na kterém probíhaly testy byl osazen procesorem Intel Core 2 Duo 1.5 GHz.

bez optimalizace	move-to-front	prohození	logartimický krok
20,5 s	5, 9 s	14,4 s	6,3 s

tab. 4.2: Měření optimalizací kompilace

Všechny jednotkové testy proběhly úspěšně na 64 bitové x86 platformě s operačním systémem *FreeBSD*, několika 32 bitových x86 platformách s operačním systémem *Linux* a 32

na bitové platformě typu *Win32*.

Výběr instrukční sady byl založen na informaci, že se běžně vybraná instrukční sada používá. Jako rozšíření práce by bylo možné implementovat další typy instrukčních sad a porovnat je mezi sebou. Což znamená mít k dispozici větší množství praktických aplikací, aby bylo možné výsledky objektivně zhodnotit. Dalšími vylepšeními by mohlo být zvážení použití filtru pro ukládání názvů slov pomocí obdoby Lempel-Ziv-Welch komprese, jako v případě analyzované Juice VM a jejího *SlimBinary* formátu.

V části návrhu mikroprocesorové architektury, byla navržena struktura instrukčního slova, která zajišťuje velmi dobré podmínky pro implementaci interpretu typu Forth. Ve srovnání s interpretem spuštěným na jiných 16 bitových architekturách se velikost paměťové reprezentace kódu slov zmenší na polovinu a eliminací vnitřního interpretu se rychlost vykonávání kódu zvětší na rychlost aplikace běžící bez virtuálního stroje. Za následek to bude mít mírné zvýšení složitosti kompilátoru. Návrh mikroprocesoru skončil z časových důvodů pouze u této fáze. Pokračování by bylo architekturu dospecifikovat, implementovat a provést testy.

Reference

[01] **Intel Corporation**: Intel® Core™ i7 Processor Family for the LGA-2011 Socket Datasheet Volume 1, 2011, http://www.intel.com/content/www/us/en/processors/core/core-i7-lga-2011-datasheet-vol-1.html (1.5. 2012)

[02] **Intel Corporation**: Intel® 64 and IA-32 Architectures Software Developer's Manual, 2011, http://download.intel.com/products/processor/manual/325462.pdf (1.5. 2012)

[03] **Tim Lindholm, Frank Yellin**: The Java™ Virtual Machine Specification Second Edition, 1999, http://docs.oracle.com/javase/specs/jvms/se5.0/html/VMSpecTOC.doc.html (1.5. 2012)

[04] **Martin Odersky, Philippe Altherr a kol.**: An Overview of the Scala Programming Language Second Edition, 2006, École Polytechnique Fédérale de Lausanne

[05] **Rich Hickey**: Clojure reference, http://clojure.org/Reference (1.5. 2012)

[06] **Charles Oliver Nutter**: Mirah Programming Language, http://www.mirah.org/ (1.5. 2012)

[07] **Jim Hugunin, Ted Leung, Frank Wierzbicki a kol.**: The Jyton Project, http://jython.org/ (1.5. 2012)

[08] **Jan Arne Petersen, Charles Nutter, Thomas Enebo, Ola Bini, Nick Sieger a kol.**: JRuby, http://jruby.org/ (1.5. 2012)

[09] **Kresten Krab Thorup a kol.**: Erjang, https://github.com/trifork/erjang (1.5. 2012)

[10] **Kenneth R. Anderson, Timothy J. Hickey a Peter Norvig**: JScheme, http://jscheme.sourceforge.net/jscheme/main.html (1.5. 2012)

[11] **Ian Farmer, Jim Roseborough**: Luaj, http://sourceforge.net/projects/luaj (1.5. 2012)

[12] **Ravenbrook Limited, Nokia Corporation**: Jill, https://code.google.com/p/jillcode/ (1.5. 2012)

[13] **Brian Alliet, Adam Megacz**: Complete Translation of Unsafe Native Code to Safe Bytecode, 2004, Rochester Institute of Technology, University of California, Berkley

[14] **David A Roberts**: LLJVM - Low Lever Java Virtual Machine, http://da.vidr.cc/projects/lljvm/ (1.5. 2012)

[15] **William Kahan**: IEEE Standard 754 for Binary Floating-Point Arithmetic - Lecture notes, University of California, Berkeley

[16] **Sun Microsystems**: The Da Vinci Machine Project, http://openjdk.java.net/projects/mlvm/ (1.5. 2012)

[17] **Sun Microsystems**: The Java HotSpot™ Virtual Machine - technical white paper, 2001, http://java.sun.com/products/hotspot/docs/whitepaper/Java_HotSpot_WP_Final_4_30_01.pdf (1.5. 2012)

[18] **Barry K. Rosen, Mark N. Wegman, F. Kenneth Zadeck**: Global Value Numbers and Redundant Computations, 1988, IBM Thomas J. Watson Research Center, Brown University

[19] **Sun Microsystems**: Memory Management in the Java HotSpot™ Virtual Machine, 2006, http://www.oracle.com/technetwork/java/javase/memorymanagement-whitepaper-150215.pdf (1.5. 2012)

[20] **Sun Microsystems**: J2ME Building Blocks for Mobile Devices - white paper, 2000, http://java.sun.com/products/cldc/wp/KVMwp.pdf (1.5. 2012)

[21] **Gilad Bracha, Tim Lindholm, Wei Tao a Frank Yellin**: CLDC Byte Code Typechecker Specification, 2003, Sun Microsystems, https://iguanadoc.googlecode.com/svn/trunk/Base%20documental/moviles/cldc-1_1-fr-spec/Appendix1-verifier.pdf (1.5. 2012)

[22] **Sun Microsystems**: Connected Limited Device Configuration, 2003, http://download.oracle.com/otndocs/jcp/7247-j2me_cldc-1.1-fr-spec-oth-JSpec/ (1.5. 2012)

[23] **Dan Bornstein a kol.**: Dalvik VM documentation, http://source.android.com/source/downloading.html (1.5. 2012)

[24] **David Ehringer**: The Dalvik Virtual Machine Architecture, 2010, http://www.kiddai.com/NCTU/ebl/dex/The_Dalvik_Virtual_Machine.pdf (1.5. 2012)

[25] **Nik Shaylor, Douglas N. Simon, William R. Bush**: A Java Virtual Machine Architecture for Very Small Devices, 2003, Sun Microsystems Research Laboratories

[26] **Microsoft, Hewlett-Packard, Intel Corporation, Ecma International**: Common Language Infrastructure (CLI) Partitions I to VI, ECMA-335, 5th edice, 2010

[27] **LLVM Team**: Low Level Virtual Machine Documentation, 2011, http://llvm.org/releases/3.0/llvm-3.0.tar.gz (1.5. 2012)

[28] **Silicon Graphics International, CodeSourcery, Hewlett-Packard, Humboldt-Universität**: Itanium C++ ABI Exception Handling, revize 1.22, 2005, http://sourcery.mentor.com/public/cxx-abi/abi-eh.html (1.5. 2012)

[29] **Hewlett-Packard**: HP Itanium®-based Systems programmers guide, verze A.01.15, 1998

[30] **Phil Winterbottom, Rob Pike**: The design of the Inferno virtual machine, 1997, Bell Labs, Lucent Technologies, http://doc.cat-v.org/inferno/4th_edition/dis_VM_design (1.5. 2012)

[31] **Lucent Technologies, Vita Nuova Limited**: Dis Virtual Machine Specification, 2003, Inc http://doc.cat-v.org/inferno/4th_edition/dis_VM_specification (1.5. 2012)

[32] **Vita Nuova Limited**: dis(6) Manual, 2007, http://www.vitanuova.com/inferno/man/6/dis.html (1.5. 2012)

[33] **Charles Antony Richard Hoare**: Communicating Sequential Processes, 2004

[34] **Angelo Corsaro, Corrado Santoro**: Optimizing JVM Object Operations to Improve WCET Predictability, 2004, Washington University, University of Catania

[35] **Michael Franz**: Adaptive Compression of Syntax Trees and Iterative Dynamic Code Optimization: Two Basic Technologies for Mobile-Object Systems, 1997, University of California

[36] **Thomas Kistler a Micahael Franz**: A Tree-Based Alternative to Java Byte-Coddes, 2005, University of California

[37] **Michael Franz, Thomas Kistler**: Slim Binaries, 1997, Communications of the ACM, Vol. 40, No. 12

[38] **FirmWorks**: The Open Firmware Source, 2002, http://firmworks.com (1.5. 2012)

[39] **Forth Inc**: Forth Applications, http://www.forth.com/resources/apps/index.html (1.5. 2012)

[40] **Brad Rodriguez**: Moving Forth, 1993, http://www.bradrodriguez.com/papers/moving1.htm (1.5. 2012)

[41] **Stephen Pelc**: Programming in Forth, 2005, MicroProcessor Engineering Limited

[42] **Anton Ertl**: Threaded Code + Benchmarks V2, http://www.complang.tuwien.ac.at/forth/threading/ (1.5. 2012)

[43] **Technical Committee X3J14**: ANS Forth Specification, American National Standard for Information Systems, http://www.taygeta.com/forth/dpans.html (1.5. 2012)

[44] **John Bayko**: Great Microprocessors of the Past and Present, verze 13.4.0, 2003, http://jbayko.sasktelwebsite.net/cpu.html (1.5. 2012)

[45] **Jens Sparso**: Principles of Asynchronous Circuit Design, 2006, Technical University of Denmark, http://www2.imm.dtu.dk/~jsp/ (1.5. 2012)

[46] **Wayne Wolf**: FPGA-Based System Design, 2004, Pretnice Hall PTR, ISBN 0-13-142461-0

[47] **Philip J. Koopman, Jr**.: Stack Computers: the new wave, 1989, Ellis Horwood Ltd., ISBN: 0-7458-0418-7

Přílohy

Dokumentace interpretu

výpis help.forth:

```
: help
"
lit documentation
=================

      usage
      -----

      exit interpret
            bye

      single line comments
            # this whole line is a comment space, after # is mandatory

      multi line comments
            (
                  multiline comment is also useful for stack notations
                  nested comments are not supported
            )

      word execution
            word-to-execute

      number execution = pushing it on the stack
            777

      interpret instructions
      ----------------------

      abbrevations
            pc          program counter
            tos         top of data stack
            nos         second item of data stack

      stack words
            lit         ( -- num ) inserts number at pc stack
            dup         ( a -- a a ) duplicates tos
            second      ( a b -- a b a ) copies nos
            third       ( a b c -- a b c a ) copies third item
            nip         ( a -- ) drops tos
            tuck        ( a b c -- c a b ) rotates stack
            pull        ( a b c -- b c a ) rotates stack
            from-rs     ( r:a -- a ) move data from return stack
            to-rs       ( a -- r:a ) move data to return stack

      arithmetic words
            inc         ( a -- a+1 ) increments tos
```

```
        dec             ( a -- a-1 ) decrements tos
        add             ( a b -- a+b ) adds tos and nos
        sub             ( a b -- a-b ) decrements nos from tos
        mul             ( a b -- a*b ) multiplies nos and tos
        divmod          ( a b -- a%b a/b ) divides tos by nos

    bit words
        invert          ( a -- ~a ) bit-wise inversion of tos
        and             ( a b -- a&b ) bit-wise and of tos and nos
        or              ( a b -- a|b ) bit-wise or of tos and nos
        xor             ( a b -- a^b ) bit-wise exclusive or tos and nos
        lshift          ( a b -- a<<b ) left shifts tos by nos bits
        rshift          ( a b -- a>b ) right shifts tos by nos bits

    control flow
        jump            ( -- ) jumps to relative offset at pc
        branch          ( cond -- ) branch to relative offset if cond holds
        not-branch      ( cond -- ) branch to relative offset unless cond holds

    comparison words
        not             ( a -- !a ) logic negation
        is-equal        ( a b -- flag ) pushes 1 if tos equals nos
        is-lesser       ( a b -- flag ) pushes 1 if nos is lesser than nos
        is-elesser      ( a b -- flag ) pushes 1 if nos equals or is
                        lesser than tos

    memory words
        fetch           ( addr -- value ) fetches cell value from addr
        store           ( value addr -- ) stores cell value to addr
        fetch-from      ( i addr -- value ) fetches value from cell array
        store-to        ( i value -- addr ) stores value to cell array
        fetch-byte      ( addr -- value ) fetches byte value from addr
        store-byte      ( value addr -- ) stores byte value to addr
        copy            ( src dst size -- ) copies source to destination

    constants
        cell            ( -- cell-size ) pushes cell size on stack

    variables
        state           ( -- state ) pushes state variable on stack
        latest          ( -- latest ) pushes latest variable on stack
        here            ( -- here ) pushes here variable on stack
        buffer          ( -- buffer ) pushes buffer variable on stack

    interpret words
        enter           ( -- r:pc ) enters subroutine
        leave           ( r:pc -- ) exits subroutine
        exit            ( r:pc -- ) user exit subroutine
        key             ( -- char ) reads one character
        emit            ( char -- ) outputs one character

    defining words
        :               ( -- ) reads input and creates dictionary word entry,
                        immediate

    others
        compare         ( srca lena srcb lenb -- flag ) compares memory blocks
```

```
        is-blank      ( char -- flag ) test if char is blank character
        is-number     ( str len -- flag ) tests if string is number
        word          ( -- str len ) reads one word from input
        number        ( str len -- num ) converts string to number
        create        ( str len -- ) creates dictionary entry
        interpret     ( r:... -- ) outer interpret
        reset         ( r:... -- ) outer interpret loop, never ends
        hidden        ( -- ) toggles hidden flag of the latest word
        [             ( -- ) starts compiling
        ]             ( -- ) starts executing, immediate
        ,             ( value -- ) stores cell at free memory and increment
                       here variable
        ;             ( -- ) compiles end of the latest word
        immediate     ( -- ) sets immediate flag of the latest word, immediate
        execute       ( token -- ) executes word token
        data-stack    ( -- dsize ) pushes data stack size
        return-stack  ( -- rsize ) pushes return stack size
        bye           ( -- ) exits interpret

word definition
        : name body ;

extensions.forth
----------------

conditions
        condition if true-branch else false-branch then

loops
        begin loop-body again
        begin loop-body condition until
        begin condition while loop-body repeat

variables
        var name            # declaration
        name fetch          # reads a variable
        value name store    # writes to a variable

constants
        value const constant-name  # definition
        constant-name              # pushes constant on the stack

anonymous words
        :: body ;

recursion
        : name
                first-part
                recurse
                other-parts ;

switch structure
        switch
                valueA case bodyA else
                valueB case bodyB else
                        bodyC
        endswitch
```

```
      memory allocation
             20 cells alloc             # allocates 20 cells
             forget name                # frees all memory up to
                                        # and including name

      print string
             ' this is string' println # note space after the first
                                        # quotation mark and instead
                                        # of single quotation marks
                                        # use double ones
      print top of the stack
             .

      print value at memory
             address ?

      print stack
             stack

      list of all words
             word-list

      list of words starting with letters
             words abc

      disassemble word
             see word-name

" print ;
```

Obsah CD

diplomová práce: *cd/*

- *dip_prochazka.pdf* – diplomová práce v pdf formátu
- *dip_prochazka.odt* – zdrojový soubor formátu Open Document
- *dip_prochazka_schemata.odg* – zdrojový soubor se schématy ve formátu Open Document Graphics
- *dip_prochazka_tabulky.ods* – zdrojový soubor s tabulkami ve fromátu Open Document Sheet

interpret lit: *cd/soubory/lit*

- *lit.c* – zdrojový kód interpretu
- *litwords.txt* – seznam slov interpretu pro vygenerovani do litram.h
- *litgen.py* – skript generující obsah ram iterpretu do litram.h
- *litram.h* – vygenerovany obsah ram interpetu
- *makefile* – soubor pro program make
- *lit.exe* – předkompilovaný interpret pro windwos
- *lit.cmd* – spouštěcí soubor interpretu s extensions.forth a help.forth
- *poznamky.txt* – stručný popis používání interpretu
- **.forth* – zdrojové soubory aplikací psaných v jazyce interpretu lit

soubory sloužící ke generování bytekódů na různých virtuálních strojích

- *jvm/** – zdrojové soubory JVM
- *cil/** – zdrojové soubory pro platformu .NET a CIL
- *llvm/** – zdrojové soubory LLVM