VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

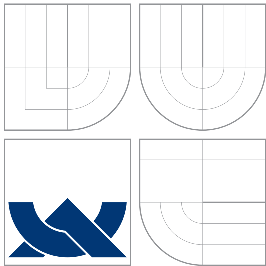
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

INTERPRETER BYTOVÉHO KÓDU JAVA

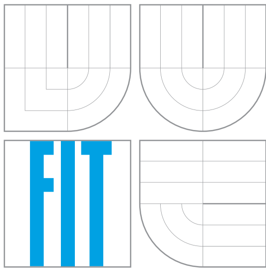
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE JIŘÍ HUSÁK
AUTHOR

BRNO 2013



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

INTERPRETER BYTOVÉHO KÓDU JAVA

JAVA BYTE-CODE INTERPRETER FOR FITKIT PLATFORM

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — JIŘÍ HUSÁK
AUTHOR

VEDOUCÍ PRÁCE — Doc. Dr. Ing. OTTO FUČÍK
SUPERVISOR

BRNO 2013

Abstrakt
Cílem bakalářské práce je navrhnout a implementovat interpreter bytového kódu Java na platformě FITkit. V práci je nejprve rozebrána problematika jazyka Java, zejména vlastnosti přenositelného bytekódu a virtuálního stroje Javy. Dále je v práci popsán mikroprocesor MSP430 od firmy Texas Instruments. Výsledkem práce je interpreter napsaný v jazyce C pro mikroprocesor a aplikace pro PC, která zajišťuje překlad a zavedení bytekódu přes sériovou linku do zařízení FITkit. Na konci práce jsou uvedeny demonstrační aplikace napsané v jazyce Java, které využívají periferie FITkitu nebo také FPGA k akceleraci výpočtů.

Abstract
The aim of this bachelor's thesis is to design and implement the Java bytecode interpreter for FITkit platform. At first is analyzed issues of Java programming language, especially properties of portable Byte-Code and Java Virtual Machine. The study also describes the MSP430 microcontroller from Texas Instruments. The result of bachelor's thesis is the interpreter written in C for microprocessor and application for PC that provides compilation and loading Byte-Code with serial port to the device FITkit. At the end of the work are presented some demonstration applications written in Java that use FITkit peripherals or FPGA to accelerate calculations.

Klíčová slova
Java, bytekód, FITkit, interpereter

Keywords
Java, Byte-Code, FITkit Platform, Interpreter

Citace
Jiří Husák: Interpreter bytového kódu Java, bakalářská práce, Brno, FIT VUT v Brně, 2013

Interpreter bytového kódu Java

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Otto Fučíka. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

....
Jiří Husák
10. května 2013

Poděkování

Rád bych poděkoval svému vedoucímu práce doc. Otto Fučíkovi za směrování správným směrem při tvorbě bakalářské práce.

© Jiří Husák, 2013.
Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Obsah

1 Úvod **3**

2 Jazyk Java u vestavěných zařízení **4**
- 2.1 Jazyk Java . . . 4
  - 2.1.1 Bytekód . . . 4
  - 2.1.2 Java Virtual Machine . . . 4
  - 2.1.3 Datové typy . . . 5
  - 2.1.4 Datové oblasti . . . 5
  - 2.1.5 Rámce (Frames) . . . 7
  - 2.1.6 Překlad pro JVM . . . 8
  - 2.1.7 Instrukční sada JVM . . . 8
- 2.2 Implementace virtuálního stroje Javy . . . 14
  - 2.2.1 Přímá interpretace a JIT compilation . . . 14
  - 2.2.2 HotSpot . . . 14
  - 2.2.3 JamVM . . . 14
  - 2.2.4 Ostatní implementace . . . 14
- 2.3 Platforma FITkit . . . 15
  - 2.3.1 Mikrokontroléry řady MSP430 . . . 15

3 Návrh interpreteru bytového kódu Javy **17**
- 3.1 Požadavky . . . 17
- 3.2 Aplikace pro PC – `ByteCode Loader` . . . 17
- 3.3 Aplikace pro FITkit – `FITkit Virtual Machine` . . . 18
- 3.4 Protokol komunikace mezi PC a FITkitem . . . 18

4 Implementace interpreteru bytového kódu Javy **20**
- 4.1 `ByteCode Loader` . . . 20
  - 4.1.1 Knihovna pro sériový port . . . 20
  - 4.1.2 Překlad Java programů . . . 21
  - 4.1.3 Úprava bytekódu pro FITkit . . . 21
- 4.2 `FITkit Virtual Machine` . . . 22
  - 4.2.1 Datové struktury Bytekódu . . . 23
  - 4.2.2 Datové struktury pro běh JVM . . . 25
- 4.3 Významné funkce interpreteru . . . 27
- 4.4 Rozšíření o hardwarovou akceleraci v FPGA . . . 28
- 4.5 Vývoj aplikací pro Interpreter na FITkitu . . . 29
  - 4.5.1 Nativní funkce - třída FITkit . . . 29
  - 4.5.2 Výpisy ladících informací . . . 29

5 Zhodnocení výsledků **31**
5.1 Demo aplikace . . . 31
5.1.1 Kalkulačka . . . 31
5.1.2 Přístupový terminál . . . 31
5.1.3 Regulátor teploty . . . 32
5.1.4 Sčítání s akcelerací v FPGA . . . 32
5.2 Dosažené výsledky . . . 33

6 Závěr **34**

A Obsah DVD **36**

B Instalace softwaru **37**
B.1 Instalace `ByteCode Loader` . . . 37
B.1.1 Instalace knihovny sérového portu RxTx . . . 37
B.1.2 Instalace JDK . . . 37

Kapitola 1

Úvod

Při tvorbě aplikací je pro vývojáře důležité, jaký programovací jazyk použijí. Pokud máme možnost si vybrat jazyk, potom záleží, jaké máme zkušenosti s daným jazykem, jaká existuje dokumentace nebo jaká je přenositelnost jazyka mezi různými platformami. Právě přenositelnost je jedna z významných vlastností interpretovaných jazyků. Výhodou je, že můžeme vyvíjet aplikace, které jsou nezávislé na cílové architektuře počítače. Tyto jazyky však potřebují ke svému spuštění interpreter, který zajišťuje mezivrstvu mezi architekturou stroje a interpretovaným jazykem.

V této práci byl vybrán jazyk Java, který je zástupcem interpretovaných jazyků na úrovni bytekódu. Java je velmi rozšířená na desktopových, serverových i vestavěných systémech. Pro tento jazyk existuje také více implementací virtuálních strojů pro různé platformy. Cílem této bakalářské práce je však navrhnout a implementovat interpreter na platformě FITkit. Jedná se o vestavěné zařízení s mikroprocesorem řady MSP430 od firmy Texas Instruments a s FPGA Spartan 3 od firmy Xilinx.

Výhodou takového přístupu je, že můžeme program napsaný v jazyce Java vyzkoušet a odladit na počítači. A potom pouze na úrovni přeloženého bytekódu poslat do vestavěného zařízení. Proto byl v rámci bakalářské práce implementován program pro PC, který zajistí překlad a nahrání bytekódu do FITkitu a dále interpreter, zajišťující chod programu na platformě FITkit. Součástí interpreteru je rovněž sada nativních metod, které obsluhují periferie jako je klávesnice, display nebo teploměr.

Druhá kapitola (2) bakalářské práce se věnuje teorii jazyka Java a platformy FITkit. Je zde popsán princip fungování virtuálního stroje Javy, instrukce z instrukční sady nebo známé implementace virtuálních strojů. Rovněž zde jsou popsány parametry mikroprocesoru MSP430. V Třetí kapitole (3) je popsán návrh aplikace. Popisuji zde požadavky na funkčnost, způsob překladu programů, jejich nahrání do vestavěného zařízení a protokol komunikace mezi PC a zařízením FITkit.

Čtvrtá kapitola (4) obsahuje popis implementace, jednotlivých datových struktur a funkcí interpreteru i způsob implementace aplikace pro PC. V předposlední páté kapitole (5) jsou zhodnoceny výsledky na demonstračních aplikacích. Poslední závěrečná kapitola (6) informuje o možnostech pokračování v rámci diplomové práce a předkládá další možné rozšíření aplikace.

Kapitola 2

Jazyk Java u vestavěných zařízení

V této kapitole jsou uvedeny teoretické informace o problematice jazyka Java, zejména o virtuálním stroji Javy (JVM) a o přenositelném bytekódu. Jsou zde uvedeny také některé současné implementace JVM. Poslední část kapitoly se věnuje platformě FITkit. Informace o JVM v této kapitole jsem čerpal z těchto zdrojů [1, 2, 5].

2.1 Jazyk Java

Java je programovací jazyk, který byl uveden firmou Sun Microsystems v roce 1995. Mezi základní vlastnosti jazyka patří, že je objektově orientovaný, interpretovaný, používá správce paměti (Garbage collector), multiplatformní a syntaxí podobný jazyku C.

2.1.1 Bytekód

Bytekód je instrukční sada, která je navržená, aby byla snadno přenositelná na různé platformy [6]. Při spuštění na cílovém počítači je potřeba běhové prostředí, které tvoří mezivrstvu mezi hardwarem a bytekódem. U jazyka Java se běhové prostředí nazývá Java Virtual Machine (JVM). Některé implementace virtuálních strojů, například u jazyka Java nebo Python, nejprve přeloží zdrojový kód do bytekódu, který je přenositelný a při spuštění programu se bytekód překládá do nativního kódu procesoru pomocí JIT 2.2.1. Sice je zde určitá časová prodleva při startu programu, ale běh programu je poté rychlejší.

Struktura bytekódu je následující: číslo instrukce v dané metodě, operační kód instrukce a poté několik volitelných parametrů, podle typu instrukce. Parametry jsou rovněž čísla - buď konstanty nebo odkazy. Proto je zpracovávání bytekódu rychlejší než při interpretování zdrojových kódů.

2.1.2 Java Virtual Machine

JVM je nejdůležitějším prvkem platformy Java. Je to vrstva, která zajišťuje nezávislost bytekódu na hardware či operačním systému. Jde o virtuální stroj, který má rovněž jako reálný stroj svoji instrukční sadu a paměť potřebnou pro běh. V současné době existuje JVM pro mobilní zařízení, desktopové i serverové počítače i pro různá vestavěná zařízení 2.2. Existuje mnoho implementací JVM pro různé platformy. Jak je daný virtuální stroj implementován, zda je použit správce paměti nebo jaké algoritmy jsou použity, záleží na autorovi. Důležité je dodržet specifikaci, správně číst a vykonávat instrukce bytekódu.

2.1.3 Datové typy

Jazyk Java a JVM pracuje se dvěma druhy datových typů: *Primitivní typy* a *Referenční typy*. Typová kontrola probíhá před samotným během programu, typicky při překladu. JVM již neprovádí typovou kontrolu. Mezi *Primitivní typy* patří `int, long, float` a `double` a při běhu programu již není možné zjistit, o jaký datový typ se jedná, proto pro každý typ existuje instrukce v instrukční sadě. Například můžeme najít instrukce pro sčítání v těchto tvarech pro *primitivní datové typy*: *iadd, ladd, fadd* a *dadd*.

Referenční typy jsou buď dynamicky alokovány jako instance třídy nebo jako pole. Odkaz na objekt vyžaduje typ `reference`. Hodnota tohoto typu je ukazatel na daný objekt.

Primitivní datové typy

Datový typ	**rozsah hodnot**	**počáteční hodnota**
boolean	true, false	false
byte	znaménkový 8 bitů	0
short	znaménkový 16 bitů	0
int	znaménkový 32 bitů	0
long	znaménkový 64 bitů	0
char	kódován UTF-16	`'\u0000'`
float	IEEE 754 32 bitů	0
double	IEEE 754 64 bitů	0
returnAdress	ukazatel na kód instrukce	

Tabulka 2.1: Přehled primitivních datových typů.

Datový typ `boolean` je mapován překladačem na datový typ `int`.

Referenční datové typy

U těchto datových typů existují tři druhy: třídy, pole a rozhraní. Jejich hodnota odkazuje na dynamicky vytvořené instance těchto datových typů. `Reference` může nabývat také hodnoty `null`, což znamená, že neodkazuje na žádný objekt. Je to rovněž inicializační hodnota těchto proměnných.

2.1.4 Datové oblasti

Java Virtual Machine obsahuje několik typů datových oblastí. Některé oblasti trvají po celou dobu běhu virtuálního stroje, některé mohou vznikat a zanikat v průběhu podle aktuálně spuštěných vláken.

Registr PC

Program counter (PC) registr vzniká právě jednou pro každé vlákno. Pokud se ve vlákně nevykonává nativní metoda, tento registr obsahuje adresu následující instrukce. Pokud je volaná nativní metoda, je tento registr nedefinovaný.

Java Virtual Machine Stacks

Pro každé vlákno aplikace je vytvořen tento zásobník. Vzniká zároveň s novým vláknem. Jeho význam je podobný jako v ostatních jazycích: uchovává lokální proměnné, dílčí výsledky, parametry metod a návratové hodnoty.

Native Method Stacks

Rovněž tato paměť je vytvářena zvlášť pro každé vlákno. Je určena pro volání nativních metod a předávání parametrů. Nativní metody mohou být napsány i v jiném programovacím jazyce než v Javě.

Halda (Heap)

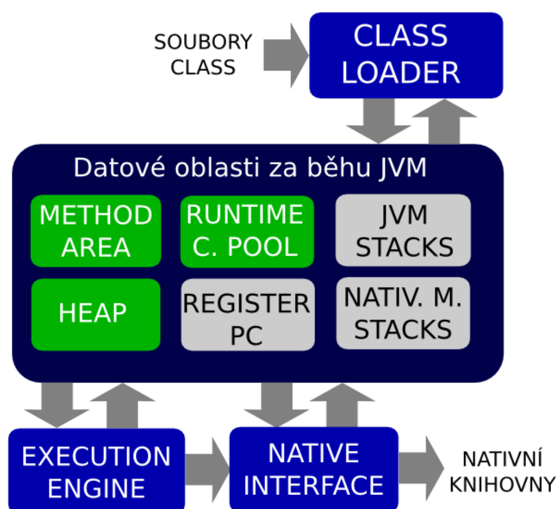
Jedná se o sdílený paměťový prostor, ve kterém jsou uloženy alokované instance tříd a polí. Tato paměť je vytvořena při spuštění programu a je spravována pomocí Garbage collectoru. Alokované objekty nemůžeme dealokovat nikdy explicitně z programu, ale mažou se automaticky správcem paměti.

Method Area

Opět se jedná o sdílenou paměť mezi vlákny. V této oblasti je uložen kód metod, Runtime Constant Pool, pole a kód konstruktorů. Tato oblast je analogií k *storage area* u kompilovaných jazyků.

Runtime Constant Pool

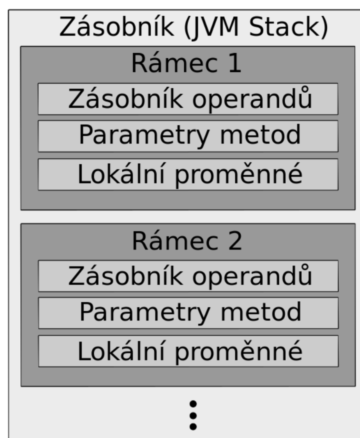
Runtime Constant Pool obsahuje konstanty, jako jsou odkazy na řetězce, jména tříd, rozhraní a polí. Má analogickou funkci jako tabulka symbolů v konvenčních programovacích jazycích.



Obrázek 2.1: Schéma datových oblastí v JVM. Zelené oblasti jsou sdílené, šedé jsou zvlášť pro každé vlákno aplikace.

2.1.5 Rámce (Frames)

Při každém volání metody vzniká nový rámec a zaniká při jejím dokončení. Rámec je alokovaný v oblasti na zásobníku (Java Virtual Machine Stack). Jsou zde umístěny lokální proměnné dané metody, parametry, návratová hodnota a zásobník operandů. Instrukce bytekódu (například aritmetické instrukce) pracují právě se zásobníkem operandů. Příklad stavu zásobníku při volání dvou metod můžeme vidět na obrázku 2.2.



Obrázek 2.2: Struktura zásobníku při volání metod.

2.1.6 Překlad pro JVM

Software Java Development Kit [8] (JDK) od firmy Oracle je sada programů, určená pro vývojáře v jazyce Java. JDK obsahuje překladač, zavaděč programů, degugger, program pro generování dokumentací, program pro tvorbu archivů `jar` a také program pro generování bytekódu v textové podobě ze zdrojových kódů.

Javap

`Javap` je program, který vypisuje zvolené informace o souborech obsahující třídu. Tyto soubory mají příponu `.class`. V parametrech programu můžeme zvolit, zda chceme vypsat informace o metodách `public, protected` a `private` nebo lze zvolit parametr `-c`, který provede zpětný překlad metod v souboru `class` a vypíše je po instrukcích bytekódu.

```
public static int main(String[] args) {
  int a = 1;
  int b = 2;
  int c = a + b;
  return c;
}
```

Tento vzorový úsek programu v jazyce Java se přeloží do následujících instrukcí bytekódu. Zde je ukázka výstupu programu `javap`:

```
public static int main(java.lang.String[]);
   Code:
     0: iconst_1
     1: istore_1
     2: iconst_2
     3: istore_2
     4: iload_1
     5: iload_2
     6: iadd
     7: istore_3
     8: iload_3
     9: ireturn
```

2.1.7 Instrukční sada JVM

Každá instrukce z instrukční sady má operační kód dlouhý 8 bitů, z toho vyplývá, že instrukční sada může obsahovat až 256 instrukcí, ovšem ne všechny jsou používané. Dále za operačním kódem mohou být až další 2 byty, které představují buď konstantu nebo index do Constant Poolu.

Při volání metody se vytvoří nový rámec na zásobníku, který obsahuje zásobník operandů. Instrukce poté pracují s tímto zásobníkem operandů. Instrukce můžeme rozdělit do několika kategorií: Instrukce pro ukládání a načítání ze zásobníku, aritmetické a logické instrukce, instrukce na konverzi typů, pro vytváření a manipulaci s objekty, pro řízení toku programu a další.

Mnoho instrukcí pracuje pouze s určitým datovým typem. Zpravidla se jedná o `int, long, float` a `double`. První písmeno v názvu instrukce určuje, se kterým typem se pracuje. Kompletní instrukční sadu si můžete prohlédnout zde [7].

Instrukce pro ukládání konstanty na zásobník operandů

Tyto instrukce slouží k uložení konstanty na zásobník operandů. Konstanta je buď dána přímo instrukcí nebo se jedná o index do Constant Poolu, kde je uložena daná konstanta. V Constant Poolu může být uložená daná proměnná nebo se může jednat o referenci, například u řetězců. Nejpoužívanější konstanty jsou dány přímo instrukcí. Například u datového typu `int` je celkem 7 instrukcí pro uložení konstant (-1 až +5). Celkem je těchto instrukcí 20, zde je přehled některých z této skupiny.

Instrukce	**Op. kód**	**Data 1**	**Data 2**	**Datový typ**	**Popis**
iconst_m1	0x02			int	uloží konstantu -1
iconst0	0x03			int	uloží konst. 0
lconst0	0x09			long	uloží konst. 0L
fconst1	0x0c			float	uloží konst. 0.0F
dconst0	0x0e			double	uloží konst. 0.0
ldc	0x12	index		string, int, ref, float	načte konst. z Constant Poolu

Tabulka 2.2: Příklad instrukcí pro ukládání konstant na TOS.

Instrukce pro přesun dat mezi lokálními proměnnými a zásobníkem operandů

Tato skupina instrukcí umožňuje přesun dat mezi lokálními proměnnými nebo parametry metody a zásobníkem operandů. V této rozsáhlé skupině je 50 instrukcí. Lokální proměnné a parametry jsou určeny indexem a u zásobníku operandů se pracuje jen s poslední proměnnou na vrcholu zásobníku (TOS), k ostatním se dostat nemůžeme. Všechny instrukce se jmenují buď load nebo store. Před tímto názvem je ještě jedno písmenko označující, s jakým datovým typem se bude pracovat a za názvem může být číslo, které určuje index v lokálních proměnných nebo parametrech metody. Instrukce load vždy ukládá cíl na vrchol zásobníku a instrukce store bere za zdroj vždy rovněž vrchol zásobníku.

Instrukce pro přímou manipulaci s prvky na zásobníku operandů

Tato skupina obsahuje celkem 9 instrukcí, které mohou přímo manipulovat se zásobníkem operandů. Jsou vhodné, když například chceme odstranit položku z vrcholu zásobníku nebo ji naopak zduplikovat, či prohodit první dva prvky na zásobníku. U těchto instrukcí se už nerozlišuje, s jakým datovým typem pracují, protože se řídí podle *datového tagu*, který je uložen zároveň s daty na zásobníku. V tabulce jsou příklady některých instrukcí.

Instrukce	Op. kód	Data 1	Datový typ	Popis
iload	0x15	index	int	načte lokální prom. z pozice index, uloží na TOS
aload	0x19	index	reference	načte lokální prom. typu reference z pozice index, uloží na TOS
iload0	0xA1		int	načte lokální prom. typu int z pozice 0, uloží na TOS
istore	0x36	index	int	přesune hodnoty typu int z TOS do lokál. prom. na pozici index
dstore3	0x4A		double	přesune hodnoty typu double z TOS do lokál. prom. na pozici 3

Tabulka 2.3: Příklad instrukcí pro přesun dat mezi TOS a lokální proměnnou.

Instrukce	Op. kód	Popis
pop	0x57	odstraní položku z TOS (int, float, reference na objekt)
pop2	0x58	odstraní 1 položky z TOS (long, double) nebo 2 položky (int,float,reference na objekt)
dup	0x59	zduplikuje položku z TOS (int, float, reference na objekt)
swap	0x5F	prohodí 2 prvky na TOS (int, float, reference na objekt)

Tabulka 2.4: Příklad instrukcí pro manipulaci s prvky zásobníku.

Instrukce pro konverzi mezi datovými typy

V této skupině se nachází 15 instrukcí, provádějících konverzi mezi datovými typy. V jazyce Java jsou některé konverze typů prováděny automaticky, například `byte` se změní na `int`. Obsažené instrukce nestačí na konverzi mezi všemi datovými typy navzájem, proto některé konverze jsou prováděny pomocí dvou instrukcí bytekódu.

Instrukce	Op. kód	Popis
i2c	0x92	int převede na char
f2i	0x8b	float převede na int
l2d	0x8a	long převede na double

Tabulka 2.5: Příklad instrukcí pro konverzi datových typů.

Instrukce pro aritmetické a logické operace

Počet Instrukcí, které počítají aritmetické operace je celkem 28 a pracují většinou se dvěma operandy uloženými na vrcholu zásobníku. Po provedení dané operace jsou oba původní operandy odstraněny a na vrcholu zásobníku se objeví výsledek operace. Instrukce provádí tyto operace: sčítání, odčítání, násobení, dělení a výpočet zbytku po dělení. Každá operace existuje ve 4 variantách pro různé datové typy: `int, long, float` a `double`. Ostatní datové typy se převádí na ty výše zmíněné, například `byte, short` či `char` se převádí na `int`. V této skupině se také nacházejí instrukce pro změnu znaménka, které použijí prvek na vrcholu zásobníku, poté ho znegují, původní hodnota je odstraněna a nová se objeví na

vrcholu zásobníku. Najdeme zde také instrukce pro inkrementování proměnných, které se často vyskytují například v cyklech.

JVM obsahuje i instrukce pro bitové operace. Lze je provádět pouze s datovými typy `int` a `long`. Jsou to operace AND, OR a XOR. Ostatní bitové operace se provádějí pomocí těchto tří operací. Dále existují také instrukce pro bitové posuvy doleva a doprava.

Instrukce	**Op. kód**	**Data 1**	**Data 2**	**Popis**
iadd	0x60	int	int	sčítání
lsub	0x65	long	long	odečítání
fmul	0x6A	float	float	násobení
ddiv	0x6F	double	double	dělení
irem	0x70	int	int	zbytek po dělení
ineg	0x74	int		negace
iinc	0x84	8bit index	8bit konst	přičtení 8bit konstanty
ishl	0x78	int	int	aritm. bit. posun doleva
iand	0x7E	int	int	bitový součin
ior	0x80	int	int	bitový součet
ixor	0x82	int	int	nonekvivalence

Tabulka 2.6: Příklad instrukcí pro aritmetické a logické operace.

Instrukce pro řízení toku programu

Instrukce z této skupiny jsou potřeba, abychom mohli porovnat určité hodnoty a na základě výsledku řídit program. Patří sem tedy instrukce pro porovnávání dvou hodnot, instrukce podmíněného a nepodmíněného skoku a instrukce pro ukončení metody a předání návratové hodnoty.

Při operaci porovnávání hodnot pracujeme s datovými typy `long, float` a `double` a výsledek operace je typu `int` a zůstane na vrcholu zásobníku. Výsledek 1 je, když operand 1 je větší, než operand 2. Nulový výsledek je, když se operandy rovnají a výsledek -1, když operand 1 je menší, než operand 2.

Ukončení metody a předání parametrů provádí různé tvary instrukce return. Při volání této instrukce se zruší celý rámec dané metody a návratová hodnota se objeví na vrcholu zásobníku metody, která končící metodu volala.

Další podskupinou jsou instrukce pro podmíněné a nepodmíněné skoky. Nepodmíněný skok provádí instrukce `goto`. Zde platí omezení, že lze skočit pouze v rámci metody, ne napříč celým programem.

U podmíněných skoků nás zajímá výsledek porovnávání, který zůstal na vrcholu zásobníku operandů. Výsledek je datového typu `int` a odstraní se automaticky po provedení testu. Při splnění podmínky se skáče na adresu danou dvěma byty operandů instrukce, kde výsledná adresa skoku se spočítá jako 256 * highbyte + lowbyte. Existuje 6 instrukcí, které testují, zda prvek na vrcholu zásobníku operandů je roven nule, není roven nule, je menší, větší, menší nebo rovno a větší nebo rovno nule. Existují také dvě instrukce, provádějící skok podle hodnoty reference. Rozhodují se podle toho, zda je na vrcholu zásobníku operandů reference na `null` či nikoli.

Kvůli vyšší efektivitě byly přidány ještě další instrukce, které přímo porovnávají první dva prvky na vrcholu zásobníku a skok na zadanou adresu provedou na základě výsledku

Instrukce	Op. kód	Data 1	Data 2	Popis
lcmp	0x94	long	long	porovnání 2 operandů
ireturn	0xAC			návrat z metody, návratová hodnota typu int
return	0xB1			pouze návrat z metody
goto	0xA7	highbyte	lowbyte	skok na adresu
ifeq	0x99	highbyte	lowbyte	skok na adresu, pokud TOS = 0
iflt	0x9B	highbyte	lowbyte	skok na adresu, pokud TOS < 0
ifnull	0xC6	highbyte	lowbyte	skok na adresu, pokud TOS = null

Tabulka 2.7: Příklad instrukcí pro řízení toku programu.

porovnání. První dva prvky na zásobníku musí být typu `int` nebo `reference`.

V instrukčním souboru také existují instrukce pro řízení běhu programu při použití větvení typu `switch`. Instrukce přináší vyšší efektivitu při běhu programu.

Instrukce	Op. kód	Data 1	Data 2	Popis
if_icmpeq	0x9F	highbyte	lowbyte	skok na adresu, pokud prvek1 = prvek2 na TOS
if_icmplt	0xA1	highbyte	lowbyte	skok na adresu, pokud prvek1 < prvek2 na TOS

Tabulka 2.8: Příklad instrukcí podmíněného skoku.

Instrukce pro manipulaci s objekty

Následující instrukce se používají k načtení nebo uložení atributu v objektu. Existuje varianta instrukce pro statické a nestatické atributy.

Dále existují rovněž instrukce typu `newarray`, které vytvářejí nové pole objektů

Instrukce	Op. kód	Data 1	Data 2	Popis
getfield	0xB4	highbyte	lowbyte	načte hodnotu atributu, jehož reference je na TOS, výslednou hodnotu uloží na TOS
putfield	0xB5	highbyte	lowbyte	hodnotu z TOS uloží do atributu, jehož reference je na 2. místě zásobníku

Tabulka 2.9: Příklad instrukcí pro manipulaci s objekty.

Instrukce pro volání metod

Při volání metod rozlišujeme, zda voláme statickou nebo virtuální metodu. Statickou metodu můžeme volat, i když nemáme vytvořenou instanci objektu. Virtuální metodu naopak musíme volat, pouze pokud je vytvořená instance objektu a jako parametr se předává reference na objekt this. Při volání metod se vytvoří nový rámec na zásobníku pro danou metodu.

Instrukce	Op. kód	Data 1	Data 2	Popis
invokestatic	0xB8	highbyte	lowbyte	voláni statické metody
invokevirtual	0xB6	highbyte	lowbyte	voláni virtuální metody, předá this

Tabulka 2.10: Instrukce pro volání metod.

Instrukce pro manipulaci se statickými proměnnými

Následující dvojice instrukcí ukládá nebo načítá hodnoty ze statických proměnných. Statické proměnné jsou vytvořeny právě jednou v rámci třídy a přistupuje se k nim pouze přes index v Constant Poolu.

Instrukce	Op. kód	Data 1	Data 2	Popis
putstatic	0xB3	highbyte Index	lowbyte Index	načte na TOS hodnotu statické proměnné, na kterou ukazuje položka CP o daném indexu
getstatic	0xB2	highbyte Index	lowbyte Index	uloží hodnotou z TOS do statickké proměnné, na kterou ukazuje položka CP o daném indexu

Tabulka 2.11: Instrukce pro manilupaci se statickými proměnnými.

2.2 Implementace virtuálního stroje Javy

Jelikož je Java jeden z nejpoužívanějších jazyků a také proto, že je multiplatformní, najdeme mnoho implementací virtuálního stroje Javy na nejrůznějších platformách. Proto můžeme vidět programy v Javě na mikrokontrolérech, v mobilních telefonech, tabletech, stolních počítačích i v serverech. Kompletní seznam známých virtuálních strojů najdeme v tomto odkaze [9].

2.2.1 Přímá interpretace a JIT compilation

Přímá interpretace bytekódu nedosahuje takového výkonu, jako kdybychom spouštěli program přeložený do nativního kódu [10]. Proto se v některých implementacích JVM používá překlad za běhu (Just in Time compilation). Tento překlad do nativního kódu překládá za běhu často spouštěné metody, aby se docílilo vyššího výkonu.

2.2.2 HotSpot

HotSpot[1] je referenční implementace virtuálního stroje od firmy Oracle. Tato implementace používá vyspělé technologie, jako je správce paměti, program pro nahrávání tříd, interpretr bytekódu a JIT překladač. Je určena pro počítače s architekturami x86, amd64 a Sparc. Tento virtuální stroj je napsaný v jazyce C++ a může být použit v operačních systémech Microsoft Windows, Linux, Solaris a MAC OS X.

2.2.3 JamVM

JamVM[2] je implementací JVM pro vestavěná zařízení, mobilní telefony a tablety. Jedná se o plnohodnotnou implementaci JVM. Má extrémně malou velikost a na rozdíl od implementace HotSpot lze relativně snadno přidávat podporu pro další architektury. JamVM je napsána v jazyce C, některé úseky kódu jsou optimalizované v assembleru. Nevýhodou může být absence plnohodnotného JIT překladače a dalších pokročilejších funkcí implementace HotSpot.

2.2.4 Ostatní implementace

Z dalších implementací je zajímavá implementace s názvem *JOP*[3], ve které je implementována JVM přímo do hardwaru. Tento procesor je vhodný pro časově náročné real-time aplikace. Implementace má malou velikost, takže lze naprogramovat do FPGA.

Další zajímavostí je *JC virtual machine*[4], která převádí soubory tříd s bytekódem do jazyka C a dále ho překládá pomocí překladače GCC. Takto můžeme dále manipulovat a optimalizovat bytekód.

NanoVM[5] je virtuální stroj určený pro mikrokontroléry AVR. Obsahuje interpret, několik nativních tříd a celková velikost kódu je menší než 8 kB. Také z 1 kB RAM je nejméně 75 % k dispozici běžícímu programu. Dosahuje rychlosti 20 tisíc operací za sekundu při taktu 8 MHz.

[1] *HotSpot* http://openjdk.java.net/groups/hotspot/
[2] *JamVM* http://jamvm.sourceforge.net/
[3] *JOP* http://www.jopdesign.com/
[4] *JC VM* http://jcvm.sourceforge.net/
[5] *NanoVM* http://www.harbaum.org/till/nanovm/index.shtml

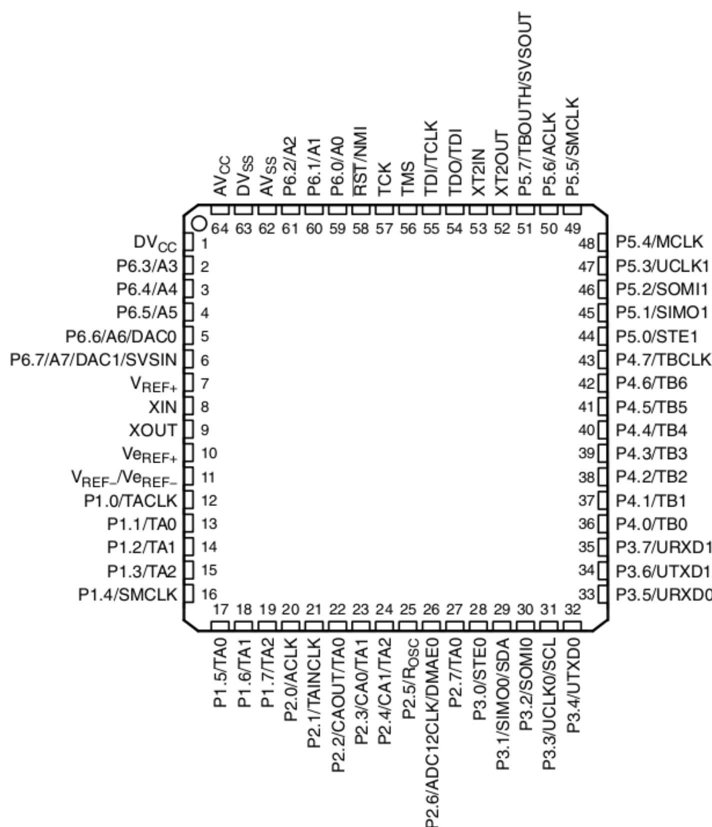
2.3 Platforma FITkit

Výsledná aplikace je určena pro platformu FITkit. FITkit vznikl na Fakultě informačních technologií na VUT v Brně, jako výuková platforma pro studenty, aby mohli prakticky navrhovat softwarové i hardwarové aplikace. FITkit je hardware, který obsahuje mikrokontrolér z řady MSP430 od firmy Texas Instruments a také programovatelné hradlové pole FPGA Spartan 3 od firmy Xilinx.

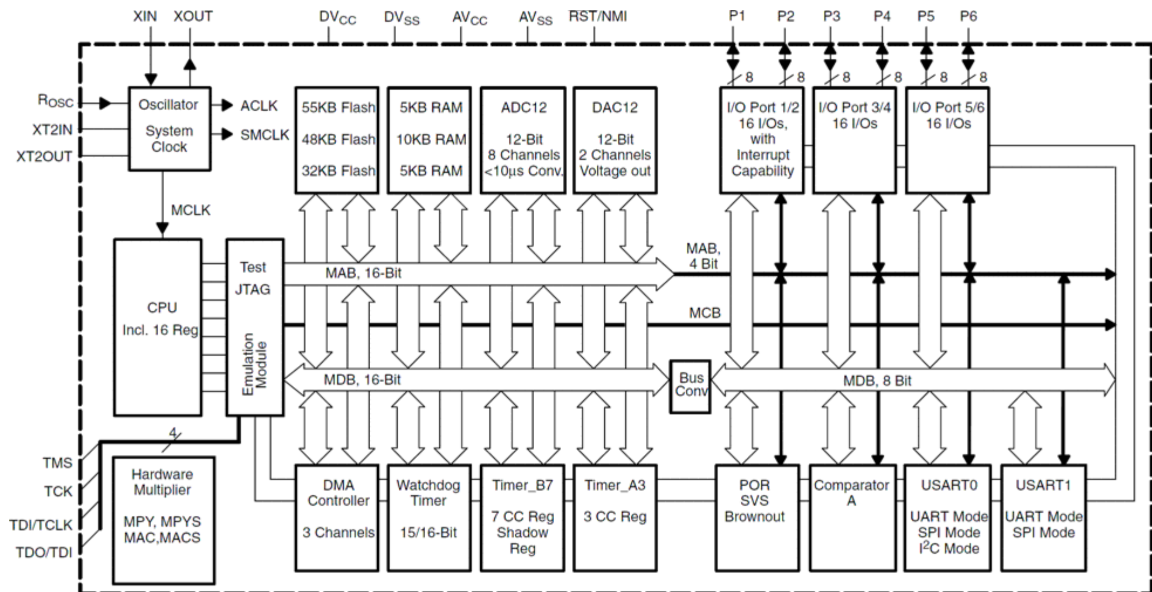
FITkit také obsahuje DRAM s 8 MB paměti, dvouřádkový LCD display, audio vstup a výstup, rozhraní sériové linky RS232, rozhraní pro PS2 (myš a klávesnice), maticovou klávesnici 4x4, konektor pro VGA a další programovatelné konektory připojené k MCU nebo FPGA [4].

2.3.1 Mikrokontroléry řady MSP430

Mikrokontrolér MSP430 je navržen jako nízkopříkonový, v aktivním módu je příkon 330 μA při 1 MHz a 2.2 V napájecího napětí, 1.1 μA ve stand-by režimu a 0.2 μA v režimu vypnuto. Mikrokontrolér může být napájen v rozmezí 1.8 V až 3.6 V. MCU má 16-ti bitovou RISC architekturu a instrukční cyklus je 125 ns. Dále obsahuje 12-ti bitové A/D převodníky, 16-ti bitové čítače, sériové asynchronní rozhraní UART a synchronní SPI nebo I^2C rozhraní. MCU MSP430F168 obsažený ve FITkitu 2.0 obsahuje 92 kB paměti typu FLASH a 8 kB paměti RAM. Celkem můžeme libovolně programovat 6 portů, kde každý obsahuje 8 pinů [11].



Obrázek 2.3: Mikrokontrolér MSP430F168 [11]



Obrázek 2.4: Diagram funkčních bloků mikrokontroléru MSP430F168 [11]

Mikrokontrolér MSP430F168 je možno programovat v jazyku symbolických instrukcí nebo v jazyce C. Překlad z jazyka C zajišťuje překladač mspgcc [3]. Architektura mikrokontroléru má společnou paměť pro data a kód. Procesor má 16 bitovou architekturu, ale umožňuje číst a zapisovat do každé 8-bitové adresy. Celočíselné typy mohou být buď znaménkové nebo neznaménkové. Ukazatel zabírá vždy 16 bitů. Zde tabulka základních datových typů:

char	8 bitů
int	16 bitů
long	32 bitů
long long	64 bitů
float	32 bitů

Tabulka 2.12: Přehled datových typů v překladačí mspgcc

Překladač mspgcc umožňuje programovat v jazyce C obsluhy přerušení, vkládat úseky assembleru do jazyka C a obsahuje také základní funkce ze standardních knihoven C. Jsou to například funkce pro práci s řetězci, práci s oblastí paměti, pro tisk formátovaného řetězce do pole nebo převody mezi datovými typy.

Kapitola 3

Návrh interpreteru bytového kódu Javy

3.1 Požadavky

Cílem aplikace je spouštět programy napsané v jazyce Java na vestavěném zařízení – platformě FITkit. Samotný překlad ze zdrojových kódů Javy do bytekódu probíhá na PC. Upravený výsledek je poté posílán přes sériovou linku do FITkitu. Zde se přijímaný bytekód ukládá do paměti RAM a po dokončení přenosu se program spustí. Po provedení programu se FITkit znovu vrátí do stavu, kdy lze začít posílat bytekód do zařízení, nicméně většina vestavěných aplikací běží v nekonečné smyčce. V tom případě ukončíme program stisknutím RESET tlačítka na FITkitu. Výsledná aplikace se tedy skládá ze dvou částí:

- Aplikace pro PC, která zajišťuje překlad ze zdrojových textů Javy do bytekódu a následně posílá bytekód přes sériovou linku do FITkitu.
- Aplikace pro FITkit, která obsahuje interpreter bytového kódu, obsluhuje sériovou linku s PC a zajišťuje obsluhu s periferiemi (klávesnice, display, . . .).

3.2 Aplikace pro PC – `ByteCode Loader`

Vstupem aplikace je zdrojový text v jazyce Java, který aplikace přeloží a bytekód odešle do FITkitu. Je podporována pouze jedna třída, která musí obsahovat statickou metodu `main`. Program zajistí překlad do souboru `class`. V počítači proto musí být nainstalovaný software Oracle JDK [8]. Následně je zobrazen v textové podobě bytekód jednotlivých metod, informace o potřebné velikosti zásobníku pro dané metody a také Constant Pool. ByteCode Loader provádí rovněž zpracování bytekódu z textu do jednotlivých sekvencí, které budou přenášeny po sériové lince do FITkitu.

Program funguje jako terminál, který umožňuje komunikovat se zařízeními po sériové lince. Lze nastavit port[1], na kterém je zařízení připojeno a jeho rychlost[2]. Příkazy lze buď posílat jednotlivě jako v běžném terminálu nebo spustit automatizované nahrání bytakódu do FITkitu.

[1]MCU Fitkitu se typicky připojí pod OS Windows na `COM4` a u OS Linux do souboru `/dev/ttyUSB0`.
[2]FITkit komunikuje rychlostí 460800 bd.

3.3 Aplikace pro FITkit – `FITkit Virtual Machine`

Tato část aplikace je určena pro platformu FITkit. Program obsluhuje sériový port, přes který přichází vstupní bytekód z PC. Hlavní částí programu je interpreter bytekódu, který vykonává jednotlivé instrukce z instrukční sady jazyku Java. Jelikož se jedná o vestavěné zařízení, tak nebyly implementovány všechny možnosti jazyku Java jako například Garbage Collector, objektově orientovaný přístup, práce s vyjímkami nebo abstraktní datové typy.

Jazyk Java pracuje se zásobníkovou architekturou, proto instrukce pracují s vrcholem zásobníku (TOS) a nevyužívají například pomocné registry. Kromě funkcí pro uložení na vrchol zásobníku a načtení z vrcholu zásobníku musí interpreter obsahovat rovněž funkce, které ukládají konstantu různých datových typů na vrchol zásobníku nebo obsluhují práci s lokálními proměnnými.

Při příjmu bytekódu po sériové lince se bytekód ukládá do paměti RAM mikrokontroléru. Dynamicky se zde vytváří seznamy, podle typu příchozích zpráv. Existují seznamy s položkami Constant Poolu, kde se nachází konstanty jako jsou řetězce, odkazy na metody nebo ostatní primitivní datové typy. Další je seznam statických proměnných a seznam metod, kde každá položka metody obsahuje seznam instrukcí, které se nacházejí v dané metodě. Speciální metodou je konstruktor třídy `Object`, který vytvoří prvotní rámec a zavolá metodu `main`.

Voláním metody `main` se začíná provádět naprogramovaný kód. Aktuální zpracovávaná instrukce je uchovávaná v `PC registru`, což je datová struktura obsahující operační kód instrukce, operandy, číslo instrukce v dané metodě a odkaz na následující instrukci, na kterou se má přejít, není-li zrovna prováděna instrukce skoku, volání metod a podobně.

Při každém volání metody se vytvoří nový rámec na zásobníku. Rámec se skládá ze dvou částí. Lokální proměnné metody a zásobník metody. Překladač Javy už při překladu určí, jaká je maximální velikost zásobníku a počet proměnných, proto můžeme již při volání metody vytvořit rámec potřebné velikosti. Lokální proměnné jsou dostupné pomocí indexu a můžeme k nim přistupovat kdykoli během kódu metody. U zásobníku metody však můžeme přistupovat pouze k vrcholu zásobníku. Pokud má metoda nějaké parametry, musí být při volání nové metody na vrcholu zásobníku a poté jsou nakopírovány do lokálních proměnných nové metody. Jestli metoda vrací nějakou hodnotu, tak výsledek zůstane na vrcholu zásobníku po ukončení a odstranění rámce ukončené metody.

Pro obsluhu periferií FITkitu jsou k interpereteru připojeny nativní funkce. Když se při interpretaci kódu zjistí, že je volána nativní funkce, nevytváří se nový rámec, ale přímo se spustí daná funkce. Může jít například o čtení z maticové klávesnice, zápis znaku na LCD display, zjištění teploty mikrokontroléru nebo ovládání LED diod.

3.4 Protokol komunikace mezi PC a FITkitem

Aplikace na PC posílá upravený bytekód v řetězcích, aby FITkit mohl jednoduše ukládat data do paměti. Aplikace na PC odešle vždy jeden řádek, čeká na odpověď od FITkitu a poté pokračuje. Odesílaná data můžeme rozdělit do dvou částí. Nejprve se posílají položky Constant Poolu, kde každý řádek odpovídá jedné položce. Poté se posílají statické proměnné a jednotlivé metody. Vždy se pošle nejdříve hlavička metody s názvem a informacemi o potřebném počtu proměnných v rámci a poté se posílají jednotlivé instrukce. Na konci posílaných dat se pošle řádek s ukončovací sekvencí znaků. Zde je jednoduchý příklad krátkého programu:

```
==CP==                  // začátek posílání Constant Poolu
1=8=4=main()V           // 1. položka typu metoda(8)
2=8=3=add(II)I          // 2. položka typu metoda(8)
3=6=example             // 3. položka typu třída(6)
4=6=java/lang/Object    // 4. položka typu třída(6)
==M==                   // začátek posílání kódů metod
=m=1=1="konstruktor"    // inicializační metoda konstruktor
0=42=                   // aload_0
1=183=1                 // invokespecial #1 - volání metody č.1 z CP - main()
4=177=                  // return
=m=2=4=main()V          // metoda main, 2 pol.zásobník, 4 lokální prom.
0=4=                    // iconst_1
1=60=                   // istore_1
2=5=                    // iconst_2
3=61=                   // istore_2
4=27=                   // iload_1
5=28=                   // iload_2
6=184=2                 // invokestatic #2 - volání metody č.2 z CP - add()
9=62=                   // istore_3
10=177=                 // return
=m=2=2=add(II)I         // metoda add, II - vstup int, int, návrat I - int
0=26=                   // iload_0
1=27=                   // iload_1
2=96=                   // iadd
3=172=                  // ireturn
==E==                   // konec posílaných dat
```

Tabulka 3.1: Příklad dat zasílaných do FITkitu.

Položky v Constant Poolu vždy začínají číslem, přes které jsou identifikovány. Za pořadovým číslem následuje typ položky. Celkem existuje 8 typů položek v CP: 1 – String, 2 – Integer, 3 – Float, 4 – Long, 5 – Double, 6 – Class reference, 7 – Field reference, 8 – Method reference. Pokud se jedná o metodu, tak následuje identifikátor třídy, ke které metoda patří. Posledním údajem jsou data. Je to buď řetězec, číslo, nebo reálné číslo, podle typu dané položky.

Každá metoda je nejprve uvozena písmenem `m`. Poté nasledují dva parametry. První určuje maximální počet jednotek na zásobníku, které budou během volání metody potřeba. Druhý parametr je počet lokálních proměnných, včetně vstupních parametrů. Při součtu těchto čísel dostaneme velikost rámce, který bude zabírat daná metoda. Dále následuje řetězec se jménem metody. V závorkách za metodou se nacházejí velká písmena, která představují počáteční písmena vstupních parametrů. V – Void, I – Integer, atd . . . Stejným způsobem je poté označen návratový typ metody uvedený za závorkami.

Po hlavičce metody se posílají jednotlivé instrukce. Na počátku jsou uvozeny opět pořadovým číslem, které identifikuje instrukci v rámci metody. Poté následuje operační kód v dekadickém zápise. Nakonec mohou být připojeny dva byty parametrů. V protokolu jsou však posílány oba byty dohromady a případné bitové rozdělení se provádí až při interpretaci instrukce. Konec zasílaných dat je poté oznámen zasláním řetězce `==E==`.

Kapitola 4

Implementace interpreteru bytového kódu Javy

V této kapitole bude popsána implementace aplikace `FITkit Virtual Machine`. Budou rozebrány a popsány důležité datové struktury a funkce z aplikace. V krátkosti bude také zmíněn princip implementace aplikace `ByteCode Loader`, která zajišťuje překlad a nahrání bytekódu do zařízení FITkit.

4.1 `ByteCode Loader`

Grafická aplikace pro PC je napsána v jazyce Java. Je určena jak pro OS Windows, tak pro OS Linux. Na obrázku 4.1 je vidět, že okno aplikace je rozděleno do dvou částí. Levá část okna zobrazuje zdrojový program v jazyce Java ve třech záložkách. První zobrazuje zdrojový program, druhá záložka bytekód z výstupu aplikace `javap` a třetí záložka obsahuje bytekód zakódovaný v číslech, který se posílá po sériové lince.

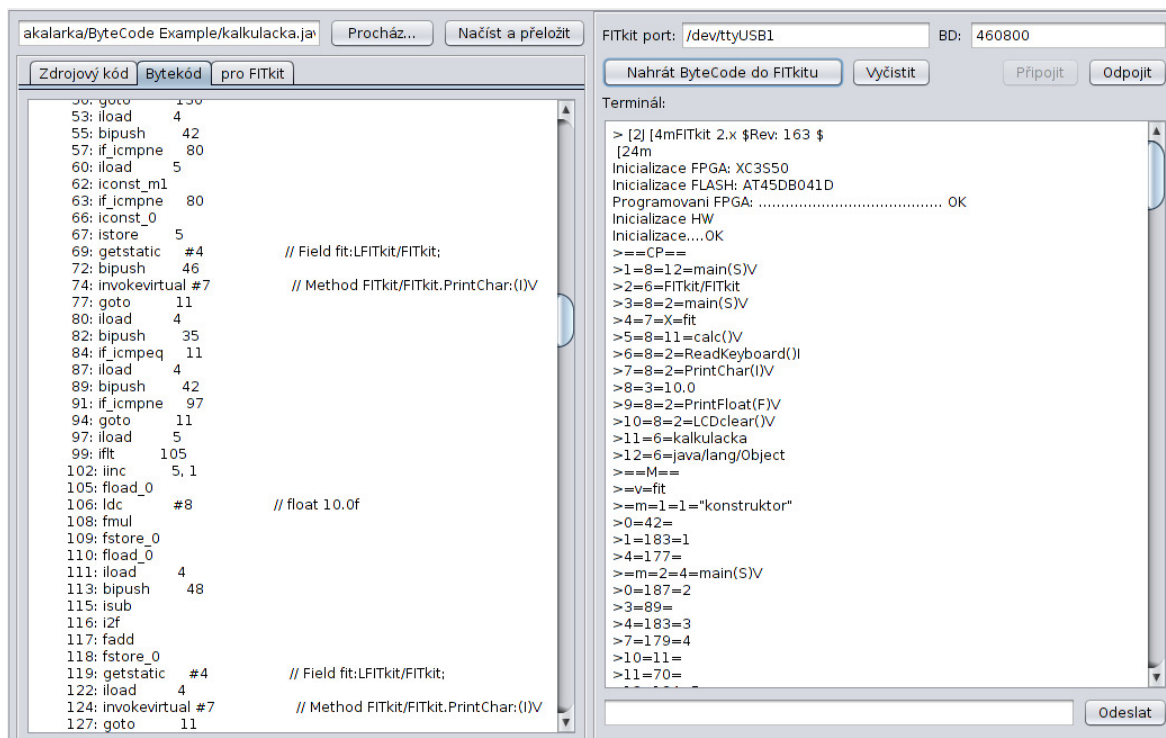
Pravá část okna je určena pro zobrazování informací o spojení sériové linky. Dále je zde textové pole zobrazující komunikaci po sériové lince, podobně jako v terminálu. Jednotlivé příkazy buď můžeme psát ručně do pole a odesílat nebo je možné využít odeslání celého programu v dávce.

4.1.1 Knihovna pro sériový port

Pro obsluhu sériového portu v jazyce Java byla použita knihovna RxTx[1]. Jedná se o multiplatformní a open source knihovnu. Pro spojení se zařízením se zadávají informace o názvu portu, rychlosti, paritě a stop bitech. Pokud je vše správně zadáno, vytvoří se spojení. V jazyce Java se ze sériového portu čte pomocí třídy `InputStream` a zapisuje pomocí třídy `OutputStream`. Když přijdou nějaká data, spustí se událost pro obsluhu přijatých dat. Obsluhu sériové linky zajišťuje třída `SerialAPI`.

Při dávkovém posílání řetězců do zařízení FITkit se vytvoří nové vlákno, které odešle jeden řádek. Vlákno je synchronizováno semaforem, proto poté čeká, dokud nedojde odpověď od FITkitu. Poté se semafor uvolní a opět se může posílat další řádek. Obsluha dávkového zasílání dat je implementována ve třídě `ThreadSerial`. Pro ukončení spojení se zařízením je třeba uzavřít sériové spojení. Bližší informace o instalaci knihovny a webové stránce projektu najdeme v příloze B.1.1.

[1] *RxTx* `http://rxtx.qbang.org`



Obrázek 4.1: Ukázka z aplikace `ByteCode Loader`

4.1.2 Překlad Java programů

Překlad ze zdrojových kódů zajišťuje externí program `javac` z balíku programů JDK. Po úspěšném překladu se vytvoří v adresáři, kde je zdrojový kód, soubor `class`. Pokud nastala chyba při překladu, zobrazí se uživateli okno s výstupem překladače.

Soubor `class` obsahuje bytekód a strukturu třídy v zakódovaném binárním tvaru. Pro lepší následnou úpravu bytekódu je zavolán externí program `javap`. Při použití parametrů `javap -v` (verbose) vypíše veškeré informace o třídě v textové podobě. Tento výstup se objeví v záložce *Bytekód*.

4.1.3 Úprava bytekódu pro FITkit

Důležitou funkcí programu je převést textový výstup programu `javap` do protokolu pro posílání bytekódu do FITkitu. Protokol je popsán v této kapitole 3.4. Třída `ParserByteCode` implementuje pomocí regulárních výrazů úpravu bytekódu.

Program postupně po řádcích prochází vstup a nejprve zjistí název třídy, poté se aplikují regulární výrazy a hledají se výskyty položek Constant Poolu. Dále se hledají výskyty hlaviček metod, informace o potřebné velikosti zásobníku nebo počtu a typu parametrů. Po hlavičce metody se hledají výskyty instrukcí. Například formát vstupní instrukce může vypadat takto:

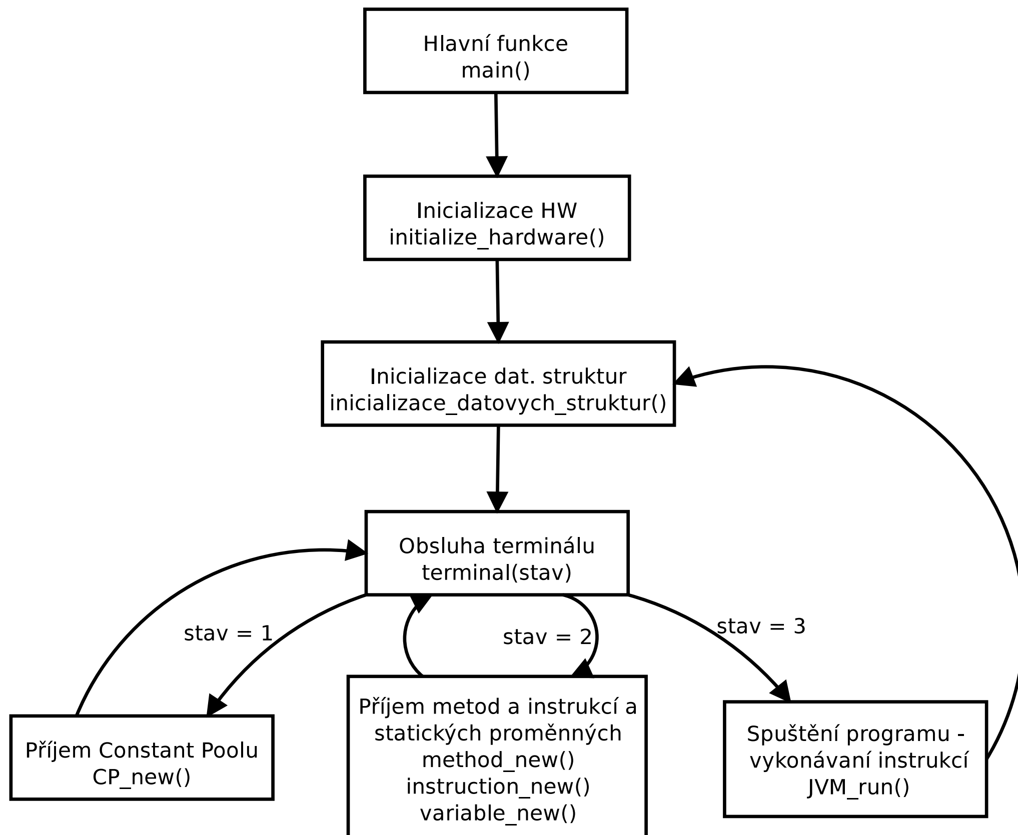
```
13: putstatic      #5            // Field delka_klice:I
```

Výsledná forma bude `13=179=5`. Dosáhne se jí pomocí následujícího regulárního výrazu, kde dodatečně bude ještě převeden název instrukce na operační kód.

```
String instrukce = "(\\s*)(\\d*)(:\\s*)([^\\s]*)(\\s*)([^\\s]*)([^/]*)(.*)";
```

4.2 FITkit Virtual Machine

Aplikace pro vestavěné zařízení FITkit je napsána v jazyce C. Vývojový diagram aplikace je na obrázku 4.2.



Obrázek 4.2: Vývojový diagram aplikace `FITkit Virtual Machine`

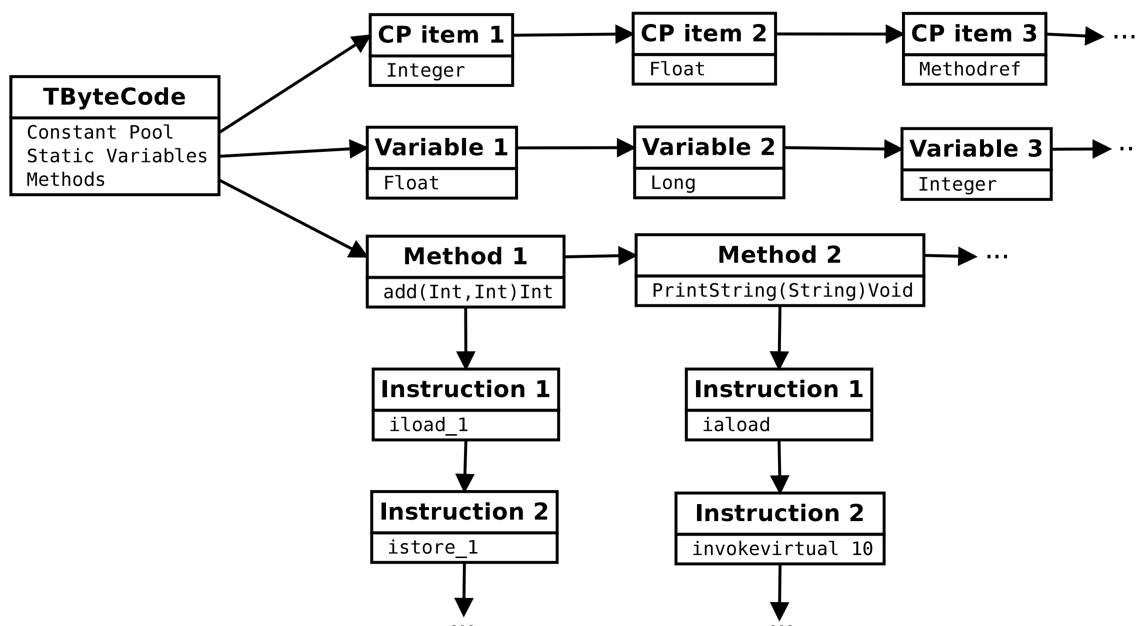
Celá aplikace je rozčleněna do několika souborů. Jak je vidět na obrázku 4.2, tak program začne ve funkci `main` nejprve inicializací hardware a datových struktur. Tyto funkce jsou definovány v souboru `main.c`. V inicializaci hardwaru se zinicializují periferie – LCD a klávesnice. V inicializaci datových struktur se ukazatele definují na konstantu `NULL` a dále je vytvořen na haldě zásobník o velikosti 50 jednotek (jednotka odpovídá 32 bitům).

Po inicializaci se přijímá bytekód z PC. Podle aktuálního stavu se buď vytváří nové položky v Constant Poolu, ve statických proměnných nebo v instrukcích a metodách. Obsluha terminálu a volání jednotlivých funkcí je implementována v souboru `terminal.c`. Funkce `terminal` porovnává prefixy přijatých řetězců a podle toho rozhoduje, o jakou položku se jedná.

Až přijde poslední ukončovací řetězec `==E==`, přejde program do stavu 3 a spustí se vykonávání programu. Po dokončení poslední instrukce se volá funkce `deleteByteCode`, která vyčistí RAM a aplikace opět čeká na příjem bytekódu. Pokud je programu v Javě napsána nekonečná smyčka, musí se zmáčknout tlačítko RESET na FITkitu, které hardwarově ukončí program a znovu se spustí funkce `main`.

4.2.1 Datové struktury Bytekódu

Do datových struktur bytekódu se ukládají přijatá data po sériové lince. Jedná se o položky Constant Poolu, statické proměnné a o instrukce jednotlivých metod.



Obrázek 4.3: Schéma uložení bytekódu do jednotlivých datových struktur v paměti RAM.

Na obrázku 4.3 se nachází schéma uložení bytekódu do paměti RAM v mikrokontroléru. Způsob uložení je založen na datových strukturách lineárního seznamu. Tento způsob jsem zvolil kvůli rychlosti vkládání jednotlivých položek na konec seznamu. Rovněž při čtení instrukcí, které se většinou provádí za sebou, je tento způsob uložení vhodný. Protože aktuálně prováděná instrukce obsahuje přímo ukazatel na následující instrukci. Jen při instrukcích skoku nebo volání metody je potřeba projít sekvenčně seznam, dokud se nenarazí na hledanou položku.

Reference na seznam Constant Poolu, statických proměnných a seznam metod jsou umístěny v globální proměnné typu `TByteCode`.

Položky Constant Poolu mohou obsahovat různé datové typy, nejdelší však je 64 bitový typ `long`. Proto položka s daty je datový typ `union` a obsah dat závisí na typu položky. Identifikací jednotlivých položek v CP je položka `number`. Instrukce bytekódu pokud čtou nebo zapisují položky v Constant Poolu, vždy mají v parametrech instrukce index, který odpovídá právě této identifikaci položky.

```
typedef struct IConstantPool
{
  int number;
  unsigned char type;
  unsigned char class; //pouze u metod a odkazu na promenne
  union Tdata data;
  struct IConstantPool *next;
}
TConstantPoolItem;
```

Položky statických proměnných jednoduchých datových typů rovněž mohou obsahovat maximálně 64 bitů pro datový typ `long`. Proměnné jsou identifikovány podle názvu proměnné, proto proměnná `name` ukazuje na řetězec s názvem. Při přístupu ke statickým proměnným se používají instrukce `putstatic` a `getstatic`, které zjišťují jméno proměnné přes položky v Constant Poolu, přímo ke statické proměnné tedy nelze přistoupit.

```
typedef struct IStaticVariable
{
  char *name;
  int64_t data;
  struct IStaticVariable *next;
}
TStaticVariable;
```

Metody jsou rovněž uloženy v lineárním seznamu, kde každá metoda obsahuje opět lineární seznam instrukcí. Každá metoda je identifikovaná pomocí jména, parametrů a návratové hodnoty. Například identifikátor metody může mít tvar `add(II)I`, kde metoda očekává v parametrech dva typ `int` a vrací opět typ `int`. Z toho vyplývá, že v aplikacích lze využít přetěžování funkcí. Můžeme mít více metod se stejným názvem, lišící se pouze parametry. Každá metoda má také uloženo ve struktuře, jaké má požadavky na paměť zásobníku a na paměť lokálních proměnných, tyto informace se využívají při vytváření nových rámců při volání metod.

```
typedef struct IMethod
{
  char *name;
  int stack;
  int locals;
  TInstructionsItem *firstInstr;
  TInstructionsItem *actInstr;
  struct IMethod *nextMethod;
}
TMethodItem;
```

Posledním datovým typem v přijímaném bytekódu jsou položky jednotlivých instrukcí. Instrukce jsou identifikovány číslem v rámci metody. Dále následuje jeden byte s operačním kódem a následují dva byty s parametry.

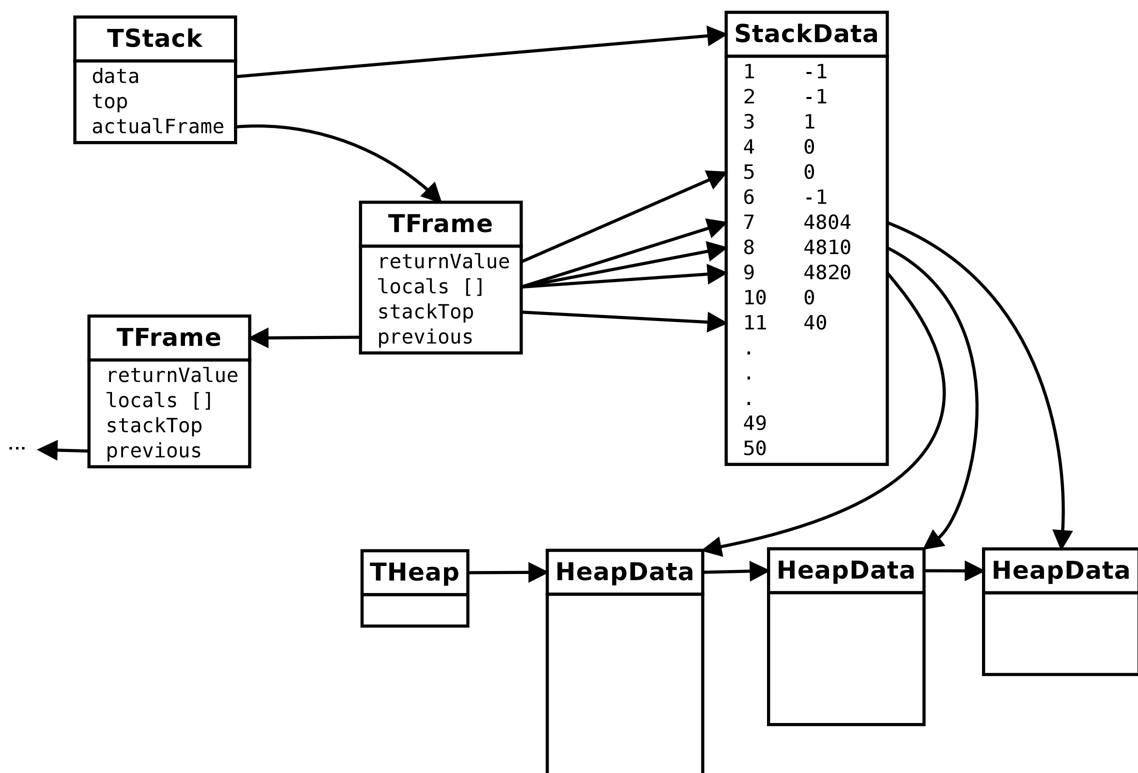
```
typedef struct IInstructions
{
  int number;
  unsigned char opCode;
  int parameter;
  struct IInstructions *next;
}
TInstructionsItem;
```

Funkce pro obsluhu operací s bytekódem se nacházejí v souboru `bytecode_obsluha.c`. Zajišťují vkládání položek, vyhledávání a mazání seznamů. Všechny funkce jsou implementovány pro každý typ položek. Funkce rovněž získávají jednotlivé informace z řetězců

přijatých po sériové lince. Bohužel překladač *mspgcc* neobsahuje implementaci funkce `scanf` nebo její modifikace `sscanf`. Tato funkce ze standardní knihovny zajišťuje získávání jednotlivých datových typů ze vstupu podle zadaného formátovacího řetězce. Za různé datové typy proměnných lze použít zástupných znaků. Kvůli efektivitě této funkce je v projektu obsažena implementace v souboru `scanf.c`. Původní zdrojový kód je převzat z projektu *embox*[2]. Tento projekt obsahuje implementaci funkce `scanf` pro vestavěná zařízení. Výsledná podoba funkcí v bakalářské práci se však částečně odlišuje kvůli provedené revizi kódu a odstranění několika chyb z originálu.

4.2.2 Datové struktury pro běh JVM

V této části budou popsány jednotlivé datové struktury a s nimi související funkce, které se starají o chod zásobníkového automatu virtuálního stroje Javy. Datové oblasti JVM jsou popsány v kapitole 2.1.4. Nyní se budeme zabývat zásobníkem (Java Virtual Machine Stacks), haldou (Heap) a obsluhou rámců (Frames). Na obrázku 4.4 je znázorněn příklad struktury jednotlivých datových oblastí.



Obrázek 4.4: Schéma datových struktur potřebných pro běh JVM.

Globální proměnná typu `TStack` uchovává referenci na hlavní zásobník (data) a index (top) na poslední alokovanou jednotku zásobníku. Najdeme zde také referenci na aktuální rámec právě vykonávané metody.

Hlavní zásobník je ve výchozím nastavení alokován pro 50 jednotek. Jedna jednotka podle specifikace JVM má velikost 32 bitů. Datové typy jako je `long` a `double` zabírají

[2]Embox, http://code.google.com/p/embox/source/browse/trunk/embox/src/lib/stdio/scanf.c?r=3467

dvě jednotky, čili 64 bitů. Na mikrokontroléru MSP430 však nelze používat 64 bitovou proměnnou typu `double`, proto všechny typy s pohyblivou řádovou čárkou jsou typu `float` na 32 bitů. Překladač Java zároveň převádí datové typy `byte`, `short` atd. na 32 bitů. Na hlavním zásobníku jsou uloženy lokální proměnné a operandy postupně ze všech rámců. Aktuální metoda však nemůže přistupovat k lokálním proměnným jiné metody. Proto každá metoda při svém volání alokuje určitý počet jednotek na hlavním zásobníku a k ostatním nemá přístup. Zde je popsána struktura rámce.

```
typedef struct JVMFrame
{
  int32_t **locals;
  int32_t *localsTop;
  int32_t *stackTop; //TOS
  int32_t *returnValue;
  TInstructionsItem *PCregisterBackup;
  TMethodItem *actMethod;
  struct JVMFrame* previous;
  char localsCount;
  char frameSize;
}
TFrame;
```

Ve struktuře rámce je důležitou proměnnou reference na vrchol lokálního zásobníku rámce (stackTOS). Tento vrchol zásobníku rámce používá velké množství instrukcí pro své operace. Tato proměnná je ovšem různá od indexu vrcholu hlavního zásobníku (top) ve struktuře `TStack`, který pouze říká, kde má případný nový rámec začínat. Instrukce bytekódu nemohou přistupovat k libovolné položce na zásobníku, ale pouze mohou odebrat vrchol zásobníku. Velikost zásobníku dané metody je známa již během překladu.

Další reference ve struktuře rámce jsou určeny pro obsluhu lokálních proměnných dané metody. K lokálním proměnným přistupují instrukce pomocí indexu. Počet položek, které budou potřeba v rámci(Frame) pro lokální proměnné je znám během překladu, ovšem nevíme, zda proměnná na daném indexu bude zabírat pouze jednu položku nebo u typu `long` položky dvě. Proto ve struktuře existuje pole ukazatelů na lokální proměnné (`locals`). Při vytváření rámce jsou naalokovány pouze tyto ukazatele a až během provádění kódu metody, když je potřeba výsledek uložit do lokálních proměnných, načte se hodnota volné adresy v lokálních proměnných (`localTop`). Tato adresa se uloží do pole ukazatelů a poté již můžeme k lokální proměnné přistupovat pomocí indexu.

Pokud metody nejsou typu `void` a tudíž vracejí nějakou hodnotu, je potřeba, aby se výsledek objevil na vrcholu zásobníku volající metody. U návratové hodnoty je jediný případ, kdy metoda může přistupovat na zásobník jiné metody, jinak bychom při rušení rámce přišli o výsledek. Reference `retrunValue` již při vytváření rámce obsahuje adresu, kam má při ukončení zkopírovat výsledek.

Registr PC uchovává právě prováděnou instrukci. Jde o strukturu, která uchovává operační kód instrukce, parametry a odkaz na následující instrukci. Při volání metody se skáče na první instrukci nové metody a registr se přepíše. Při ukončení se však musí pokračovat na následující instrukci po volání metody. Proto struktura rámce obsahuje i zálohu PC registru před voláním metody (`PCregisterBackup`).

Při provádění instrukcí skoku nebo podmíněných skoků je potřeba mít informaci v jaké metodě se právě nacházíme. Tato informace je v položce `actMethod` a je inicializována při

vytváření nového rámce.

Rámce tvoří lineární seznam, jen s tím rozdílem, že poslední rámec ukazuje na předchozí (proměnná `previous`). Tato implementace je výhodná, protože potřebujeme mít přístup pouze k poslednímu rámci. Při rušení se aktuální rámec uvolní a posledním se stane předchozí rámec.

Poslední dvě položky ve struktuře rámce jsou používány při odstraňování rámce. Proměnná `localsCount` uchovává počet lokálních proměnných pro uvolnění ukazatelů na lokální proměnné. Proměnná `framesize` udává počet jednotek, které rámec zabírá na hlavním zásobníku a tento počet je po ukončení uvolněn.

V programu lze rovněž dynamicky vytvářet proměnné primitivních datových typů. Tyto proměnné nebo pole proměnných se alokují v RAM mikrokontroléru. Jazyk Java obsahuje automatickou správu paměti (Garbage Collector), která v tomto projektu nebyla implementována. Struktura `THeap` si pouze zaznamenává do lineárního seznamu reference na alokované bloky. Tyto bloky jsou následně po ukončení programu uvolněny, aby nedocházelo ke ztrátě paměti(memory leaks).

Tyto datové struktury jsou deklarovány v souboru `typy.h`. Obsluha práce se zásobníkem a rámci je uložena v souboru `jvm.c`. V tomto souboru se rovněž nachází funkce `JVM_run`, která dekóduje jednotlivé instrukce a volá funkce na jejich obsluhu.

4.3 Významné funkce interpreteru

V předcházející sekci byly popsány datové struktury pro běh JVM. Nyní bude popsána implementace některých funkcí, které jsou často využívané. Základními funkcemi je dvojice funkcí pro vložení prvku na zásobník a odebrání prvku ze zásobníku.

```
void pushTOS (void *data, int size){
  if(data != NULL){
    memcpy(Stack.actFrame->stackTop, data, size * sizeof(int32_t));
  }
  else{
    *(Stack.actFrame->stackTop) = -1;
  }
  Stack.actFrame->stackTop += size;
}
```

Funkce `pushTOS` zajišťuje uložení prvku na zásobník, který je zadán adresou data. Velikost prvku je zadaná pomocí parametru size, který může nabývat hodnot 1 nebo 2 pro 64 bitové typy. Pokud se na zásobník ukládá reference na `NULL`, uloží se zde záporné číslo -1. Po uložení dat se zvýší adresa v ukazateli o velikost nově přidané položky.

```
int32_t *popTOS(int size){
  Stack.actFrame->stackTop -= size;
  return Stack.actFrame->stackTop;
}
```

Druhou funkcí je funkce `popTOS`, která vrací referenci na vrchol zásobníku operandů z aktuálního rámce a zároveň snižuje hodnotu vrcholu zásobníku.

4.4 Rozšíření o hardwarovou akceleraci v FPGA

Nad rámec zadání jsem se věnoval hardwarové akceleraci na platformě FITkit. U interpretovaných jazyků totiž dochází k určitému zpomalení běhu a nižšímu výpočetnímu výkonu, než kdybychom spustili přímo přeložený program pro daný procesor. Tato negativní vlastnost interpretovaných jazyků lze řešit mnoha způsoby. Například některé virtuální stroje jsou vybaveny JIT překladačem 2.2.1, které při spuštění provádí optimalizovaný překlad. Tento přístup může být však problematický u real-time vestavěných zařízení, kdy by docházelo k velkému zpoždění, nehledě na to, že by bylo obtížné přesunout překladač do mikrokontroléru.

Elegantním řešením však je u platformy FITkit využití FPGA k vykonávání časově náročných operací nebo obsluhovat operace časově náročné na odezvu. Základní myšlenka je, že v FPGA jsou napsány v jazyce VHDL některé časově náročné funkce. Mikrokontrolér pouze pošle přes rozhraní SPI do FPGA parametry a adresu funkce. Poté FPGA vrátí po sběrnici výsledek operace.

Při implementaci jsem nastudoval použití rozhraní SPI mezi MCU a FPGA na platfformě FITkit[3]. Na straně mikrokontroléru je vhodné použít funkce na přenos dat po sběrnici SPI. Funkce rovněž existují v modifikacích pro různě velkou datovou šířku adresy a dat. Funkce má následující tvar:

```
  //zapíše 16 bitové parametry na 8 bitovou adresu
FPGA_SPI_RW_A8_D16(SPI_FPGA_ENABLE_WRITE, ADRESA, parametry);
  //přečte 16 bitový result z 8 bitové adresy
result = FPGA_SPI_RW_A8_D16(SPI_FPGA_ENABLE_READ, ADRESA, 0);
```

Pro demonstraci využití hardwarové akcelerace v FPGA jsem použil proces, který sčítá dvě 8 bitová čísla a výsledek vrací na 16 bitech. V praxi lze samozřejmě napsat složitější proces, u kterého by bylo možné využít paralelismu FPGA a tak docílit výrazného zrychlení aplikace. Pro správnou funkčnost aplikace je potřeba na straně FPGA vytvořit entitu `add`, která je umístěna v souboru `add.vhd`. Důležité je, aby sčítačka provedla součet pouze pokud je aktivní signál `EN`.

Musíme rovněž doplnit soubor `top_level.vhd`, který obsahuje řadiče pro přijímání a odesílání dat pomocí sběrnice SPI. V tomto souboru nalezneme obsluhu LCD displeje nebo klávesnice. Každá komponenta má svou adresu, kterou musíme zadávat i na straně mikrokontroléru. Řadiče v tomto souboru musí předávat sériově přijatá data po sběrnici SPI na paralelní vektory signálů, které se spojí s cílovou komponentou. Konkrétně v tomto případě se jedná o 16 bitový vstup a výstup a logický signál `write_en`. V tomto souboru je tedy zapotřebí propojit řadič sčítačky s rozhraním SPI a entitu sčítačky spojit s řadičem sčítačky. Poté dosáhneme toho, že po zapsání dat z mikrokontroléru na adresu v FPGA se objeví data na vstupu určené entity. Obdobně, pokud chceme data z FPGA číst.

Tímto způsobem lze dosáhnout, aby aplikace byla napsána ve vysokoúrovňovém a interpretovaném jazyce Java a přitom zvládala i obsluhovat real-time aplikace nebo umožnila rychleji počítat některé funkce pomocí FPGA.

[3]Zdeněk Vašíček, http://merlin.fit.vutbr.cz/FITkit/docs/navody/tutor_blikac.html

4.5 Vývoj aplikací pro Interpreter na FITkitu

Při psaní aplikací pro vestavěné zařízení FITkit v jazyce Java musíme dodržet určitá omezení. Výsledná aplikace musí být v jedné třídě a jednovláknová. Jedinou třídu, kterou lze importovat je třída FITkit pro obsluhu nativních funkcí vestavěného zařízení.

4.5.1 Nativní funkce - třída FITkit

Nativní funkce jsou implementovány v souboru `FITkit_nativni.c`. Pro využití nativních metod ze třídy FITkit je nejprve nutné vytvořit instanci této třídy. Poté je možné přes rozhraní objektu volat jednotlivé metody. Pokud chceme použít nativní metody mezi různými metodami naší třídy, je vhodné deklarovat tento objekt jako statický. Následující příklad vypíše řetězec "Hello!"na LCD FITkitu:

```
import FITkit.*;
public class mojeTrida {
    static FITkit fitkit;  //deklarace statické proměnné třídy FITkit

    public static void main(String[] args) {
        fitkit = new FITkit();
        fitkit.PrintString("Hello!");
    }
}
```

Následující tabulka zobrazuje implementované nativní funkce ze třídy FITkit.

Hlavička metody	**Popis**
public int ReadKeyboard()	čeká na stisk klávesy, vrátí ascii hodnotu
public int ReadKeySt()	vrátí aktuální stav klávesnice
public void PrintChar(int c)	připojí na LCD znak c
public void PrintString(String s)	připojí na LCD řetězec s
public void PrintFloat(float f)	připojí na LCD číslo float f na 4 desetinná místa
public void PrintInteger(int i)	připojí na LCD Integer i
public void LCDclear()	vymaže LCD display
public int readTepmerature()	vrátí teplotu na MCU v celočíselném tvaru
public void delayMs(int ms)	zpoždění na zadaný počet milisekund ms
public void d5on()	rozsvítí diodu D5
public void d5off()	zhasne diodu D5
public void d6on()	rozsvítí diodu D6
public void d6off()	zhasne diodu D6
public int fpgaAdd(int a, int b)	pošle do FPGA spočítat součet čísel a,b

Tabulka 4.1: Přehled nativních metod.

4.5.2 Výpisy ladících informací

Při implementaci interpreteru v zařízeni FITkit jsem využíval ladících výpisků do terminálu. Ve výchozím stavu se tyto ladící výpisky nepoužívají, ale implementaci funkcí, které toto prováděly jsem zanechal zakomentované ve zdrojových kódech interpreteru. V souboru

`jvm.c` ve funkci `JVM_run` jsou zakomentované řádky, které zajišťují výpis aktuálně prováděné instrukce a stav zásobníku. Podle těchto výpisků lze zpětně sledovat, jaké hodnoty byly před a po provedení instrukce, zda vrchol zásobníku není mimo meze a podobně. Tyto funkce se mohou hodit při případném rozšiřování interpreteru.

```
   :182
--Stack--
|0 -1: 4740|
|1 -1: 4744|
|2 -1: 4748|
|3 10: 4752|
|4 32: 4756|
|5 0:  4760|
|6 42: 4764|
|7 10: 4768| <
|8 32: 4772|
--END STACK--
```

Zde je uveden příklad výpisu zásobníku po provedeni instrukce 182 (invokevirtual). V prvním sloupci je číslo položky zásobníku, ve druhém sloupci je hodnota v dekadickém tvaru položky zásobníku a ve třetím sloupci je adresa v paměti RAM mikrokontroléru. Zde je vidět, že položky na zásobníku zabírají 4 byty. Ukazatel vrcholu zásobníku zobrazuje položku pro zápis na zásobník (pro operaci push), ale pro čtení (pop) je položka o jednu pozici menší – v tomto případě je TOS položka číslo 6.

Kapitola 5

Zhodnocení výsledků

V této kapitole jsou představeny výsledky implementace a ukázky demonstračních aplikací.

5.1 Demo aplikace

Pro ověření funkčnosti interpretru bytového kódu byly vytvořeny demonstrační aplikace v jazyce Java. Aplikace jsou navrženy tak, aby demonstrovaly reálné využití interpreteru na vestavěných zařízeních. Tyto aplikace lze vyzkoušet a odladit i na PC, pokud ovšem nepracují se speciálními periferiemi FITkitu jako je teploměr a podobně.

5.1.1 Kalkulačka

Demonstrační příklad kalkulačka pracuje v nekonečné smyčce, kdy nejprve načte z klávesnice číslo typu `float`. Poté si uživatel může zvolit mezi čtyřmi základními aritmetickými operacemi (klávesa `A => +, B => -, C => *, D => /`) a poté zadat druhý operand typu `float`. Po stisknutí klávesy `#` se zobrazí výsledek na LCD displeji. Desetinná čárka se zadává pomocí klávesy `*`.

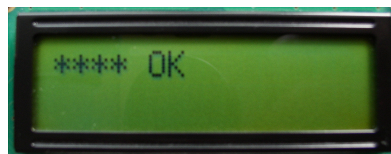
Obrázek 5.1: Demonstrační příklad kalkulačka.

V aplikaci kalkulačka se používá volání metody `calc()` v rámci třídy, dále se využívají řídící konstrukce typu `if - else` a cykly typu `while` a `for`. Pro vstup a výstup se využívají nativní funkce ze třídy `FITkit`.

5.1.2 Přístupový terminál

Tato demonstrační aplikace funguje tak, že se uživateli po napsání správného hesla objeví na LCD text `OK`. Pokud je heslo špatné, objeví se `BAD`. Program opět běží v nekonečné smyčce, během zadávání hesla se zobrazují na LCD pouze znaky `*` a po potvrzení klávesou `#` se zobrazí výsledek podle vstupu uživatele.

V aplikaci jsou použita dynamicky alokovaná pole typu `Integer`, která se alokují na haldě. Další vlastností je využití statické proměnné v rámci třídy, která je uložená v datové

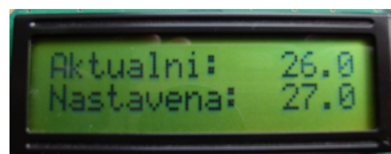


Obrázek 5.2: Demonstrační příklad přístupový terminál.

oblasti Constant Pool a na zásobníku je pouze index do CP. Využívají se běžné konstrukce `if - else`, cykly, práce s poli a využívání nativních funkcí. Například zpoždění, u kterého lze nastavit počet milisekund.

5.1.3 Regulátor teploty

Aplikace Regulátor teploty má za cíl spínat výstup (představme si například elektrickou spirálu) na základě aktuální teploty. Uživatel si může prostřednictvím klávesnice nastavit požadovanou teplotu, na kterou se má regulovat. Pokud je spirála aktivní, rozsvítí se LED diody D5 a D6 na FITkitu.



Obrázek 5.3: Demonstrační příklad regulátor teploty.

Implementace příkladu používá celočíselnou aritmetiku, teplota se měří na jedno desetinné místo stupně Celsia. Teplota je získávána přes nativní funkci, která vrací výsledek AD převodníku na mikrokontroléru. Teplotní čidlo je rovněž umístěno přímo na čipu.

Nastavování teplot probíhá pomocí tlačítek `A => +1` °C, `B => +0.1` °C, `C => -0.1` °C a `D => -1` °C. Jelikož implementovat v jazyce Java přerušení od klávesnice během smyčky s regulací by bylo problematické, zvolil jsem při zadávání teplot způsob, kdy se musí klávesa držet stisklá. Poté program zjistí při průchodu smyčkou stav kláves A, B, C nebo D a podle toho upraví požadovanou teplotu.

5.1.4 Sčítání s akcelerací v FPGA

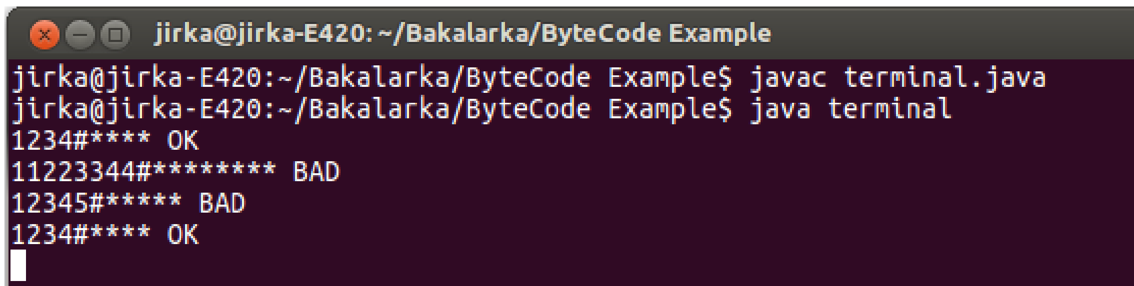
Tato aplikace demonstruje využití propojení FPGA a interpreteru v MCU k provedení výpočtu. Na LCD se zobrazí dva řádky s jednoduchým výpočtem součtu dvou celých čísel. Ovšem první řádek je vypočítán na mikrokontroléru, zato výsledek druhého řádku je vypočítán v FPGA. Do FPGA jsou pouze předány dva parametry a poté se přečte výsledek.

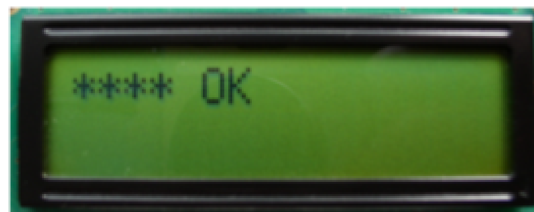
Obrázek 5.4: Demonstrační příklad pro akceleraci výpočtů v FPGA.

5.2 Dosažené výsledky

V rámci bakalářské práce jsem nastudoval problematiku jazyka Java, zejména vlastnosti přenositelného bytekódu a virtuálního stroje Javy. Zaměřil jsem se na strukturu souborů class, na jeho části jako je Constant Pool, statické proměnné a instrukce v jednotlivých metodách. Po studiu instrukční sady bytekódu byla vybrána část instrukcí, které budou implementovány v zařízení FITkit a jsou vhodné pro vestavěná zařízení.

Dalším bodem byl návrh samotného interpreteru a způsob, jak bytekód bude zaveden do FITkitu. Pro získání bytekódu ze zdrojových kódu jazyka Java jsem implementoval program pro PC `FITkit Loader` na psaný v jazyce Java. Tento program zajišťuje komunikaci po sériové lince s MCU FITkitu a dokáže upravit bytekód do navrženého protokolu, který se odešle.





Obrázek 5.5: Aplikace v Javě lze spouštět jak na PC, tak na vestavěných zařízeních.

Ve vestavěném zařízení se poté přijatý bytekód spustí a vykoná. Je zde implementována zásobníková architektura pro zpracovávání instrukční sady bytekódu jazyka Java. Na platformě FITkit byly také implementovány nativní funkce pro obsluhu periferií a příklad využití FPGA k urychlování výpočtu.

Kapitola 6

Závěr

V předchozích kapitolách byly popsány nejprve teoretické informace o problematice jazyka Java na vestavěných zařízeních. Před návrhem interpreteru jsem studoval specifikaci virtuálního stroje Javy. Například jednotlivé datové oblasti JVM, instrukční sadu a strukturu bytekódu. V další kapitole byl popsán návrh interpreteru pro vestavěné zařízení. Při návrhu jsem vytvořil protokol, pro odesílání bytekódu do zařízení FITkit. Interpreter je navržen pro jednovláknové aplikace, které mohou využívat některé periferie FITkitu, jako je klávesnice, display, LED nebo teploměr.

Dalším bodem byla implementace interpreteru v jazyce C. Zde je použita zásobníková architektura, kde instrukce pracují s vrcholem zásobníku. Pro překlad ze zdrojových kódů Javy do bytekódu a následné odeslání po sériové lince, jsem vytvořil aplikaci pro PC, která je napsána v jazyce Java. Aplikace funguje jako terminál sériové linky s možností automatického odeslání celého bytekódu. Výsledek implementace byl vyzkoušen a předveden na demonstračních aplikacích. Tímto byly splněny jednotlivé body zadání. Nad rámec zadání jsem rovněž přidal do práce příklad použití FPGA k akcelerování výpočtů. Jedná se o sčítání dvou čísel v FPGA, kde jsou parametry odesílány z MCU po sběrnici SPI.

Právě ve využití vysokoúrovňových programovacích jazyků a v akcelerovaných výpočtech v FPGA, vidím velký potenciál pro řešení nejrůznějších aplikací. Tento přístup umožňuje vývojářům psát přenositelný kód, který může být doplněn implementací časově náročných operací v hardwaru, což je velkým přínosem pro vývoj aplikací na vestavěných zařízeních.

Při pokračování v rámci diplomové práce se otvírá mnoho možností. Například by šel interpreter rozšířit o podporu objektové orientace nebo práci s vyjímkami. Další možností je implementace řady tříd pro práci s abstraktními datovými typy, práci se sítí, databázemi nebo rozšířit interpreter o automatickou správu paměti (Garbage Collector).

Literatura

[1] LINDHOLM, T.; YELLIN, F.: *The Java Virtual Machine Specification, Java SE 7 Edition*. Addison-Wesley, 2013, ISBN 978-0-133-26046-5.

[2] TIŠNOVSKÝ, P.: Seriál Programovací jazyk Java a JVM [online]. http://www.root.cz/serialy/programovaci-jazyk-java-a-jvm/, [cit. 2013-01-12].

[3] UNDERWOOD, S.: mspgcc - A port of the GNU tools to the Texas Instruments MSP430 microcontrollers. http://mspgcc.sourceforge.net/manual/, 2012 [cit. 2013-01-12].

[4] VAŠÍČEK, Z.: Fakulta informačních technologií VUT Brno. http://merlin.fit.vutbr.cz/FITkit/, 2012 [cit. 2013-01-12].

[5] VENNERS, B.: The Java Virtual Machine. http://www.artima.com/insidejvm/ed2/jvm.html, 2012 [cit. 2013-01-12].

[6] Wikipedia: Byte code. http://cs.wikipedia.org/wiki/Byte_code, 2012 [cit. 2013-01-12].

[7] Wikipedia: Java bytecode instruction listings. http://en.wikipedia.org/wiki/Java_bytecode_instruction_listings, 2012 [cit. 2013-01-12].

[8] Wikipedia: Java Development Kit. http://en.wikipedia.org/wiki/Java_Development_Kit, 2012 [cit. 2013-01-12].

[9] Wikipedia: List of Java virtual machines. http://en.wikipedia.org/wiki/List_of_Java_virtual_machines, 2012 [cit. 2013-01-12].

[10] Wikipedia: Just-in-time compilation. http://en.wikipedia.org/wiki/Just-in-time_compilation, 2013 [cit. 2013-04-29].

[11] WWW: MSP430F168. http://www.ti.com/product/msp430f168, 2013 [cit. 2013-04-29].

Příloha A

Obsah DVD

DVD obsahuje zdrojové kódy programů, dokumentaci k programům i přeložené soubory. Software lze buď ručně nainstalovat nebo na DVD je připravený obraz pro virtuální počítač, kde je již všechno nachystáno. Na DVD je soubor `readme.txt`, který obsahuje popis instalace pro jednotlivé OS.

Adresář	**Popis**
`build`	Obsahuje binární program pro nahrání bytekódu i Interpreter
`demoApp`	Zdrojové soubory demonstračních aplikací
`doc`	Dokumentace ve formátu HTML (Javadoc, Doxygen)
`ImageVirtualBox`	Obraz virtuálního počítače s nainstalovaným SW
`RxTx`	knihovna pro sériovou linku
`src`	zdrojové kódy programů

Příloha B

Instalace softwaru

Na přiloženém DVD je obraz pro virtuální PC s nainstalovaným softwarem. V případě ruční instalace je potřeba číst následující řádky. Aplikace `FITkit Virtual Machine` určená pro platformu FITkit se překládá a nahrává standardně přes `qDevKit`[1]. Stačí aplikaci nakopírovat do složky s ostatními projekty, poté například přes grafické prostředí přeložit a nahrát do zařízení. Po úspěšném překladu a nahrání aplikace se na display zobrazí `Pripraveno - prijem bytekodu`. V tomto okamžiku aplikace čeká na příjem bytekódu po sériové lince.

B.1 Instalace `ByteCode Loader`

Tato aplikace je určena pro OS Windows i Linux. Je napsána v jazyce Java, proto vyžaduje mít nainstalováno na počítači běhové prostředí Java. Nyní už aplikace lze spustit. Pravděpodobně však budete muset do knihoven Javy nakopírovat knihovnu obsluhující sériovou linku. Tato knihovna je vázána na operační systém a na 32 bitovou nebo 64 bitovou architekturu. V krátkosti tedy pro úspěšné spuštění aplikace je potřeba splnit tyto body:

- Nainstalované běhové prostředí Javy (typicky je již v systému)
- Nainstalovanou knihovnu pro sériový port RxTx
- Nainstalovaný software JDK (v případě pokud chcete vyvíjet či upravit aplikaci)

B.1.1 Instalace knihovny sérového portu RxTx

Program `ByteCode Loader` používá pro obsluhu sériové linky knihovnu RxTx[2]. Pro zprovoznění knihovny je zapotřebí nakopírovat do složky knihoven Java soubor s nativní implementací obsluhy sériového portu. Tento soubor je závislý na platformě, proto je na DVD obsažen ve verzích pro Windows i Linux a to jak pro 32 bitovou, tak i pro 64 bitovou architekturu.

B.1.2 Instalace JDK

Software Java Development Kit (JDK)[3] je sada programů pro vývojáře v jazyce Java. V aplikaci `ByteCode Loader` se využívají programy `javac` a `javap`. Program `javac` je překla-

[1] http://merlin.fit.vutbr.cz/FITkit/navody.html
[2] http://rxtx.qbang.org/wiki/index.php/Main_Page
[3] http://www.oracle.com/technetwork/java/javase/downloads/index.html

dač jazyka Java a ze zdrojového souboru vytvoří soubor class. Ten poté dekompilujeme pomocí programu `javap`, který nám vypíše v textové podobě strukturu bytekódu.

Pokud pouze chcete vyzkoušet demo příklady z DVD, není potřeba tento software instalovat. Stačí strukturu posílaného bytekódu označit a překopírovat text do oblasti *pro FITkit* a poté nahrát do zařízení. Pokud si však chcete napsat vlastní program v Javě pro FITkit, je nutné tento software nainstalovat.

V systému Linux se po nainstalování balíků dá volat z příkazové řádky pomocí příkazů `javac` a `javap`. V systému Windows je potřeba zadat do proměnné PATH cestu k adresáři s těmito programy. Typicky je potřeba přidat do proměnné PATH řetězec:

```
;c:\Program Files\Java\jdk1.7.0_17\bin
```

Kde se samozřejmě cesta může lišit podle nainstalované verze Javy. Funkčnost nainstalovaného software JDK ověříme, když v příkazovém řádku zadáme příkazy `javac` a `javap`, které by měly spustit dané aplikace a vypsat nápovědu. Poté je také možnost vyzkoušet přímo překlad v programu `ByteCode Loader`, kde lze vybrat soubor se zdrojovými texty jazyka Java a přeložit ho tlačítkem *Načíst a přeložit.*

Obrázek B.1: FITkit připraven pro příjem bytekódu.