VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

# PŘEKLADAČ JAZYKA C V PROSTŘEDÍ PYTHON

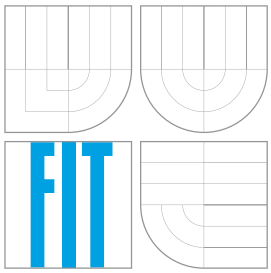
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

TOMÁŠ FIEDOR

BRNO 2012

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

# PŘEKLADAČ JAZYKA C V PROSTŘEDÍ PYTHON

C COMPILER IN PYTHON

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

TOMÁŠ FIEDOR

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. ZDENĚK VAŠÍČEK

BRNO 2012

# Abstrakt

V současné době neexistuje žádné výrazné propojení problematiky tvorby překladačů a návrhu procesorů a jejich instrukčních sad ve výuce. Cílem této práce je vytvořit snadno rozšiřitelný modulární překladač, který bude umožňovat experimentovat s instrukční sadou použitého cílového procesoru. Překladač implementuje několik optimalizačních technik, jejichž vliv je v práci diskutován. Jednou z pokročilejších použitých technik je kontextové generování cílového kódu, které vykazuje mnohem lepší metriky výsledného kódu v porovnání s prostým slepým generování.



# Abstract

There is currently no big link between creation of compilers and processor design and their instruction sets in courses. The goal of this work is to create easily extensible and modular compiler, which will enable experiments with instruction sets of used target processor. Compiler implements several optimization techniques. Their impact is more closely discused. One of the advanced used techniques is context generation of output code. This technique generates less code than common blind generation.



# Klíčová slova

Python, jazyk C, vysoko-úrovňové překladače, návrh procesorů, architektury procesoru, generování kódu, optimalizace kódu

# Keywords

Python, C language, high-level compiler, procesor design, procesor architectures, code generation, code optimization


# Citace

Tomáš Fiedor: Překladač jazyka C v prostředí Python, bakalářská práce, Brno, FIT VUT v Brně, 2012

# Překladač jazyka C v prostředí Python

## Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana inženýra Zdeňka Vašíčka

.....
Tomáš Fiedor
4. května 2012

## Poděkování

Děkuji panu vedoucímu inženýrovi Zdeňkovi Vašíčkovi za všechny konzultace a rady spojené s tvorbou bakalářské práce, ale i s pomocí při učásti na studentské konferenci EEICT.

# Obsah

# Kapitola 1

# Úvod

Kurzy zaměřené na problematiku tvorby překladačů a návrhu procesorů a jejich instrukčních sad se zpravidla vyučují izolovaně, bez širší návaznosti. Neexistuje mezi nimi žádné logické propojení, které by zaručilo zachycení všech souvislostí spojených s překladovou činností zdrojových souborů do strojového kódu nebo instrukcí assembleru a jejich následné zavedení a zpracování na cílovém procesoru.

Hlavní motivací pro vytvoření dalšího překladače je podpora výuky a umožnění experimentování s různými implementacemi jednotlivých částí překladače, s různými instrukčními sadami, případně s možností rozšíření cílového procesoru. Cílem je vytvořit překladač dostatečně dokumentující výstupy a s nimi spojené metriky všech fází překladu, možnost libovolných kombinací zvolených optimalizací či instrukčních sad a tím pak sledovat jejich dopad na velikost či efektivitu výsledného kódu.

Struktura překladače je postavena na základních postupech vyučovaných na Fakultě informačních technologií a vychází z běžných zažitých postupů pro jeho tvorbu. V kapitole 2 si definujeme základní teoretické podklady, na kterých jsou postaveny použité postupy při implementaci překladače. Současně si stanovíme základní princip překladu a modely pro jejich vytvoření. Velký význam pro tento projekt má rovněž problematika návrhu procesorů a instrukčních sad v kapitole 4.

Jako implementační jazyk byl zvolen skriptovací jazyk Python, především díky jeho objektově orientovaném přístupu a dynamickému typování, usnadňující docílení nezávislosti modulů a tím i možnost jejich výměny. Současně je pro tento jazyk dostupná řada nástrojů usnadňující tvorbu přední části překladače. Z těchto nástrojů jsem vybral tři v současné době nejpoužívanější a nejznámější a podrobil je bližší analýze v kapitole 3.

V kapitole 5 je popsáno vše spojené s implementací překladače. Všechny části jeho překladu, potřebné definované struktury a připadné problémy, které jsem musel při tvorbě překladače řešit případně možnosti, mezi kterými jsem se rozhodoval. Po samotné implementaci překladače jsem se věnoval testováním vytvořeného projektu a experimentování, kde jsem demonstroval vliv různých jevů (rozšíření procesoru, použití optimalizací, ...) na velikost výsledného cílového kódu. Více o těchto experimentech a jejich zhodnocení je uvedeno v kapitole 6.

# Kapitola 2

# Úvod do teorie překladačů

Implementace překladačů vychází z matematických disciplín a teorie formálních jazyků. Před samotným popisem konkrétní implementace překladače jazyka C je vhodné stručně vysvětlit základní pojmy, metody a modely popisu pro tvorbu překladačů. Definujme si v prvé řadě základní stavební kameny, na kterých zvolené postupy a nástroje stojí [4]. Se stručným úvodem do teorie jazyků pak lze popsat základní činnost překladače [9] a popsat jeho základní modely a metody pro realizaci tzv. *přední části* překladače [9, 3].

## 2.1 Stručný úvod do teorie formálních jazyků

Teorie formálních jazyků je jednou z hlavních oblastí informatiky, která se výrazně projevila při vývoji překladačů a mimo jiné umožnila vznik základních principů syntaxí řízeného překladu, na kterých je položena výsledná implementace. Její základní myšlenkou je formalizace popisu přirozeného jazyka tak, aby sloužil ke komunikaci mezi člověkem a počítačem. Pracujeme tak se dvěma matematickými entitami – s gramatikou a s automatem – které dohromady představují abstraktní matematický stroj.

Před samotnou definicí jazyka (viz 2.1.3) je nejprve nutno vymezit, co je to *abeceda* (viz 2.1.1) a *řetězec* (viz 2.1.2).

**Definice 2.1.1.** Abeceda $\Sigma$ *je konečná, neprázdná množina elementů, které nazýváme,* symboly.

**Definice 2.1.2.** *Nechť* $\Sigma$ *je abeceda. Pak* $\epsilon$*, tzv.* prázdný řetězec *neobsahující žádný symbol, je řetězec nad abecedou* $\Sigma$*. Pokud* $x$ *je řetězec nad abecedou* $\Sigma$ *a* $a \in \Sigma$*, pak* $xa$ *je také řetězec nad abecedou* $\Sigma$*.*

**Definice 2.1.3.** *Nechť* $\Sigma$ *je abeceda a* $\Sigma^*$ *značí množinu všech řetězců nad abecedou* $\Sigma$ *včetně prázdného řetězce. Pak každá podmnožina* $L \subseteq \Sigma^*$ *se nazývá* jazyk *nad abecedou* $\Sigma$*.*

Nejznámějším prostředkem pro reprezentaci konečných i nekonečných jazyků je gramatika (viz 2.1.4). Gramatika využívá dvou konečných disjunktních abeced – *neterminálů* (aneb syntaktické kategorie) a *terminálů* (identické s abecedou, nad kterou definujeme jazyk). Jejich sjednocení nazýváme *slovníkem gramatiky*. V gramatice lze za pomocí přepisovacích pravidel generovat řetězce tvořené pouze terminály, reprezentující věty gramatikou definovaného jazyka.

**Definice 2.1.4.** *Gramatika $G$ je čtveřice $G = (N, \Sigma, P, S)$, kde $N$ je konečná množina neterminálů, $\Sigma$ je konečná množina terminálů, kdy $N \cap \Sigma = \emptyset$, $P$ je množina přepisovacích pravidel, ve tvaru uspořádané dvojice $(\alpha, \beta)$ řetězců, která stanovuje možnou substituci řetězce $\beta$ namísto řetězce $\alpha$, a $S \in N$ je počáteční symbol gramatiky*

V překladačích pak pracujeme s bezkontextovými jazyky, kterými jsme schopni popsat většinu současných programovacích jazyků a současně pro ně existují algoritmy pro analýzu vět z nich generovaných. Popis programovacího jazyka a jeho analýza za pomocí bezkontextové gramatiky je hlavní podstatou implementace přední části překladače.

## 2.2 Fáze kompilace

Proces kompilace zpravidla sestává z šesti fází, které v každém kroku transformují vstupní program z jedné reprezentace do jiné. Počáteční vstupní reprezentace programu je rozbita pomocí lexikální analýzy do samostatných lexikálních jednotek zvaných *lexémy*, verifikuje jejich správnou formu a zasílá je syntaktickému analyzátoru ve formě tzv. *tokenů*.

Syntaktická analýza rozhoduje o syntaktické správnosti tokenizovaného vstupního souboru. Syntaktický analyzátor aplikuje pravidla speficifované gramatiky a postupně vytváří derivační strom, kde uzly jsou tokeny, a vztahy otec-syn reprezentují graficky pravidla. Vytvořený derivační strom je následně zpracován pomocí sémantické analýzy, která kontroluje sémantické konvence vstupního jazyka, jako je například typová kontrola nebo pozice příkazů v kódu. Lexikální, syntaktická a sémantická analýza společně tvoří tzv. *přední část překladače*, pro kterou existují bližší teoretické podklady (viz. podkapitoly 2.3 a 2.4).

Pokud první tři fáze proběhly bez chyb, můžeme vygenerovat mezikód reprezentující sémanticky ekvivalentní program. Hlavní význam má mezikód nejen u kompilátorů nezávislých na cílové architektuře, ale i pro následnou optimalizací a efektivního generování cílového kódu. Pátou fází je pak optimalizace, jenž je zpravidla závislá na cílové architektuře a použité výstupní instrukční sadě. Jejím hlavním významem je převod mezikódu do kratší a/nebo efektivnější formy. Ve finální fázi provádíme generování cílového kódu, kdy mezireprezentaci programu přeložíme do sekvence instrukcí assembleru.

## 2.3 Modely lexikální analýzy

Rozpoznávané lexémy vstupního souboru můžeme popsat pomocí slovního popisu, gramatik, konečných automatů a nebo regulárních výrazů. V praxi se používají všechny vyjmenované způsoby, blíže se však budeme zabývat posledními dvěma, které jsou vzájemně převoditelné a snadno implementovatelné.

*Regulární výrazy* specifikují vzor symbolu, kdy každý vzor odpovídá pojmenované množině řetězcu. Ne všechny jazyky jsou však regulárními výrazy popsatelné (zejména jedná-li se o počítání hloubky zanoření, apod.), pro základní jazyky nám však postačí. Vycházíme z následující definice:

**Definice 2.3.1.** *Nechť $\Sigma$ je abeceda.* Regulární výrazy *nad abecedou $\Sigma$ a* jazyky, *které značí, jsou definovány následovně: $\emptyset$ je regulární výraz značící prázdnou množinu, $\epsilon$ je regulární výraz značící jazyk $\{\epsilon\}$ a $a$, kde $a \in \Sigma$, je regulární výraz značící jazyk $\{a\}$.*

Tuto základní definici dále rozšíříme o operátory konkatenace (.), sjednocení (+) a iteraci (*), kde pro dva regulární výrazy $r$ a $s$ značící jazyky $L_r$ a $L_s$, definujeme sémantiku

vyjmenovaných operací následovně: $(r.s)$ je regulární výraz značící jazyk $L = L_r L_s$, $(r + s)$ je regulární výraz značící jazyk $L = L_r \cup L_s$ a $(r^*)$ je regulární výraz značící jazyk $L = L_r^*$.

Alternativním modelem jsou pak *konečné automaty*. *Konečný automat* je pětice $M = (Q, \Sigma, R, s, F)$, kde $Q$ je konečná množina stavů automatu, $\Sigma$ vstupní abeceda, $R$ konečná množina pravidel ve tvaru $pa \to q$ (kde $p, q \in Q, a \in \Sigma \cup \{\epsilon\}$), $s \in Q$ je počáteční stav automatu a $F \subseteq Q$ je množina koncových stavů. Oba definované modely jsou vzájemně zaměnitelné, jelikož existují algoritmy pro převod konečných automatů na regulární výrazy a naopak.

## 2.4 Metody syntaktické analýzy

Z nejčastěji používaných metod syntaktické analýzy budeme uvažovat pouze dva základní způsoby *shora dolů* a *zdola nahoru* [3, 9]. Metody se v principu odlišují podle postupu skrz posloupnost tokenů získaných z lexikální analýzy a tvoření výsledného derivačního stromu.

Metoda *shora dolů* vytváří derivační strom od jeho kořene. Jednou z možných implementací je *rekurzivní sestup*, kdy pro analýzu každého nonterminálního symbolu vytvoříme samostatnou proceduru. Pro každý nonterminál existují pravidla, v jejichž těle se nachází další symboly, pro které voláme příslušné procedury a postupně se zanořujeme do analyzovaného řetězce. Jednotlivé procedury zjišťují výskyt požadovaného symbolu, podle těla pravidla, a v případě jeho nepřítomnosti zahlásí syntaktickou chybu. V opačném případě se navrátí do otcovské procedury a umožní postupné vytváření derivačního stromu.

Druhý přístup, *zdola nahoru*, konstruuje derivační strom od jeho listu směrem ke kořeni, za pomocí tzv. přesunů a redukcí. Při každém redukčním kroku nahradíme podřetězec, který se shoduje s pravou stranou přepisovacího pravidla, symbolem na jeho levé straně. Implementace analyzátoru tohoto typu vyžaduje zásobník s označeným dnem. Proud tokenů z lexikálního analyzátoru postupně vkládáme na vrchol zásobníku dokud není rozpoznána pravá strana některého z redukčních pravidel. V tu chvíli lze provést redukční pravidlo, z vrcholu zásobníku odstranit tokeny pravé strany a na vrchol umístit stranu levou. Redukce provádíme tak dlouho dokud nedojde k chybě nebo dokud se na vrchol zásobníku nedostane startovací symbol, kdy se algoritmus zastaví a úspěšně ukončí analýzu.

# Kapitola 3

# Prostředky pro tvorbu překladačů

V dnešní době existuje nespočet nástrojů usnadňující vytváření překladačů, především k urychlení vývoje lexikální a syntaktické analýzy. Programovací jazyk Python v tomto ohledu není výjimkou a poskytuje spoustu parsovacích prostředků, ať už klasických či představujících netradiční přístupy k analýze zdrojového textu.

Z řady existujících modulů v Pythonu jsem se rozhodl zanedbat starší moduly jako jsou například Flex, Plex, či Bison, a zaměřil jsem se na ty novější a nejvíce používané i udržované. Mezi hlavní kandidáty pro tvorbu cílového překladače jsem vybral reimplementaci klasických nástrojů pro rozbor gramatik Lex a Yacc zvanou PLY [14], generátor lexikálních a syntaktických analyzátorů z popisu gramatiky pro cílovou platformu Python ANTLR [11] a také mírně odlišný přístup, jenž nabízí modul PyParsing [8], který analyzuje kód tvořením parsovacích elementů. Poslední diskutovanou možností je vytvoření vlastního parsovacího nástroje šitého na míru cílové gramatice. Všechny tyto čtyři přístupy jsem zkoumal a zaměřil se především na rychlost zpracování a nároků na implementaci z hlediska počtu řádků kódu.

Rychlost byla zkoumána pomocí skriptu, ve kterém jsem implementoval parser jednoduché aritmetické kalkulačky ve všech vybraných nástrojích, a časoval dobu provádění na delších a složitějších výrazech. Gramatika aritmetické kalkulačky na obrázku 3.1 sestává ze čtyřech operátorů (sčítání, odčítání, násobení a dělení), s možností uzávorkování výrazů a operandů zadávaných ve čtyřech možných číselných soustavách (dekadická, hexadecimální, oktálová a binární), viz gramatika 3.1.

```
addsubExpression ::= muldivExpression ('+'|'-' muldivExpression)*
muldivExpression ::= atomicOperand ('*' | '/' atomicOperand)*
atomicOperand ::= '(' expression ')'
         | decadicNumber | hexadecimalNumber | octalNumber | binaryNumber
decadicNumber ::= [0-9]+
hexadecimalNumber::= '0h' [0-9a-fA-F]+
octalNumber ::= '0o' [0-7]+
binaryNumber ::= '0b' [01]+
```

Obrázek 3.1: BNF jednoduché aritmetické kalkulačky, použité pro testování rychlosti parsovacích nástrojů v jazyce Python

## 3.1 PLY

PLY je parsovací nástroj vytvořený Davidem Beazleym reimplementující klasické UNIXové parsovací nástroje Lex a Yacc [14]. Hlavní důraz v tomto nástroji je kladem na použití regulárních výrazů a samozřejmě bezkontextové gramatiky pro popis syntaxe.

Tento nástroj jednoznačně rozlišuje mezi lexikální a syntaktickou analýzou. Jeho lexer je navíc tak silný, že si na zpracování jednoduchých gramatik vystačíme pouze s ním. PLY si hluboce zakládá na jmenných konvencích, které musíme při tvorbě analyzátoru respektovat. Jak lexer tak parser pracuje s proměnnou `tokens`, seznamem jmen tokenů gramatiky. Ke každému z tokenů přísluší korespondující funkce nebo proměnná s prefixem `t_` obsahující regulární výraz popisující formu tokenu nebo alternativně akce, která se při parsování tokenu vykoná (příkladem může být odstranění bílých znaků nebo převod všech znaků na malá písmena). Speciální funkcí je `t_error()`, volaná při nerozpoznaném nebo chybném tokenu a umožňuje ošetření tohoto stavu nebo ukončení analýzy.

Parser je založen na podobných konvencích, kdy pro každé pravidlo BNF vytvoříme korespondující funkci s prefixem `p_` a s *docstringem*[1], ve kterém je napsána přesná podoba pravidla. K jednotlivým částem pravidla je pak možné přistupovat pomocí indexu. Precedence a asociativita se nastavuje pomocí proměnné `precedence`, které se předá seznam podle priority, která indikuje asociativitu tokenů. Podobně jako u lexeru i zde je speciální funkce `p_error()` pro ošetření chybových stavů.

### Implementace kalkulačky

Při vytváření kalkulačky v PLY se nejprve stanovily tokeny vyskytující se v modelované gramatice. Pro každý z těchto tokenů se vytvořily příslušné funkce a proměnné specifikující jejich podobu a případnou funkci. Poté se pro každé pravidlo gramatiky vytvořila korespondující funkce, v jejimž těle se vykonávaly přidružené akce. Implementace vyžadovala 100 řádků kódu.

## 3.2 ANTLR

ANTLR (ANother Tool for Language Recognition) [11], je framework pro automatické generování lexikálních a syntaktických analyzátoru na základě gramatického popisu podporující řadu cílových jazyků. Součástí ANTLR je aplikace ANTLRWorks pro visualizaci, testování a tvorbu vstupní gramatiky. Z popisu gramatiky se poté vygeneruje modul syntaktické a lexikální analýzy v kódu podle zvoleného cílového jazyka.

Při tvorbě analyzátoru musíme nejprve vytvořit gramatiku, která popisuje formu všech lexikálních jednotek v zdrojovém souboru s příponou `.g`. Do vytvořených gramatických elementů pak vložíme na příslušné pozice úseky nativního cílového kódu, které se při analýze provedou. Po vygenerování lexeru a parseru získáme dva moduly a seznam tokenů v dobře čitelné podobě.

ANTLR mimo to poskytuje alternativní přístup využívající stromů. Použije se vytvořený popis gramatiky, ale místo vestavěného kódu přiřadíme pravidla pro vytvoření abstraktního syntaktického stromu. Po vytvoření stromu použijeme parser stromů pro spuštění vestavěných akcí. Procházení abstraktní stromu je u složitějších gramatik mnohem rychlejší než vestavěné kódy.

[1] řetězcový literál specifikovaný ve zdrojovém kódu sloužící v Pythonu pro dokumentaci funkcí a umístění dokumentačních testů

### Implementace kalkulačky

Většina práce při tvorbě kalkulačky za se odehrála v nástroji ANTLRWorks, specifikací gramatiky a přiřazení vestavěných akcí k jednotlivým pravidlům. Z tohoto popisu se ve stejném nástroji vygenerovaly moduly lexeru a parseru, které se pak inicializovaly v jednoduché několika řádkové funkci testovacího skriptu. Velkou výhodou ANTLR je především nástroj ANTLRWorks, pomocí kterého se dobře ladí, visualizuje i píše vstupní gramatika. Na druhou stranu jeho hlavní nevýhodou v souvislosti s použitím v Pythonu je fakt, že dostupnou cílovou verzí generovaného kódu i samotného modulu je Python verze 2.X.

## 3.3 PyParsing

PyParsing je mírně odlišným přístupem vytvořený Paulem McGuirem [8]. Jeho hlavní předností je schopnost vytvářet parsery s minimálním počtem řádků kódu. Narozdíl od nástrojů Lex a Yacc, PyParsing nerozlišuje mezi lexikální a syntaktickou analýzou a poskytuje funkce a třídy pro vytváření parsovacích elementů. Součástí modulu jsou předdefinované elementy, které můžeme kombinovat a vytvářet tak vlastní elementy, například spojováním dohromady pomocí operátoru +, nebo přes operátor |, který umožňuje vytváření alternativ. PyParsing je v podstatě kolekce parsovacích elementů sdružených a zkombinovaných dohromady. K takto vytvořeným parsovacím elementům můžeme přidružit akce, které se při analýze použijí.

### Implementace kalkulačky

Samotná implementace aritmetické kalkulačky zabrala pouze 50 řádků. Pro realizaci priority operátorů se v PyParsingu naskytly dvě možnosti, jak daný problém vyřešit. Buď pomocí běžných elementů pyparsingu nebo za pomocí dostupné modulové funkce `operatorPrecedence`, která priority operátoru řeší sama bez nutnosti našeho zásahu.

Při implementaci jsem vyhodnotil obě možnosti a dospěl k názoru, že i přes ulehčení práce přes modulovou funkci, se vyplatí použít pouze běžné elementy k dosáhnutí dostatečné rychlosti parsování. Verzi kalkulačky, ve které bylo využito funkce `operatorPrecedence`, totiž trvalo při větším zanořením výrazů (již 4 zanoření byly problémové) analyzování výrazů až stokrát větší dobu než ostatní zkoumané nástroje. Implementovaná funkce, realizující zpracování pomocí pyparsingu, pak sestává z definice parsovacích elementů a následné zavolání jediné funkce pro zahájení parsování. Ke každému elementu je přidružena akce buď provádějící aritmetickou operaci a nebo zpracování operandu.

## 3.4 Vlastní nástroj

Poslední zvažovanou možností je vytvoření vlastního lexikálního a syntaktického analyzátoru pro cílový překladač. Je zřejmé, že tento způsob zaručí nejvyšší rychlost zpracování na úkor doby nutné k jeho vývoji. Navrhovaný nástroj se skládá ze dvou částí – lexikálního analyzátoru a syntaktického analyzátoru.

### 3.4.1 Lexikální analyzátor

Lexikální analyzátor, který převádí vstupní soubor/řetězec na posloupnost lexikálních jednotek – tokenů – jsem modeloval jako jedinou funkci přijímající analyzovaný řetězec a vracející lexikální jednotku. Funkce byla implementována jako generátor pro zajištění nízké paměťové

náročnosti, kdy se postupně podle žádostí syntaktického analyzátoru poskytují tokeny. Samotná analýza je modelována jako konečný automat procházející vstupní řetězec po znacích a v případě rozpoznání tokenu je přes příkaz `yield` vrácen a funkce je tím dočasně pozastavena. Z hlediska udržovatelnosti však není tento přístup moc vhodný, jelikož rozšířením gramatiky je nutno přidat nové stavy a případně aktualizovat již vytvořené úseky kódu. Jakákoliv lexikální chyba je ošetřena vyvoláním vlastní výjimky, která je potom v syntaktickém analyzátoru odchycena a zpracována.

### 3.4.2 Syntaktický analyzátor

Hlavní předností vlastního syntaktického analyzátoru je možnost volby ze široké škály metod syntaktické analýzy, jako je například metoda rekursivního sestupu nebo precedenční analýza (viz 2.4). V případě jednoduché kalkulačky je nejlepší volbou právě metoda precedenční analýzy.

Metoda je založena na zásobníku a precedenční tabulce, kdy se gramatika skládá z terminálů a neterminálů uchovaných na zásobníku. Na počátku vložíme do zásobníku speciální terminál symbolizující konec analýzy a získáme první terminál od lexikálního analyzátoru. Poté se v opakované smyčce vykonává hlavní část analýzy, kdy se podle nejvrchnějšího terminálu na zásobníku (nemusí být nutně na vrcholu, jelikož jsou na zásobníku uloženy i neterminály) a terminálu přečteného na vstupu vybere z precedenční tabulky patřičná akce, která se v dané iteraci provede. Precedenční tabulka je implementována jako dvourozměrné pole indexované typy prvků zásobníku (terminálů a neterminálů).

První možnou akci je, že se vstupní terminál pouze vloží na zásobník, bez žádného dalšího zásahu. Alternativně se před vložením na zásobník najde nejvrchnější terminál a za něj se vloží zarážka, která poté slouží k vybírání variabilního počtu prvků zásobníku při redukčních pravidlech. Poslední akcí je redukční pravidlo, kdy se z vrcholu zásobníku odebere patřičný počet prvků a použije se odpovídající redukční pravidlo. Výsledek redukce je zpět uložen na zásobník jako neterminál. Cyklus se opakuje dokud se nejvrchnější terminál a terminál na vstupu nerovná ukončujícímu terminálu nebo nedojde k chybě. Výsledek analýzy se pak získá z vrcholu zásobníku.

## 3.5 Experimentální vyhodnocení rychlosti zkoumaných parserů

Pro potřeby experimentálního vyhodnocení byl vytvořen testovací skript, jehož cílem bylo měřit rychlost vybraných parserů na modelované gramatice jednoduché kalkulačky. Skript byl spuštěn se zapnutným garbage collectorem, z důvodu vysoké paměťové náročnosti (ergo mohl mírně ovlivnit výsledky experimentu), a měřil rychlost parsování modelované gramatiky na sadě různě velkých a zanořených aritmetických výrazů spuštěné tisíckrát, viz tabulka 3.1.

PLY je zástupcem klasického přístupu k analýze vstupního kódu podobně jako unixové nástroje Lex a Yacc. Současně se při vyhodnocení prokázalo, že je i nejrychlejším ze zkoumaných nástrojů, ale současně také nástrojem produkujícím parser zabírající největší počet řádků (mimo vlastního parseru). Postup při tvorbě analyzátoru je intuitivní a lexikální a syntaktická část analyzátoru je od sebe jasně oddělená a korespondence s klasickými unixovými nástroji Lex a Yacc je příjemná pro zkušené unixové programátory. PLY se díky své rychlosti i intuitivnosti psaní kódu jeví jako nejvhodnější volba pro tvorbu překladačů v Pythonu.

Naopak PyParsing je nejpomalejším ze zkoumaných nástrojů, nicméně umožňuje rychlou tvorbu a vyžaduje malé množství řádků kódu. Při testování parsování za pomocí modulové funkce `operatorPrecedence` zprostředkující precedenční analýzu operátorů došlo k výraznému zpomalení kódu a proto je vhodnější používat pouze základní jednoduché elementy při tvorbě parseru.

ANTLR se jeví jako vhodný přístup pro tvorbu rychlých a efektivních překladačů. S veškerými testovacími výrazy si poradil bez problémů, i s hlubším zanořením. Vstupem tohoto nástroje je gramatický popis syntaxe, což poskytuje intuitivní postup při tvorbě parserů. Výstupem generátoru jsou potom dva moduly, které se použijí v překladači, kdy samotná analýza vstupního kódu v samotné aplikaci zabere nejvýše deset řádků.

Podle očekávání je analyzátor šitý na míru gramatice nejlepší z hlediska rychlosti, ovšem na úkor velikosti kódu a době vývoje. Tento postup navíc vyžaduje vysokou míru invektivity a spoustu plánování a analýzy, což by za nás mohly dělat předchozí zkoumané nástroje. Využití vlastního analyzátoru pro tvorbu výsledného překladače jsem tímto zavrhl, jelikož pro gramatiku velikosti jazyka C by se výrazné zpomalení vývoje za cenu vyšší rychlosti dle mého názoru nevyplatilo.

Evaluací výsledků se potvrdil prvotní předpoklad, že vlastní parsovací nástroj je nejrychlejší ze zkoumaných možností. Dostupné nástroje jsou tedy pomalejší na úkor urychlení vývoje a generičnosti použití. Z těchto nástrojů je pak nejrychlejší klasický přístup pythonovského modulu PLY, který současně poskytuje intuitivní způsob popisu parseru. Z hlediska složitosti kódu a rychlosti vývoje jsou na tom dostupné nástroje podobně a nejedná se tedy o vlastnosti, které bychom měli při výběru nástroje zohledňovat. Všechny nástroje jsou intuitivně použitelné a dobře zdokumentované s kvalitní dostupnou literaturou. Důležitým aspektem bych však označil ještě verzi cílového kódu, kdy nástroj ANTLR neposkytuje cílový kód ve verzi Python 3, narozdíl od PLY a PyParsingu.

| **Velikost výrazu** [znaky] | **ANTLR** [s/cyklus] | **PyParsing** [s/cyklus] | **PLY** [s/cyklus] | **Vlastní nástroj** [s/cyklus] |
|---|---|---|---|---|
| 3 | 0.00012 | 0.00081 | 0.00165 | 0.00009 |
| 300 | 0.00696 | 0.01411 | 0.00514 | 0.00245 |
| 400 | 0.00972 | 0.02127 | 0.00614 | 0.00304 |
| 800 | 0.01939 | 0.04512 | 0.01208 | 0.00621 |
| 1200 | 0.03172 | 0.06570 | 0.01627 | 0.00965 |
| 20000 | 0.50598 | 0.97399 | 0.23571 | 0.15383 |
| 40000 | 1.02822 | 1.95895 | 0.47967 | 0.31881 |
| 60000 | 1.58434 | 2.95396 | 0.73639 | 0.49561 |
| 80000 | 2.14490 | 3.96229 | 1.00295 | 0.68467 |
| 100000 | 2.84151 | 5.09380 | 1.28975 | 0.88024 |

Tabulka 3.1: Srovnání rychlostí parsovacích nástrojů pomocí testovacího skriptu.

# Kapitola 4

# Návrh procesoru a instrukčních sad

Pro potřeby zadní části překladače je nutno specifikovat instrukční sadu cílového procesoru, ve které bude výstup překladu. Z programového hlediska nás především zajímá rozhraní, které nám poskytuje abstraktní pohled na konstrukci procesoru nazývané *ISA*[1]. ISA definuje základní soubor procesorových instrukcí, dostupné adresové módy, datové typy a uložiště, jako jsou sady registrů a paměťové buňky viditelné programátorovi, ale také dostupné periferie a hardwarové jednotky.

## 4.1 Instrukce

Účelem instrukce je specifikace prováděné operace a jejich operandů, které se se při provádění operace využijí, zahrnujíc argumenty, vstupy a výstupy. Operační kód (někdy taký *opcode*) i operandy jsou zpravidla reprezentovány bitově, částmi instrukce. Operandy pak zahrnují adresy do datových uložišť v paměti nebo procesoru. Alternativní součásti operací jsou někdy i implicitní operandy, které se před provedením operace musí nahrát do procesoru předem známe oblasti.

Jednou z používaných klasifikací je rozdělení na horizontální a vertikální instrukce [10], kdy horizontální instrukce jsou charakteristické dlouhým formátem, schopností vyjádření vysoké míry paralelismy a řídce zakódovanou řídicí informací. Oproti tomu vertikální instrukce mají krátké formáty s hustým kódováním řídicích informací a malou podporou paralelismu.

Instrukce můžeme dělit na aritmetické, logické, ukládací, nahrávací, instrukce pro změnu toku a další, podle jejich sémantiky. Instrukce využívají různé adresovací módy, které členíme na přímé, relativní, registrové a pseudo-přímé. Přímé adresování není vždy možné realizovat, pokud nemáme velmi limitovaný adresový prostor, abychom pokryli všechny možné adresy pomocí adresové části instrukce. Tento prostor je právě vždy přístupný pomocí registrového nebo registrově nepřímého adresování, za předpokladu, že velikost registru je minimálně stejně velká jako velikost adresy. Často používané pak může být relativní adresování, kdy využíváme malých konstant vměstnaných do instrukčního slova. Pseudo-přímé adresování využívá segmentového registru nebo vrchní bity programového čítače spojeného s přímou adresou adresové části instrukce.

Nejjednodušším způsobem adresování paměti je bezprostřední[2] adresování, kdy požadovaná data vložíme jako konstantu přímo do instrukčního slova. Při registrovém adresování značíme v operandu číslo registru, ve kterém se nacházejí data, u nepřímého registrového

[1] Instruction Set Architecture

[2] anglicky „*immediate*“

adresování určujeme číslo registru, ve kterém se nachází adresa požadovaných dat. Velice užitečné je pak adresování s posunutím[3], kdy v části instrukci máme index, který připočteme k adrese uložené v registru pro výpočet finální použité adresy. Jistou modifikací tohoto chování je výpočet adresy ze dvou registrů.

Zpracování operandů a bitová struktura instrukcí se liší podle použitých konvencí, které tuto formu definují. Jednou z nejstarších je použití jednoho a více *akumulátorů* pro uložení jednoho z operandů a výsledku. Druhý, explicitní, operand můžeme uložit buď v paměti nebo do některého z registrů. Použitím akumulátorové architektury dosáhneme úspory paměťového místa i výraznou redukci potřebných bitů pro zakódování instrukce. Další paměťově výhodnou architekturou je *zásobníková architektura*, kdy operace pracují s vrcholem zásobníku a výsledek operace uloží zpět na vrchol. Nejobecnější a současně i v dnešní době nejčastěji užívaná je *registrová architektura*. Instrukce zahrnují několik indexů registrů (podle počtu vstupujících operandů a výsledků), které se v operaci použijí. Existuje více typů registrových architektur. Nejběžnější z nich je *load-store*, kde musíme nejprve nahrát operandy do registrů pomocí instrukcí *load* a výsledek uložit přes instrukce typu *store*. Další typy registrových architektur rozlišují, zda-li se jeden nebo více operandů mohou nacházet v paměti – hovoříme tak o architekturách *register-memory* a *memory-memory*.

## 4.2 Návrh procesoru

Při návrhu procesoru se většinou snažíme přiblížit ke konkrétnímu programovacímu jazyku vyšší úrovně a optimalizovat pro něj výstup překladače. Tento přístup však v mnohých aspektech ztroskotává. Nikdy totiž nepoužíváme pouze jediný programovací jazyk pro psaní programů pro mikrokontrolery. Navíc se procesory vesměs používají pro specifické účely a nejsou vhodné pro běžné použití. V zásadě nezáleží na programovacím jazyce, ale na charakteristice algoritmu, který hodláme na procesoru implementovat, podle kterého bychom měli navrhovat, měnit a vybrat vhodný procesor. Hovoříme o tzv. *ASIP*[4] procesoru.

Proces návrhu procesoru začíná zachycením funkčních a nefunkčních požadavků. Funkčními požadavky chápeme předpokládané aplikace, které na procesoru pojedou a jeho operační prostředí. Jedná se o vesměs jednoduché požadavky. Dalším krokem je výčet operací nebo prototypových instrukcí, které se využijí pro námi známe algoritmy. Současně musíme definovat adresaci operandů a nejčastěji používané datové typy. Některé požadavky jsou závislé na operačním prostředí nebo paměťovém podsystému či vstupně výstupních zařízeních. Na pomezí můžeme považovat požadavky na počet hodinových cyklů, případně latenční dobu okolních částí systému. Mezi nefunkční požadavky řadíme cenu, počet pinů nebo spotřebu.

Návrh instrukčních sad a jejich kódování do binární formy často provádíme iterativně. Hledáme kompromis mezi horizontálním a vertikálním přístupem. Snažíme se o maximální paralelizaci a zachycení všech řídicích signálů v instrukčním slově, což však vede k dlouhým instrukčním slovům. V opačném přístupu kódujeme operace do co nejmenšího počtu bitů, což nás omezuje u použitelných kombinací prostředků. Vhodným kompromisem je použití instrukcí ve formě několika různých formátů s několika různými velikostmi. Součástí návrhu můžeme udělat profilování, díky kterému se dozvíme, které instrukce se používají nejčastěji. Jedním z problému, na které však můžeme narazit je příliš malý adresový prostor. Adresy zpravidla limitujeme primárně pro snížení velikosti instrukcí, velikosti registrů a počtu pinů.

[3] anglicky „*offset*“

[4] Application-specific instruction-set

## 4.3 Cílová architektura

Cílovou architekturou překladače bude jeden z jednodušších dostupných mikrokontrolérů. Hlavním účelem vytvořeného překladače je především podpora výuky, demonstrace a experimenty. Proto jsem jako cílovou architekturu zvolil FPGA projekt využívaný ve výuce v předmětu INP[5]. FPGA projekt obsahuje jednoduchý 16-bit procesor s registrovou architekturou, pamětí RAM pro uložení dat a programu, řadu podpůrných komponent a především jednoduchou instrukční sadu. Procesor lze ovládat mikrokontrolerem FITkitu za pomocí sběrnice SPI, ke které je procesor a paměť připojena prostřednictvím adresových dekodérů. Cílová architektura je flexibilní a lze ji dále rozšiřovat pomocí kódu v jazyce VHDL.

Paměť programu a dat je implementovaná jako společná. Je dvouportová s organizací 1024 šestnácti bitových slov. Komunikace s periférierniemi je umožňěna pomocí 16-ti bitového vstupně-výstupního portu, pomocí kterého je možné adresovat až 256 externích zařízení.

Základní instrukční sada je výpočetně úplná a sestává z instrukcí pro práci s pamětí a registry, skokových instrukcí, sčítání a práce s porty. Instrukcím jsou dostupné čtyři registry, jejichž počet je však možné v projektu rozšířit.

Tento velice jednoduchý procesor proto představuje řadu výzev. Především z důvodu omezené instrukční sady je pro podporu instrukcí z množiny gramatiky jazyka C zapotřebí více instrukcí, čímž naroste velikost cílového kódu. V kombinaci s omezeným počtem instrukcí programu je nutno věnovat výraznou pozornost optimalizacím výsledného kódu a vyhnout se zbytečnému generování kódu. Dalším omezením cílového procesoru je chybějící hardwarový zásobník, díky kterému je nutno volání podprogramu realizovat vlastními silami, například jeho simulací v paměti za pomocí ukazatelů.

[5] Návrh počítačových systému

# Kapitola 5

# Implementace překladače jazyka C v prostředí Python

Překladač je napsán kompletně v programovacím jazyce Python verze 3.x. parametrizovatelný pomocí argumentů z příkazové řádky. Překladač je tvořen skupinou spolupracujících modulů a několika extérních knihoven. Vstupem je seznam parametrů zadaných na příkazové řádce a posloupnost zdrojových souborů s programy v jazyce C, které skript transformuje do assembleru cílového procesoru. Samotný překlad sestává z šesti kroků. Každý vstupní soubor je nejprve předzpracován pomocí *Preprocesoru jazyka C* (alternativně může být vstupem skriptu již zpracovaný soubor s příponou `.i`), kdy se provede zpracování na textové úrovni a dojde k rozvinutí maker a podmíněných překladů. Výstup je následně zpracován do formy abstraktního syntaktického stromu (dále jen AST) přes lexikální a syntaktickou analýzu. Při zvolení argumentu optimalizace je výsledný AST optimalizován a převeden do vnitřní reprezentace instrukcí mezikódu. Nad těmito instrukcemi můžeme provést další vlnu optimalizací. Z optimalizovaného mezikódu získáme jeho transformací program v cílovém kódu.

## 5.1 Preprocesor jazyka C

Samotnému vytvoření abstraktního syntaktického stromu a jeho následné kompilaci do cílového kódu, předchází předzpracování vstupního souboru tzv. preprocesorem. Preprocesor jazyka C je implementován jako samostatný modul, jehož vstupem je soubor se zdrojovým kódem jazyka C a jeho výstupem je zpracovaný a vyčištěný kód připravený pro lexikální a syntaktickou analýzu. Výsledný preprocesor byl vytvořen v souladu se standardy jazyka C99 [1] a s podobným chováním jako má preprocesor referenčního UNIXového překladače GCC [13]. Hlavním podnětem pro vytvoření vlastního preprocesoru byla nedostupnost jiného vytvořeného nástroje v Pythonu dodržující nejnovější normu a současně získání nezávislosti od programů třetí stran, což by nastalo v případě použití preprocesoru nástroje GCC.

Předzpracování probíhá ve dvou logických celcích. Nejprve dochází k počátečnímu předzpracování kódu, který se poté v druhém kroku rozbije na posloupnost řádků, které se sekvenčně prochází.

V počáteční fázi se nejprve spojí všechny rozbité řádky (rozbitými řádky myslíme řetězce s escapovaným koncem řádku). Ze zdrojového textu jsou následně nahrazeny všechny řádkové a blokové komentáře mezerou a rozvinuty tzv. *trigraphy*. *Trigraphy* [13] jsou sekvence tří znaků, které se používají jako substituce znaků důležitých pro syntaxi jazyka C, viz tabulka 5.1. Jako sekundární efekt počáteční fáze se do zpracovaného textu přidají direk-

| **Trigraph:** | ??( | ??) | ??< | ??> | ??= | ??/ | ?¿ | ??! | ??- |
|---|---|---|---|---|---|---|---|---|---|
| **Nahrazen za:** | [ | ] | { | } | # | \ | ^ | \| | ~ |

Tabulka 5.1: Překladová tabulka trigraphů na znaky [13]

tiva s informacemi o řádkování, které se využijí při výpisu informacích o vzniklých chybách v přední části překladače.

Předzpracovaný text je poté rozbit na seznam řádků, které jsou sekvenčně procházeny a prohledány na výskyt direktiv preprocesoru, které jsou pří úspěšném vyhledání provedeny. V opačném případě dojde k rozvinutí všech maker v právě zpracovávaném řádků a následně připojení rozvinutého textu do cílového obsahu. Každá direktiva začíná pevným znakem `#`, který může být předcházen libovolným počtem bilých znaků. Jiné znaky jsou považovány za chybu. Každý příkaz direktivy se skládá z kličového slova, popisující operaci, a alternativně i jeho těla.

Základním příkazem preprocesoru je directiva `define` pro definování maker. Makro je definováno svým identifikátorem, počtem argumentů a tělem. Identifikátor může obsahovat alfanumerické znaky, čárku, závorky a podtržítko, musí však vždy začínat písmenem (stejně jako každý validní identifikátor jazyka C). Jako identifikátor makra lze tedy uvést libovolné kličové slovo jazyka C, s výjimkou již definovaného makra `defined`, které se používá v podmíněném překladu pro testování existence makra. V těle makra mohou být obsaženy libovolné znaky a definují posloupnost znaků, kterými je identifikátor nahrazen po rozvinutí parametrů maker.

Při definování makra se nejprve zpracuje jeho tělo, které se normalizuje pro sémantickou jednoznačnost a poté se společně s počtem parametru uloží do slovníku maker, kde je identifikátor makra kličem ve slovníku. Rozlišujeme dva typy direktiv – funkční a jednoduché. Rozdíl mezi nimi je v počtu argumentů, kdy funkčnímu makru musí následovat závorka s reálnými parametry i při nulovém počtu argumentů maker. Jinak se makro nerozvine. Při redefinici makra platí, že pro sémanticky odlišné makra[1] je vypsáno varování o jeho redefinici. Každé makro lze oddefinovat pomocí direktivy `undef` přijímající identifikátor makra jako tělo příkazu.

Standard C99 představil makra s variabilním počtem argumentů, tzv. *variadická makra*. Variadické makro může obsahovat libovolný počet pozičních argumentů a jako poslední argument se uvede řetězec „...“ reprezentující variabilní počet parametrů. Veškeré realné parametry jsou poté při rozvinutí spojeny do jediného řetězce, oddělené čárkou, a vloženy na poziční místo. K variabilnímu parametru lze v těle makra přistupovat přes klíčové slovo `__VA_ARGS_` (toto klíčové slovo se nesmí vyskytovat jinde než ve variadických makrec) nebo pomocí vlastního identifikátoru specifikovaného před řetězcem „...“.

Úseky kódu, především definice funkcí a konstant, se mohou nacházet v tzv. *hlavičkových souborech*, které se k zpracovávanému souboru připojují přes direktivu `include`. Direktiva musí nejprve lokalizovat, kde v adresářové struktůře se hlavičkový soubor nachází, prohledáním stanovených adresářů (adresař s překládaným souborem, adresáře specifikované uživatelem z příkazové řádky a systémové adresáře). Obsah nalezeného hlavičkového souboru je načten, zpracován počátečním předzpracováním, rozbit na seznam řádku a připojen k seznamu řádků čekajících na zpracování. Jako optimalizace připojování hlavičkových souborů je zaznamenávání již připojených souborů z kompilovaného zdrojového kódu, tedy není

[1] dva makra mají stejnou sémantiku, pokud se rovnají jejich těla i pozice argumentů v těle

třeba používát direktivy `pragma` nebo podmíněného překladu.

Obsah překládaného souboru může být řízen pomocí podmíněného překladu. O podmíněný překlad se starají direktiva `if`, `ifdef`, `ifndef`, `elif`, `endif` a `else`. Základní sémantikou těchto příkazů je vyhodnocení podmínky, která pokud je pravdivá, tak zahrneme kód těla do překladu. Tělem direktivy `if` je sémanticky platná podmínka z běžné syntaxe C rozšířené o funkci `defined`, která přijímá jako argument identifikátor makra a vrací booleovskou hodnotu, o tom, zda je makro daného identifikátoru definované a současně jsou všechny ostatní identifikátory nahrazeny nulou. Oproti tomu direktivy `ifdef` a `ifndef` přijímají v těle identifikátor makra a následující řádky zahrne pouze, pokud je makro o daném identifikátoru definované a nebo ne. Alternativní podmínky v případě neúspěchu lze stanovit pomocí direktivy `elif` a `else`.

Implementačně je každá podmínka normalizovaná (funkce `defined` a další operátory jsou převedeny do pythonovské notace) a vyhodnocena. Pro každý blok `if-endif` je vyhrazeno, podle zanoření, místo v zásobníku booleovských příznaků, které určují poslední evaluovanou podmínku a příznak, zda se mají zahazovat další řádky, nebo nikoliv. Podle kombinací těchto dvou parametrů a typů direktiv se řádky na vstupu připojují k výslednému výstupu nebo se zahodí.

Posledními podporovanými typy direktiv jsou direktiva `error` a `warning` pro vypsání chyb a varování na standardní chybový výstup a v neposlední řadě direktiva s informacemi o řádkování tvořená uvnitř zejména uvnitř preprocesoru. Tato direktiva pouze aktualizuje informace v preprocesoru o číslu řádku a souboru, ze kterého jsou řádky zpracovávány. Norma C99 navíc definuje direktivu pragma, u které nespecifikuje konkrétní význam a proto nebyla do preprocesoru zahrnuta.

Pokud se na řádku nejedná o direktivu preprocesoru, je text rozdělen pomocí generátoru na makro-podezřelé[2] a zbytky textu, které jsou postupně připojovány k výslednému řádku. Generátor funguje jako konečný automat o pěti stavech – žádné makro nerozpoznáno (tj. základní stav), detekován podezřelý na makro, uvnitř stringu, hledání levé závorky a načtení argumentů makra. Generátor postupně prochází řádek po znacích, dokud nenarazích na znak z množiny povolených znaků identifikátoru. V tom případě přejde do dalšího stavu a ještě předtím připojí dosud načtené znaky k výsledku. Následně načítá povolené znaky dokud nerozpozná identifikátor, který je vyhledán v slovníku definovaných maker a v případě úspěchu jsou podle počtu jeho argumentů načteny reálné argumenty z textu. Poté je možno rozvinout tělo makra, které je znovu zkontrolováno pomocí tohoto generátoru, a se všemi makry rozvinutými, je připojeno k výsledku a automat se navrací do počátečního stavu. V případě detekce uvozovek v počátečním stavu, jsou veškeré znaky v uvozovkách nekontrolovány, a nakonec připojeny k výsledku. Jedním z parametrů generátoru je taky seznam již rozvinutých maker, které již nebude generátor kontrolovat. Tento seznam slouží pro ochranu před zacyklením v případě vlastní nebo nevlastní rekurze, kdy by se makra cyklicky v těle expandovalo až do vypršení paměti. Každé makro je tedy ve svém těle rozvinuto pouze jednou. Výsledný obsah poté můžeme získat pomocí členské funkce a dále zpracovat a případně uložit. Argumenty funkčních maker se mohou vyskytovat na několika řádcích. Pokud při analýze procházením po znacích narazíme na konec řádku a aktuálním stavem konečného automatu je zpracovávání argumentů, pak připojíme znaky následujícího řádku k právě zpracovávanému textu.

[2] rozpoznaný identifikátor, který může být makrem

## 5.2 Přední část překladače

Kód zpracovaný preprocesorem jazyka C je nutno následně převést do vhodné vnitřní reprezentace pro následné převedení do kódu cílové architektury. Při tomto procesu probíhá lexikální a syntaktická validace, zda se jedná o korektně zapsaný program v jazyce C. Navíc jsou provedeny sémantické operace jako je kontrola typů operandů a řízení toků. Hovoříme o jisté formě tzv. *syntaxí řízeného překladu*.

V popředí překladače stojí syntaktický analyzátor, který od lexikálního analyzátoru postupně požaduje rozpoznané tokeny. V běžném syntaxí řízeném překladu by společně se syntaktickými pravidly byly prováděny i sémantické akce, ty jsou však odděleny a prováděny až poté co proběhne syntaktická analýza. Za vnitřní reprezentaci programu v překladači byl zvolen abstraktní syntaktický strom. AST je vytvořen syntaktickým analyzátorem při formování validních vět náležících gramatice jazyka C z tokenů. Vytvořený AST je pak zpracován sémantickou analýzou, jejimž výstupem je korektní reprezentace programu připravená pro překlad.

### 5.2.1 Abstraktní syntaktický strom

Abstraktní syntaktický strom je grafová struktura, v niž uzly reprezentují jednotlivé elementy modelované gramatiky a hrany vyjadřují vztahy a přislušnost mezi nimi. K uzlům mohou být navíc přiřazeny atributy, vyjadřující dodatečné informace, jako jsou typy, jména nebo hodnoty uzlů. Využitím AST zachytíme podstatnou strukturu vstupu a současně zanedbáme zbytečné syntaktické detaily, jako jsou například středníky ukončující příkazy nebo složené závorky pro blokové příkazy.

Existuje několik způsobů pro vnitřní implementaci AST. Při výběru vhodného kandidáta se zaměříme na základní idiomy dle typů programovacích jazyků [7]. Díky objektové orientaci Pythonu můžeme použít Objektově orientovaný idiom, který implementuje uzly stromu jako instance tříd, kdy všechny třídy uzlů jsou přímo nebo nepřímo odvozeny od základní abstraktní bázové třídy. Alternativně bychom mohli použít idiom pro funkcionální jazyky s využitím datatypů a nebo složitějších schémat.

Pro modelování konkrétních tříd uzlů pak musíme zvolit vhodnou míru abstrakce. Příkladem mohou být binární operace, kdy každou binární operaci lze modelovat jako samostatný uzel, nebo je sdružit do jednoho globálního typu uzlu reprezentujícího binární operace a rozlišit jednotlivé typy vnitřním atributem objektu.

Implementace stromu vychází ze struktury popsané v technické zprávě [6] zaměřené pro jazyk C++. V Pythonu je tato struktura realizována pomocí dědičnosti. Definujeme bázovou třídu `Node`, která reprezentuje jeden uzel AST, od kterého dědí další blíže specifikované uzly gramatiky jazyka C, jako mohou být například výrazy, bloky, volání funkcí, deklarace apod. Každý uzel je reprezentován svým typem, a svými potomky. Tento způsob je zahrnut uvnitř použitého extérního modulu pro syntaktickou analýzu `pycparser`.

Existuje několik možností, jak procházet AST, lišících se převážně v umístění v objektu [7, 6]. Vše záleží na míře abstrakce jednotlivých uzlů a požadované nezávislosti struktury nad funkčností vykonávanou při průchodu stromem. Jednou z možností průchodu stromu je zahrnutí prováděné funkce jako členské metody uzlu. Průchod stromem se započne voláním funkce kořenového uzlu, který postupně volá metody svých potomků a vykonává tak implementovaný algoritmus. Tímto však získáme výraznou závislost algoritmu na uzlech a špatnou udržovatelnost kódu. Alternativiní možností by bylo implementovat jedinou funkci, která by uvnitř kontrolovala, jaký typ uzlu má na vstupu, tento způsob by však byl

příliš složitý a neefektivní. Daleko vhodnější je návrhový vzor `Visitor` [5].

Tento návrhový vzor umožňuje odstinit realizaci algoritmu od definované struktury a s využitím možností jazyka Python navíc modifikovat strom a přidávat uzlům nové atributy. Jeho implementace nevychází z jeho běžné struktury, ale je využito předností jazyka Pythonu. Pro každý typ uzlu je definována v objektu `Visitor` vykonávaná funkce obsahující jméno uzlu jako suffix, která pobírá jako argument uzel stromu. Nad těmito funkcemi mimoto existuje generická průchodová metoda, která vybere konkrétní metodu, podle třídy přijmutého uzlu. To vše je možné pomocí získání informace o názvu třídy objektu spojené s dynamickým vyhledáním definovaných metod objektu v jeho slovníku obsahující členské proměnné a metody (i metoda je v Pythonu proměnnou a tudiž může být uvnitř kontejneru). V těle konkrétních funkcí pak specifikujeme prováděný algoritmus a případné volání návštěvnické metody nad dětmi daného uzlu. V základu tento vzor pouze prochází stromem a neprovádí žádnou modifikaci struktury stromu. Při některých algoritmech (např. optimalizace stromu, viz 5.3.1) však požadujeme redukci nebo vkládání nových uzlů, kdy musíme vzor modifikovat, aby každý průchod uzlem vracel hodnotu uzlu.

### 5.2.2 Syntaktická analýza

AST je vytvořen pomocí spolupráce lexikální a syntaktické analýzy implementované v extérním modulu **pycparser**, založeném na PLY (viz 3.1), který byl zvolen jako vhodný nástroj pro implementaci přední části překladače. Vstupem syntaktického analyzátoru je kód zpracovaný preprocesorem napsaný v jazyce C a výstupem je pak abstraktní syntaktický strom. Jediným omezením tohoto modulu je zpracování chyb, kdy při první syntaktického chybě nenastane zotavení zpět do procesu syntaktické analýzy, ale tvorba stromu se ukončí.

Vzhledem k tomu byl modul upraven pro potřeby překladače a sémantické analýzy. První změnou bylo vlastní zpracování chybových hlášení za pomoci modulu `ErrorManager`, seskupujího všechny proběhlé varování a chyby vzniklé v modulech podílejících se na činnosti překladu. Další změnou modulu je podpora značkování listu definicí pořadovým číslem. V základu totiž **pycparser** jsou všechny definice vyskytující se na jednom řádku, tedy například „`int i, j, k;`“, transformovány do posloupnosti definicí vyskytujících se na samotných řádcích. Současně však, při definici strukturovaných typů a enumerátorů, lze i definovat proměnné těchto typů, čímž se pro každou takovou proměnnou vytvoří uzel s celou definicí jejího typu a vytvořením proměnné tohoto typu a tím znesnadní sémantickou analýzu redefinice struktur. Tyto rozbalené definice jsou tedy označeny postupně generovaným pořadovým číslem a umožní tím detekovat redefinici typu.

### 5.2.3 Sémantická analýza

Vytvořený AST je zpracován pomocí sémantických kontrol, které zahrnují akce typového systému, implicitní přetypování, kontrolu toku, vytvoření tabulky symbolů a další. Sémanticky správný kód, ve formě AST, poté může být převeden do vnitřní reprezentace ve formě tří-adresného kódu. Veškeré sémantické analýzy jsou realizovány pomocí návrhového vzoru `Visitor`. Sémantickou analýzu realizují tři třídy `Visitor` pro tvorbu tabulky symbolů, typovou analýzu a kontrolu toků.

#### Tvorba tabulky symbolů

V prvním kroku sémantické analýzy se pokusí překladač vytvořit tabulku symbolů naplněnou v programu definovanými proměnnými. Tabulka symbolů je objektem skládajícím se

z jednotlivých rozsahů. Rozsah je vytvořen při vstupu do každého složeného příkazu a definice těla funkce. Implementačně je uvnitř tabulky pouze odkaz na kořenový rozsah (globální rozsah souboru) a poté odkaz na rozsah, ve kterém se právě nacházíme podle pozice kódu při průchodu stromem. Tabulka symbolů musí umožňovat vstup do rozsahu přiřazenému konkrétnímu bloku kódu a současně jeho opuštění. Každý rozsah uchovává odkaz na svého otce, tedy nadřazený rozsah, slovník deklarovaných proměnných, jmen enumerátorů a poté také definovaných uživatelských typů.

Proměnná je definovaná rodinou tříd odvozených od bázové třídy `Variable`, která obsahuje základní informace o proměnné, jako je její identifikátor, základní typ (zda se jedná o pole, ukazatel, strukturu, unii nebo běžné základní typy), konkrétní podtyp (např. ukazatel na `integer`), inicializační hodnotu a v rámci programového a optimalizačního hlediska také informace o živosti a konkrétní hodnotě. Specifičtější třídy obsahují konkrétnější informace o proměnných a dodatečné pomocné metody. Pole uchovává informace o svých dimenzích, struktury a unie o svých členech, ukazatel pak odkazovanou paměť. Pro každou proměnnou uchováváme v tabulce také její bližší specifikace, jako jsou informace o umístění (`extern` a `static` a jiných kvalifikátorech, jako jsou například příznaky `const` pro vynucení konstantní hodnoty proměnné, tedy vynucení nemožnosti zápisu do proměnné a nebo `volatile` pro zakázání optimalizací nad danou proměnnou).

Tabulka symbolů musí umožňovat přidávat a získávat proměnné zadaného idenfitikátoru z aktuálního rozsahu nebo se dotazovat na existenci konkrétního identifikátoru. V jazyce C vzniká hierachický strom rozsahů, ve kterém jsou definované proměnné. Pokud není proměnná nalezena v aktuálním rozsahu, přesune se hledání o úroveň výš do otcovského rozsahu, až dokud se nenarazí na globální rozsah souboru. Tímto je umožňěno v programech v jazyce C tzv. překrývání proměnných, kdy se proměnná o daném identifikátoru může nacházet v tabulce symbolů na několika místech. Výjimkou jsou návěští, která jsou vždy definovány v globálním rozsahu.

Při průchodu stromem nás zajímají veškeré uzly obsahující identifikátory a deklarace. Uzel třídy `Decl`, reprezentující veškeré deklarace a definice v programu, je rozcestníkem, který volá konkrétní metody realizující přidávání proměnných daných typů do tabulky symbolů. V každém případě se vkládá do tabulky symbolů proměnná o specifikovaném identifikátoru a současně se uvnitř tabulky provede š, zda nedochází k redefinici proměnné v daném rozsahu. Při deklaraci pole, se z řetězu uzlů získají informace o jeho dimenzích a jeho typ, u struktur a unií, které se liší pouze svým základním typem, se pak získají jejich členové, kteří se přidají k proměnné do tabulky.

U funkcí se navíc rozlišuje, zda jde o jejich deklaraci nebo jejich definici, podle čehož se provádí kontrola. Byla-li funkce deklarovaná a probíhá její definice, pak jsou zkontrolovány, zda odpovídají parametry a návratová hodnota prototypu a právě definované funkce. Pokud se snažíme znovu definovat nebo znovu deklarovat funkci stejného jména, je zahlášená chyba, jelikož tak dochází k redefinici proměnné. U funkcní navíc probíhá, v případě přítomnosti specifikátoru `inline`, kontrola, zda funkce vůbec může být inlinována. *Inline* funkce jsou speciálním typem funkcí s implementačně-specifickým chováním. Jejich hlavním cílem je zajištění efektivnějšího kódu, zejména rozbalením těla funkce v místě volání. Vytvořený překladač `inline` funkce rozbaluje, ale pouze za předpokladu, že neobsahují rekursivní volání sama sebe.

Speciální analýzu si vyžádává funkce pojmenovaná `main`. Tyto funkce jsou ve většině C programech počátečním místem provádění programu a mají specifickou hlavičku. `Main` funkce vždy vrací celočiselný výsledek a obsahuje buď žádné argumenty nebo celočíselný počet argumentů z příkazové řádky a pole ukazatelů na typ `char` uchovávající textovou

podobu argumentů. Toto je při této sémantické analýze kontrolováno.

Při průchodu uzlu reprezentujícího identifikátor nebo referenci struktury je provedena kontrola definovanosti daného identifikátoru, jména struktury a jejího členu. Výjimkou při kontrole jsou návěští u příkazů `goto`, které v tuto počáteční chvíli nemohou být zkontrolovány, jelikož mohou odkazovat na návěští, definována později v kódu.

Výčtový typ, enumerátor, je uvnitř kódu kompatibilní s celočíselným typem `integer`. Norma nespecifikuje žádnou konkrétní kontrolu proměnných typu `enum`. Uvnitř překladače jsou tedy všechny proměnné výčtového typu převedeny na celočíselné konstanty a je nad nimi prováděna stejná typová kontrola jak nad běžnými proměnnými.

### Kontrola programového toku

Další krokem sémantické analýzy je kontrola toku, která ověřuje správnou pozici příkazů `return`, `case`, `default`, `break` a `continue`. Při každé návštěvě cyklu, příkazu `switch` či funkce se poznačí pozice v kódu. Zkontrolujeme, zda se nacházíme ve správné pozici v kódu, pokud nikoliv pak zahlásíme chybu.

Dle normy C99 [1] nemusí obsahovat funkce příkaz `return`, nicméně při této analýze je navíc provedeno ověření veškerých toků informací ve funkci pro zamezení možných chyb vzniklých nedbalostí programátora. Pro každý podmíněný skok je do seznamu nevyhodnocených cest přidán patřičný počet cest, podle počtu podmínek. Pokud pak narazíme na příkaz `return` je podle aktuálního rozsahu odečten patřičný počet cest (záleží na zanoření a větvi v podmínkách). Na konci funkce je poté seznam vyhodnocen a v případě, že se v kódu nachází cesty, které nemají příkaz `return` je vypsáno varování, že ne všechny cesty funkce mají návratový příkaz.

Jako vedlejší účinek této sémantické analýzy se u všech vytvořených návěští v programu analyzuje, zda mají v kódu korespondující příkaz `goto` a tato informace se zaznamená do tabulky symbolů. Nyní je již možno tuto kontrolu provést, jelikož všechny návětší již byla definována během tvorby tabulky symbolů. Tato analýza nemá žádný význam pro sémantiku programu, ale využije se v dalších volitelných fázích překladu při optimalizacích.

### Typová analýza

Součástí typové analýzy je kontrola typů výrazů, identifikátorů, počtů dimenzí polí, realných parametrů a současně i implicitní konverze podle typového systému jazyka C [1]. Programovací jazyk C obsahuje sérii již definovaných typů a současně umožňuje vytvářet programátorské vlastní typy. Mezi základní typy patří typy z rodiny `integer` (`int`, `short int`, `long int`, ...) a z rodiny realných čísel (`double`, `float`, ...), které souhrně nazýváme jako *aritmetické*. Aritmetické typy společně s ukazateli nazýváme typy *skalárními*. Při typové analýze se každý element jazyka, který může mít typ, označí konkrétním typem, který vznikne vyhodnocením jeho operandů – synovských uzlů.

Při každé deklaraci se v případě přítomnosti inicializačního výrazu kontroluje, zda odpovídá typu deklarované proměnné a v případě, že se jedná o aritmetický typ, provede se implicitní konverze inicializačního výrazu. V případě polí se navíc kontroluje typ výrazu při definování dimenzí, které musí být typu z rodiny `integer`. Potom při samotné referenci pole musí mít použitý index celočíselný typ a současně se i testuje, v prípadě, že meze polí šly vyčíslit, zda se neindexuje mimo pole.

U binárních operací se kontrolují oba operandy a podle možností se provede implicitní přetypování operandů a získá se výsledný typ. Podobně je to i u unárních operací, kde se

kontroluje jejich jediný operand. Pro binární i unární operace platí následující sémantika a typové omezení:

- **Bitové operace** (&, ^, |, ~, «, ») – oba operandy musí mít typ z rodiny `integer`, výsledek bitové operace je typu `integer`

- **Modulo** (%) – levý operand operace modulo musí být typem aritmetickým a pravý operand musí být `integer`, neprobíhají zde žádné implicitní typové konverze. Výsledek je stejného typu jako je levý operand.

- **Logické spojky** (&&, ||) – oba operandy musí být skalárního typu. Výsledek operace je typu `integer` nabývající hodnoty 0 nebo 1.

- **Relační operátory rovná se a nerovná se** (==, !=) – oba operandy porovnání musí být buď aritmetického typu, oba ukazateli a nebo jeden z nich ukazatelem a druhý tzv. *null pointer*, který je definován jako nulová konstanta nebo nula přetypovaná na prázdný ukazatel (typ `void*`)[3]. Výsledek operace je typu `integer` nabývající hodnoty 0 nebo 1.

- **Ostatní relační operátory** (>, >=, <, <=, !) – oba operandy relačních operací musí být aritmetického typu nebo oba ukazateli. Výsledek operace je typu `integer` nabývající hodnoty 0 nebo 1.

- **Aditivní operace** (+, ++) – oba operandy musí být aritmetického typu nebo první ukazatelem a druhý z rodiny `integer`. Výsledkem operace je první z typů.

- **Substraktivní operace** (−, −−) – oba operandy musí být aritmetického typu nebo první ukazatelem a druhý z rodiny `integer`. Výsledkem operace je první z typů. Alternativně mohou být oba operandy ukazateli, kdy je výsledek operace v typu implementačně specifickém reprezetující rozdíl mezi dvěmi ukazateli.

- **Multiplikativní operace** (*, /) – oba operandy musí být aritmetického typu.

Při operacích zahrnujících aritmetické typy dochází k implicitnímu přetypování operandů a získání výsledného typu operace následujícím způsobem: pokud oba operandy mají stejný datový typ, nedochází k žádné typové konverzi a výsledkem je společný datový typ. Pokud jeden z operandů je typu `long double`, druhý operand je přetypován a výsledek je rovněž typu `long double`. Pokud je jeden z operandů typu `double`, postupujeme stejně jako v předchozím případě, pouze s typem `double`. Jinak provedeme stejnou kontrolu pro typ `float`. V opačném případě se jedná o celočíselné typy. Pokud oba operandy jsou se znaménkem, jako výsledek vypočítáme největší znaménkový typ `integer`, který pojme oba rozsahy a ten bude výsledkem. Pokud jsou oba bezznaménkové, pak vypočítáme největší bezznaménkový

[3] „`(void *) 0`“ nebo „`0`“

celočíselný typ. Je-li jeden operand bezznaménkového typu a pojme rozsah druhého operandu, pak bude výsledkem tento typ. Stejná kontrola následuje pro znaménkový typ. Pokud nebyla ani jedna podmínka splněna, pak získáme největší možný bezznaménkový typ, který by pojal oba rozsahy.

Konverzi typů zajišťuje v jazyce C tzv. `cast` operátor, reprezentovaný názvem typu v závorkách před přetypovaným výrazem. Rozlišujeme dva typy konverzí – explicitní (ty uvedené programátorem) a implicitní (ty, které provádí sémantický systém překladače). Z implementačního hlediska znamená implicitní konverze obalení výrazu uzlem `Cast`. Při sémantické analýze se navíc u tohoto uzlu kontroluje, zda je vůbec možno přetypovat daný typ. Mezi přetypovatelné typy pak patří aritmetické typy a ukazatele.

Operatory přiřazení, inkrement a dekrement vyžadují na levé straně operátoru tzv. *lvalue*, což je prvek obsahující adresu. Tímto prvkem tedy může být proměnná nebo člen struktury či pole, u kterých není specifikován modifikátor `const` a tedy je možno do něj zapisovat Konkrétní typy operandů přiřazení, se poté liší podle příslušné operace přiřazení respektující stejné podmínky jako jejich binární ekvivalenty.

## 5.3 Optimalizace

Vzhledem k tomu že cílová architektura obsahuje jednoduchý 16-bit procesor s omezenou instrukční sadou a pamětí, je vhodné mezikód i výsledný kód v průběhu překladu optimalizovat (především redukovat počet instrukcí). Protože je nutné některé binární instrukce řešit výraznějším počtem instrukcí cílového procesoru, nabývá výstupní program na velikosti. Mezi dostupné optimalizace tedy patří zejména rozbalení konstant a vyhodnotitelných výrazů, odstranění redundantního kódu a nebo také transformace binárních operací na jejich sémantické ekvivalenty vyžadující méně instrukcí.

Z hlediska rozsahu lze obecně optimalizace rozdělit na globální a lokální. Globální optimalizace jsou prováděny nad celým vstupním programem v C, zatímco lokální jsou provedeny v rámci konkrétních bloků kódu. Z implementačního hlediska pak můžeme rozlišovat optimalizace podle struktury, nad kterou jsou vykonávány – optimalizace AST a optimalizace vnitřního tří-adresného mezikódu.

### 5.3.1 Optimalizace AST

Veškeré optimalizace AST jsou realizovány upraveným návrhovým vzorem `Visitor` [5], podle struktury popsané v podkapitole 5.2.1, který byl modifikován, aby umožňoval navrácení hodnoty a tím i modifikaci stromu. Mezi možné optimalizace stromu patří analýza konstantích výrazů a jejich rozbalení v stromu a transformace aritmetických výrazů aplikováním matematických vztahů a zákonů.

#### Rozbalení vyhodnotitelných výrazů

První z implementovaných optimalizací je rozbalení a analýza konstantních výrazů. Všechny operandy binárních a unárních operací jsou testovány na konstanty. Pokud jsou všechny operandy operace konstantami, je možné tento výraz vyhodnotit již v době překladu a nahradit jedinou konstantou vzniklou evaluací operace. Postupným průchodem stromu a vyhodnocováním uzlů do hloubky tak dochází k výrazné redukci počtu uzlů a tím i počtu potřebných provedených operací ve výsledném kódu.

Součástí rozbalení konstant je i analýza vyhodnotitelných podmínek řídicích příkazů a eliminace mrtvého kódu. Je-li v podmínce konstantní výraz, který lze vyhodnotit jako pravdivý nebo nepravdivý, pak můžeme modifikovat strukturu stromu odstraněním některých uzlů. Cykly s nepravdivou podmínkou na začátku (`For` a `While`) nebudou nikdy vykonány a tudiž mohou být odstraněny z výsledného kódu. Oproti tomu cyklus `DoWhile` je při nepravdivé podmínce nahrazen jedinou posloupností příkazů svého těla, která by při spuštění kódu byla vykonána pouze jednou. Naopak při pravdivé podmínce je generováno varování o možném nekonečném cyklu. Tento úkaz nelze považovat za chybu, ani nijak odstrantit z kódu, jelikož nekonečný cyklus může být zastaven z extérního zdroje pomocí přerušení, změnou volatilních hodnot, nebo příkazem `break` uvnitř těla.

U podmíněných příkazů analyzujeme všechny větve. V případě pravdivé podmínky zahrneme pouze její tělo, jinak odstraníme všechny nepravdivé části podmíněných větví, dokud nenarazíme na vždy pravdivou nebo nevydnotitelnou podmínku, kterou musíme v kódu ponechat. Podobný postup je použit pro příkaz `switch` v případě, že se v jeho podmínce nachází konstantní výraz. Procházením jeho větví `case` pak porovnáváme získanou hodnotu s hodnotou větve a podle rovnosti zahrneme částí kódu do výsledku.

U vyhodnotitelných podmínek však může nastat problém s chybným předpokladem, že tělo nebude vykonáno. Uvnitř těla se může nacházet návěští pro v budoucnu proveditelný skok a tímto ho nemůžeme odstranit realizováním optimalizace. V průběhu analýzy toků se při průchodu `goto` uzlem označí v tabulce symbolů u korespondujícího návěští, že se v těle nachází příslušná skoková instrukce a nemůžeme ji proto odstranit.

### Transformace aritmetických výrazů

Další sérií optimalizací je transformace aritmetických operací za pomocí aplikace vybraných matematických výrazů a zákonů [2]. Aplikací těchto vztahů můžeme odstranit části výrazů nebo je transformovat na jiný výraz o stejné sémantice, avšak vyžadující menší počet operací. Mezi tyto vztahy pak patří například zákony Boolovy algebry. Jinou transformací je pak nahrazení násobení a dělení mocninami čísla dvě za bitový posun doleva a nebo doprava o odmocninu z dělitele/násobitele. Posuny jsou ve výsledku instrukčně méně náročnou operací a výrazně se projeví na výsledném počtu instrukcí assembleru. Tato rodina optimalizací také zahrnuje různé běžné vztahy, které nebudeme podrobně zmiňovat, jedná se například o násobení jedničkou, přičítání nuly a podobné neutrální operace, které mohou být nahrazeny jediným operandem.

| | | |
|---|---|---|
| **Distributivita:** | $(x \vee y) \wedge (x \vee z) = x \vee (y \wedge z)$ | $(x \wedge y) \vee (x \wedge z) = x \wedge (y \vee z)$ |
| **Neutralita:** | $x \vee 0 = x$ | $x \wedge 1 = x$ |
| **Komplementarita:** | $x \vee \neg x = 1$ | $x \wedge \neg x = 0$ |
| **Absorbce:** | $x \vee (x \wedge y) = x$ | $x \wedge (x \vee y) = x$ |
| **Agresivita:** | $x \vee 0 = x$ | $x \wedge 1 = x$ |
| **Idempotence:** | $x \vee x = x$ | $x \wedge x = x$ |
| **Absorbce negace:** | $x \vee (\neg x \wedge y) = x \vee y$ | $x \wedge (\neg x \vee y) = x \wedge y$ |
| **Dvojitá negace:** | $\neg(\neg x) = x$ | |
| **De Morganovy zákony:** | $\neg x \wedge \neg y = \neg(x \vee y)$ | $\neg x \vee \neg y = \neg(x \wedge y)$ |
| **Transformace násobení:** | $x * 2^y = x << y$ | |
| **Transformace dělení:** | $x/2^y = x >> y$ | |

Tabulka 5.2: Vybrané matematické vlastnosti, vztahy a zákony aplikované při optimalizaci kódu [2]

### 5.3.2 Optimalizace Mezikódu

Samotnému procesu optimalizace mezikódu předchází statická analýza vstupního kódu – vytvoření a naplnění tabulek základních bloků. Základním blokem chápeme posloupnost příkazů, které se provedou vždy všechny nebo žádný z nich. Z pohledu kódu je blok určen tzv. *vedoucím*, což je první příkaz základního bloku, a posloupností příkazů až do výskytu dalšího vedoucího. Vedoucím je pak úplně první příkaz programu, návěští a příkaz následovaný za skokem `goto`. Tabulka základních bloků sestává z řádků, které zahrnují příslušící instrukci a informace o aktuálních stavech proměnných a jejich další použití v kódu. Z pohledu stavu je proměnná živá nebo mrtvá, kde úmrtím proměnné rozumíme, že v blízké době bude její hodnota přepsána přiřazovacím příkazem.

Pro potřeby této struktury je nutno rozšířit uložené informace v tabulce symbolů o stav proměnné a její další použití. Před samotným vyplněním provedeme inicializaci tabulky, nastavením dalších použití proměnných na `None`, stav programátorských proměnných (definované uživatelem) na živé a stav dočasných proměnných na mrtvé. Řadek tabulky pak vyplňujeme pozpátku od její poslední příslušící instrukce. Pro každou instrukci provedeme kontrolu použitých operandů, z tabulky získáme jejich stavy a další použití. Následně aktualizujeme informace v tabulce symbolů. Operandy, do kterých ukládáme výsledek, jsou operací usmrceny a další použití nastaveno na `None`, zbylé operandy jsou naopak živé. Další použití operandů je nastaveno na číslo řádku aktuálně zpracovávané instrukce.

#### Identifikace duplikátních výrazů

V kódu se mohou vyskytovat opakující se operace, které při vysoké náročnosti na počet řádků, mohou výrazně zvětšit velikost výsledného kódu. V rámci základního bloku analyzujeme všechny operace. Každou operaci, která se v kódu vyskytuje více než jednou a této operaci nepředchází přiřazovací instrukce, která by přepsala její předchozí hodnotu, vyjmeme, vytvoříme pro ni nové paměťové místo a výpočet operace vložíme na začátek základního bloku (pokud je první instrukcí návěští skoku, pak za toto návěští). Všechny opakující se operace takto výjmeme a nahradíme proměnné získávající tento výsledek. Základní blok takto opakovaně analyzujeme dokud jsou vyjímatelné operace.

#### Odstranění redundantního kódu

Jednoduchá optimalizace odstraňující instrukce a části kódu, které nemají žádný význam a provádějí redundantní akce. V tuto chvíli předpokládáme, že byly provedeny předchozí optimalizace a byl odstraněn maximální počet konstantních výrazů. První redukcí je odstranění přiřazení sama sebe ($x = x$), který může v kódu vzniknout právě v důsledku předchozích optimalizací a nemá pro výsledek žádný význam. Dalšími redundantními instrukcemi jsou instrukce mezikódu `NoOperation` a návěští bez příslušné skokové instrukce.

## 5.4 Generování mezikódu

Jako mezikód překladače jsem zvolil tří-adresný kód, při kterém instrukce sestává z prováděné operace, a třech adres – levého a pravého operandu a adresy výsledku. Z této generické instrukce jsou poté odvozeny konkrétní operační instrukce, mezi které patří binární a unární instrukce, podmíněný a nepodmíněný skok, volání a návrat z podprogramu, instrukce přesunu a práce s ukazateli a indexací polí.

Generování mezikódu je opět prováděno pomocí návrhového vzoru `Visitor`. Pro každý navštívený uzel jsou generovány odpovídající instrukce mezikódu. Pro ukládání dočasných proměnných v programu, tedy neprogramátorských proměnných, které se budou nacházet při provádění v registrech nebo na zásobníku, se generuji v tabulce symbolů dočasná uložiště podle vyhodnoceného typu uzlu během sémantické typové analýzy. Každý uzel představující výraz je označen náležející proměnnou v tabulce symbolů a odkaz na tuto proměnnou je uložen v generované instrukci.

Během průchodu a generování instrukcí je nutno uchovat několik návěští v členských proměnných a to především návěští vzniklé při generování řídících příkazů. Je nutno uchovat návěští pro příkazy `break` a `continue`, aby bylo možné vygenerovat příkaz nepodmíněného skoku při průchodu příslušným uzlem. Současně je nutno si poznačit řídící proměnnou `switch` příkazů, aby bylo možné generovat mezikód pro porovnání hodnoty `case` s touto řídící proměnnou a konec podmíněného příkazu `if` pro jeho další podmínky. Jelikož jazyk C umožňuje libovolné zanoření řídících příkazů musí být tyto informace uloženy v struktuře typu zásobník a podle zanoření použít příslušné návěští nebo proměnnou.

### 5.4.1 Zpracování polí

Jazyk C umožňuje vytvoření multidimenzionálních polí, v cílovém assembleru však není žádná specifická podpora pro tyto typy a je tedy nutné veškeré vícedimenzionální informace převést do jediného rozsahu, jedné dimenze. Pro tyto potřeby existuje mapovací funkce (5.1), která zajišťuje konverzi vícedimenzionálního prostoru do jediného rozsahu. Pole budeme ukládat v paměti po řádcích, tedy rychlost změny indexů se zvyšuje z leva do prava. Samotné mapování pak probíhá podle jednoduché myšlenky – pro každou dimenzi si vypočítáme, kolik prvků ve vyšších dimenzích je již obsazených, násobením velikostí vyšších dimenzí indexem, ke kterému v poli právě přistupujeme. Výpočet provedeme pro každý index a každou dimenzi a následnou sumou všech násobků získáme náležející index z jediné dimenze (5.2).

$$B[i_1, i_2, \ldots i_n] \rightarrow A[f(i_1, i_2, \ldots i_n)] \tag{5.1}$$

$$f(i_1, i_2, \ldots i_n) = \sum_{k=1}^{n} (D_k * i_k) \tag{5.2}$$

$$D_k = \prod_{l=k+1}^{n} (D_l) \tag{5.3}$$

Tento zmiňovaný způsob však platí jen pro indexaci pole pouze za pomocí konstant, to tedy znamená, že vytvoření jednodimenzionálního indexu je realizovatelné již během překladu. V případě použití výrazů nebo proměnných pro indexaci pole proto musíme implementovat mapovací funkci sami instrukcemi mezikódu. Pro minimalizaci výsledného kódu současně ošetříme speciální případ jednodimenzionálního pole, kdy nemá smysl použít jakékoliv mapování. V tomto případě bude operandem indexace proměnná z tabulky symbolů (indexace identifikátorem) nebo dočasná proměnná uchovávající výsledek výpočtu výrazu použitého pro indexaci.

Při multidimenzionální indexaci pak vypočteme výsledný index za pomoci instrukcí násobení a sčítání. Každý výraz (proměnná z tabulky symbolů, konstanta, výsledek výpočtu výrazu) přislušící dimenze indexu vynásobíme koeficientem dimenze (5.3) a připočteme k dočasné proměnné, ve které uchováváme výsledek mapovací funkce. Tímto vytvoříme řetězec

násobení a sčítání postupující přesně podle vzorce pro vytvoření mapovací funkce (5.2), kdy dočasná proměnná uchovávající výsledek se stane operandem pro indexaci pole.

Po získání operandu s indexem následuje vytvoření instrukce indexace. Celkově rozlišujeme tři typy indexací polí, podle jejich výskytu v kódu – přímé přiřazení hodnoty na daném indexu pole do tzv. *lvalue*[4], přiřazení hodnoty z paměti nebo registru na daný index pole a nebo uložení hodnoty z pole do dočasné proměnné. První dva výskyty jsou generovány z členského uzlu AST třídy `Assignment` (přiřazení) a třetí typ je využit ve všech ostatních binárních a unárních výrazech.

### 5.4.2 Volání funkcí

S příchodem normy C99 byly do jazyka C zavedeny tzv. *inline funkce* [1]. Inline funkce se zpravidla sémanticky chová tak, jako kdyby v místě jejího volání bylo vloženo její tělo s definovanými proměnnými, kterým byly přiřazeny hodnoty získané z funkčních argumentů. Toto chování však není vynuceno normou a překladače nejsou povinny rozbalení těla funkce provést. Navržený překladač všechny inline funkce rozbalí, pokud neobsahují volání sama sebe, ergo vlastní rekursi. Při definování inline funkce získáme posloupnost instrukcí jejího těla, kterou přiřadíme proměnné funkce v tabulce symbolu. V případě běžných funkcí se funkční tělo umístí do kódu s návěštím pro skok. Současně se, v případě, že poslední instrukcí není příkaz `return`, vygeneruje prázdný návrat z funkce (`return ;`). Instrukce těla `main` funkce, se speciální sémantikou, jsou umístěny vždy na záčátek seznamu instrukcí, aby po zahájení programu byly vykonány jako první. Uživatelské funkce se tak nachází na konci.

Při volání funkce se z tabulky symbolu se v případě, že se nejedná o inline funkci, vygeneruje podle návratového typu proměnné instrukce buď volání procedury nebo volání funkce. V opačném případě zahrneme instrukce funkčního těla do výsledného kódu. Problémem je však překrývání proměnných a současně také několikanásobné volání funkce v programu, proto musíme nejprve zpracovat a transformovat všechny jeho instrukce, tak aby mohly existovat v jediném rozsahu s unikátními jmény. To provedeme přidáním prefixu proměnným podle jména volané funkce a pořadového čísla jejího volání.

Speciální transformace je provedena nad instrukcí mezikódu `RoutineReturn` reprezentující návrat z podprogramu. V kódu z nefunkčního rozsahu se tato instrukce nesmí vyskytovat a i v případě funkce by tato instrukce neodpovídala sémantice návratu z rutiny. Proto, podobně jako u polí, rozlišujeme mezi třemi možnými vyskyty funkce – v prvním případě, může existovat funkce volně v kódu, aniž bychom vyzvedli a uložili její návratovou hodnotu do určité proměnné, alternativně můžeme její návratovou hodnotu zachytit do proměnné v paměti a nebo se může volání funkce vyskytovat v binárním nebo unárním výrazu (v tomto případu, zachytíme výsledek do dočasné proměnné).

Ve všech případech musíme vygenerovat instrukci nepodmíněného skoku na návěští na konci těla funkce symbolizující ukončení provádění podprogramu. V posledních dvou zmiňovaných připadech však ještě musíme nejprve zachytit návratovou hodnotu do dočasné proměnné nebo do paměti pomocí instrukce `CopyOperation` pro kopii mezi pamětí, registry nebo pamětí a registrem.

Před připojením seznamu transformovaných instrukcí zpracujeme reálné parametery volání inline funkce. Extrahujeme z uzlu reálné argumenty a pro každý parametr inline funkce pak vytvoříme instrukci mezikódu pro přesun dat (`CopyOperation`) mezi reálnou hodnotou parametru a přiděleným paměťovým místem parametru funkce. Prakticky bychom mohli

[4] l-value rozumíme buňku, proměnnou, mající přiřazenou adresu a s umožněným zápisem

zanedbat nepoužité parametry v těle inline funkce a negenerovat pro ně inicializační instrukce, nicméně výsledný program by nemusel být semanticky ekvivalentní s výsledným assemblerem. Proto generujeme tyto instrukce pro všechny parametry funkce.

### 5.4.3 Strukturované a ukazatelové typy

Podobně jako u polí a funkcí, i u ukazatelů rozlišujeme instrukce podle směru toku dat do/z ukazatelů a podle typu druhého účastnícího se operandu (dočasná proměnná v registrech nebo zásobníku versus programátorská proměnná v paměti). Současně existují dva typy unárních operací spojených s ukazatelovými typy – operátor dereference `*` pro získání hodnoty uložené na adrese obsaženou ukazatelem a operátor reference `&` pro získání adresy operandu. Ve spojení se směrem toku pak můžeme rozdělit ukazatelové instrukce typu `AddressPointer` na kopírování do ukazatele, kopírování z ukazatele do paměti a z ukazatele do registru a nahrání adresy do registru nebo paměti. Před samotným vytvořením ukazatelové instrukce a jejím následným připojením je však nutné získat úroveň dereference, jelikož operátory dereference mohou být do sebe libovolněkrát vnořené.

U strukturovaných typů (struktur a unií) je v případě reference jejich členů vytvořena nová proměnná reprezentující tento člen (za předpokladu, že již nebyla vytvořena jinou referencí). Jméno této proměnné je vytvořeno spojením identifikátoru struktury společně s názvem odkazovaného člena.

## 5.5 Generování cílového kódu

Poslední fází překladu je generování cílového kódu v jazyce Assembler. V této fázi se postupně generuje výstupní soubor z posloupnosti instrukcí volitelně optimalizovaného mezikódu. Jsou dva možné způsoby, jak generovat cílový kód, tzv. *slepé* a *kontextové generování*.

První z nich pro každou instrukci mezikódu, podle typu operandů, generuje šablonově posloupnost instrukcí assembleru. Registry jsou operacím přiřazovány cyklicky rovnoměrně, za předpokladu, že se nejedná o mezivýsledek, který je pak uložený v některém z registrů nebo na zásobníku, který musíme pak lokalizovat.

Alternativou tomuto generování je kontextové generování, které nepřijímá pouhou posloupnost instrukcí, ale vyžaduje vytvořenou tabulku základních bloků, podle kterých generuje cílový kód. Při tomto generování je kladen důraz na uchování proměnných v registrech co nejdelší dobu, za předpokladu, že budou v blízké době znovu použity. Tím jsou ušetřeny zbytečné instrukce přesunů mezi pamětí a registry. Kontextové generování vychází z postupů vyučovaných v předmětu IFJ[5] a uvedených ve zdroji [3]. Oba způsoby byly implementovány a lze je v překladači zvolit pomocí parametrů.

Procesu generování cílového kódu předchází normalizování tabulky symbolů do jediného rozsahu. V rámci sémantické analýzy bylo nutné členit definované proměnné do patřičných rozsahů podle zanoření, jelikož proměnné o stejném identifikátoru mohly být v rámci dvou rozsahů překryty a bez problému existovat. Generovaný kód assembleru však existuje pouze v jediném rozsahu a proto je nutné naplněnou tabulku do něj převést přejmenováním duplikátů na unikátní jména.

Oba způsoby generování vyžadují několik pomocných strukturu. První z nich, využívaná pouze pro kontextové generování, je tabulka základních bloků naplněná informacemi o stavu proměnných (živá vs. mrtvá proměnná) a také o dalším použití proměnných v rámci

[5] Formální jazyky a překladače

základního bloku. Další strukturou je tabulka registrů, ve které se nachází informace o stavu registrů – zda je registr volný, reservovaný nebo obsazen a v případě obsazení je zde uvedena uložená proměnná. Hlavní metodou této tabulky je získání registru využitého v konkrétní operaci. Implementace této metody se pak liší podle použitého způsobu generování – buď se registr získá cyklicky nebo pomocí složitějšího kontextového algoritmu. Poslední pomocnou strukturou je tabulka adres, ve které jsou uloženy detailní informace o uložení proměnných – zda se nachází proměnná v paměti, nikde nebo v registru a v kterém registru, nebo na které adrese se nachází.

### 5.5.1 Porovnání slepého a kontextového generování

Ve výchozím nastavení probíhá generování cílového kódu pomocí jednoduchých šablon, bez žádné znalosti kontextu. Jedná se o tzv. *slepé generování*. Při tomto způsobu se generují zbytečné informace přesunů mezi pamětí a registry a především instrukce spojené s jejich režií. To jsou například uložení dočasných výsledků do paměti do simulovaného zásobníku, pro uchování mezivýsledků, které budou použity později. Každé takové uložení a získání hodnoty ze zásobníku vyžaduje zhruba čtyři instrukce. Při slepém generování tak získáme mnoho řádků, které by mohly být odstraněny.

Odpovědí na neefektivní generování je *kontextové generování*, vycházející z postupů uvedených v [3]. Samotnému kontextovému generování předchází vytvoření a naplnění tabulky základních bloků. Po provedení statické analýzy můžeme generovat cílový kód. Hlavní rozdíl od slepého generování je ve způsobu získání registru pro operand v operaci.

Při slepém generování se přidělovaly registry cyklicky podle dostupnosti. Zde však postupujeme odlišně. Pokud proměnná, pro kterou hledáme registr pro uložení, se již v registru nachází a nemusí být nahrána z paměti, vrátíme tento registr. Jinak pokud existuje volný registr, tak vrátíme první z volných. Pokud není žádný registr volný, pak musíme využit některý ze zaplněných. Zde přichází na řadu tabulka základních bloků a výsledky předchozí statické analýzy. Vybereme registr v němž je proměnná, která je použitá nejpozdější, nebo je mrtvá. U každé proměnné v základním bloku máme informace o jejím dalším použití, vrátíme tedy registr té proměnné, která má nejvyšší číslo řádku, na které bude použita. Tímto udržujeme proměnné, které budou v blízké době použity v registrech a nemusíme tak generovat instrukce přesunu do paměti nebo uložení mezivýsledků na zásobník.

Předtím než poskytneme registr pro nahrání nového operandu musíme uložit předchozí obsah registru. Generujeme tak buď instrukci pro uložení do paměti nebo uložení na zásobník. Aktualizujeme tabulku registrů a tabulku adres podle změněných údajů. U instrukcí, které mohou uložit výsledek do stejného registru, jako je jeden z operandů (např. sčítání, odčítání a bitové posuny), použijeme pro výsledek tento registr, čímž ušetříme jeden registr.

### 5.5.2 Základní specifikace výstupního souboru

Výstupní soubor assembleru lze rozčlenit do několika částí. Hlavní smyčka programu je určená tělem funkce `main`, která je reprezentována počátečním návěštím a speciální funkční sémantikou. Příkazy návratu z rutiny jsou totiž překládány na instrukci zastavení provádění **halt**. Za hlavní částí kódu pak následují těla uživatelských funkcí, definované proměnné a systémově definované proměnné. Jelikož cílový procesor podporuje pouze jeden typ dat – celočíselných int – jsou všechny proměnné definovány pomocí assemblerovské direktivy `db`. Při definici struktur jsou pak definovány všechny její členské proměnné s prefixem určeným názvem proměnné struktury. Podobně se při definování pole určí počáteční návěští a poté jsou podle jeho velikosti použity instrukce `db`. Na konci paměti se pak nachází tři pomocné

systémové proměnné – `ph` pomocná proměnná pro práci s několikanásobnou dereferencí ukazatelů, `sp` ukazatel na vrcholek simulovaného zásobníku a pak `stack`, jež reprezentuje místo v paměti pro simulaci zásobníku. Cílová architektura v základu neobsahuje hardwarový zásobník, proto pro realizaci složitějších částí jazyka C (funkce) bylo nutné tento zásobník simulovat v paměti za pomocí ukazatele.

Inicializační hodnoty proměnných jsou již zpracovány během tvorby instrukcí mezikódu. U polí je implementováno pouze základní inicializování přes výčet hodnot, zbytek hodnot je inicializován na nulu. U struktur můžeme inicializovat konkrétní členy struktury. V překladači není přímo implementováno chování proměnné s nedefinovanou hodnotou. Všechny proměnné jsou totiž v případě chybějící inicializace nastaveny na nulovou hodnotu.

### 5.5.3 Implementace aritmetických a relačních instrukcí

Mezi jedno z omezení cílové architektury (viz podkapitola 4.3) patří právě minimální instrukční sada poskytující v základní verzi pouze sčítání a přesun dvojkového doplňku hodnoty uložené v registru. Při implementaci širokého repertoáru operací jazyka C je tedy nutno improvizovat a použit alternativních algoritmů pro jejich implementaci.

U všech instrukcí (binárních, unárních) musíme vždy ve všech případech načíst hodnoty všech operandů do registrů a získat registr pro uložení výsledku. Z hlediska lokací a typů operandů rozlišujeme čtyři možnosti načítání operandů. Pokud je operandem konstanta, provedeme nahrání její hexadecimální hodnoty (s kterou pracuje výstupní assembler) pomocí assemblerovské instrukce `mload`. Dalšími případy jsou neprogramátorské proměnné uchovávající dočasné výsledky, které se mohou nacházet buď v registru, který pro operand lokalizujeme, nebo (například v případě výsledku ternárního operátoru), se mohou nacházet na zásobníku (v našem případě simulovaném zásobníku). Posledním typem jsou programátorské proměnné, které načteme z paměti.

Při sčítání si vystačíme pouze s instrukcí `add`, u odčítání musíme nejprve provést negaci operandu na jeho dvojkový doplněk a následně provést součet operandů.

Sémantika logických spojek konjunkce „a“ (`&&`) a disjunkce „nebo“ (`||`) vrací jako výsledek první z operací logickou pravdu, pokud oba operandy jsou pravdivé, druhá z operací, pokud alespoň jeden z operandu je pravdivý. Při implementaci konjunkce vycházíme z předpokladu, že výsledek je nepravdivý. Poté testujeme oba operandy na nulu (tedy na nepravdu), a v případě úspěchu podmínky skočíme na návěští reprezentující konec instrukce. Pokud není ani jeden z podmíněných skoků proveden, což znamená, že oba operandy byly pravdivé, provedeme inkrementaci výsledku o jedna a tím získáme jako výsledek pravdu. U logické disjunkce postupujeme podle stejné logiky, s tím rozdílem, že na počátku předpokládáme, že výsledek je pravdivý a oba operandy testujeme na nenulovou hodnotu pro potvrzení našeho předpokladu. Pokud nebyl náš předpoklad potvrzen, výsledek dekrementujeme o jedna a tím získáme nulovou hodnotu jako logickou nepravdu.

U relačních operací, rovná se, nerovná se, větší, větší rovno, menší a menší rovno využíváme substrakce dvou vstupujících operandů a následné porovnání výsledku na nulu, kladné či záporné číslo. I u těchto operací vycházíme z předpokladu o výsledku, který podle úspěšných podmínek upravíme na pravdu nebo případně nepravdu. U čistého porovnání dvou hodnot po odečtení operandů testujeme výsledek na nulu. Pokud je nulou tak prvotní přepoklad platí (nepravda pro nerovná se a pravda pro rovná se), v opačném případě výsledek inkrementujeme nebo dekrementujeme, tedy negujeme. U zbylých čtyřech relačních operátoru předpokládáme na začátku kódu instrukce výsledek jako pravdivý a po odečtení operandů testujeme v případě operace menší než pomocí instrukce `brneg` (výsledek odečtení

je negativní, ergo je první operand menší) a v případě operace větší než pomocí instrukce `brpos` (výsledek odečtení je positivní, ergo je první operand větší). V případě nesplnění podmínek skoku je primární předpokládaný výsledek dekrementován na nulu.

Jednou z problémových partií aritmetických operací jsou bitové operace. Cílová instrukční sada v základní verzi neobsahuje žádné instrukce pro manipulaci s konkrétními bity. Bitové instrukce je tedy nutno implementovat pomocí počítaných cyklů s pevným počtem iterací podle počtu bitů v datovém typu operandu (zde se jedná o celočíselnou aritmetiku integerů, tedy se jedná o 16 bitů).

Všechny operace vycházejí z obecného algoritmu 1. Posun bitů v operandu o jedno místo doleva je realizován nasobením operandu dvěma, tedy sečtením operandu sama se sebou a jeho následné uložení stejného uložiště. Poté podle konkrétní operace provedeme testování operandu, zda je záporný, nulový nebo kladný. Záporné operandy jsou reprezentovány jedničkou na nejlevějším bitů v operandu, testováním na zápornost operandu tedy testujeme výskyt jedničky na nejlevějším bitu operandu. Posouváním operandu doleva postupně otestujeme všechny bity operandů a tvoříme výsledek přičítáním jedničky k výsledku a jeho posouvání doleva, podle vyhodnocených podmínek pro jednotlivé operace, kdy konjunkci náleží podmínka (5.5), disjunkci (5.6) a inklusivní disjunkci (5.4) (funkce právě jeden z bitů je jedničkou).

$$(A < 0 \wedge B \geq 0) \vee (A \geq 0 \wedge B < 0) \tag{5.4}$$

$$(A < 0) \wedge (B < 0) \tag{5.5}$$

$$(A < 0) \vee (B < 0) \tag{5.6}$$

**Algoritmus 1:** Generický algoritmus pro implementaci binárních bitových operací za použití sčítání a skoků

**Input**: Operand $a$ a $b$, operace $op$, kde $op \in (|, \&, \hat{})$
**Output**: Výsledek operace $result = a\ op\ b$, kde $op \in (|, \&, \hat{})$

```
result = 0
  for i = 1 to pocet_bitu_operandu do
      result += result
      if vyhodnocení podmínek operace then
          result++
      end
      a += a
      b += b
  end for
```

Protože cílový procesor neobsahuje hardwarovou násobičku ani děličku, musíme operace násobení a dělení řešit softwarově pomocí alternativních algoritmů. Pro násobení využijeme algoritmus 2 skládající se pouze z jednoho cyklu, testování nejpravějšího bitu operandu a posunů doleva a doprava. Jelikož procesor neobsahuje ani testovaní konkrétních bitů, musíme testování nejpravějšího bitu realizovat jiným způsobem. Nejjednoduším postupem je odstranit nejpravější bit posunem doprava a následně se návratit o zpět o jeden bit doleva a odečíst od původního operandu. Pokud výsledek je nenulový, pak nejlevější bit byl jedničkou a podmínka tímto byla splněna. Takto procházíme první operand po bitech a přičítáme postupně posouvaný druhý operand k výsledku.

**Algoritmus 2:** Softwarové násobení dvou celočiselných operandů

**Input**: Činitelé $a$ a $b$
**Output**: Součin činitelů

```
result = 0
while b != 0 do
    if b % 2 == 1 then
        result += a
    end
    a «= 1
    b »= 1
end
```

Větší problém nastává se softwarovým dělením a operací modulo (zbytek po dělení). Existuje řada algoritmů pro jeho realizaci, nicméně jejich přepis do instrukcí assembleru sestává z velkého počtu instrukcí a proto bylo nutné vybrat algoritmus s optimálním výsledným počtem instrukcí. Algoritmy pro celočíselné dělení se znaménkem nejsou moc efektivní a současně pro ně neexistují dostatečné prostředky na cílové architektuře, proto je vhodnější použít dělení bez znaménka. Samotnému dělení předchází úprava operandů získáním jejich absolutní hodnoty, tedy odstranění znaménka.

Jako algoritmus dělení byla zvolena metoda *Shift double word divident method* [12] (viz algoritmus 3) pro výpočet podílu a zbytku po dělení. Tento algoritmus je plně realizovatelný s dostupnou instrukční sadou a současně je implementovatelný malým počtem instrukcí. Podle realizované operace (dělení nebo modulo) je pak podle znamének počátečních operandů upraven výsledek na záporný nebo kladný. Společně s počáteční a koncovou úpravou výsledků a operandů sestává podle zvolené operace při slepém generování cílový kód v assembleru ze 33 instrukcí u dělení a 37 u modula.

Samotný algoritmus dělení zarovná dělence do dvou slova (reprezentované dvěma proměnnými v registrech $(r, q)$, kdy v $r$ získáme ve výsledku zbytek po dělení a v $q$ podíl) a poté provádíme podle počtu bitů operandů v iteracích testování, odečtení a posunutí. Každou iterací tak získáme jeden bit výsledku.

**Algoritmus 3:** Softwarové dělení dvou bezznaménkových celočiselných operandů metodou *Shift double word dividend*

**Input**: Dělenec $x$ a dělitel $y$
**Output**: Podíl operandů $q$, zbytek po dělení $r$

```
r = 0, q = x, i = počet_bitů
while i != 0 do
    r += r + (q / 2^(pocet_bitu-1))
    q += q
    if r >= y then
        r -= y
        y += 1
    end
    i -= 1
end
```

### 5.5.4 Práce s poli v assembleru

Jak již bylo zmíněno v 5.5.2, pole je reprezentováno posloupností hodnot v paměti uspořádané do jediné dimenze. Práce s poli pak probíhá s využitím pomocné proměnné v paměti `pointer_helper` pro získání a nebo uložení hodnoty. Z generování mezikódu rozlišujeme tři typy směru toku mezi pamětí a registry (pole-register, pole-paměť, paměť-pole).

Ve všech případech se nejprve vypočítá a uloží index do registru. Následně se uloží do druhého registru počáteční adresa pole a provede se její součet s indexem vypočteným v předchozím kroku. Tím získáme výslednou adresu adresovaného prvku, kterou nahrajeme do pomocného ukazatele `pointer_helper`, který můžeme využít pro dočasnou práci s poli. V prvním případě, při toku paměť-pole, nahrajeme hodnotu proměnné z paměti do registru a následně přes pomocný ukazatel na patřičné místo v poli. V opačném případě, při získávání hodnoty z pole, nahrajeme přes ukazatel tuto hodnotu do registru. V těchto případech aktualizujeme pozici dat v tabulce adres.

### 5.5.5 Práce s ukazateli v assembleru

V paměti jsou reprezentovány ukazatele pomocí dvou bytů, podobně jako proměnné celočíselného typu. Uvnitř assembleru nedochází k vnitřní typové kontrole, o správné využití operací a operandu se tedy stará v počáteční fázi kompilace sémantická kontrola.

Pro práci s pamětí a registry jsem měl k dispozici pět instrukcí – nahrání konstanty `mload`, nahrání z adresy `dload`, nahrání z adresy uložené v paměti `iload` a komplementární operace k předchozím dvou `istore` a `dstore` pro uložení na dané adresy v paměti. Pro jednoúrovňové operace tyto instrukce bohatě stačí, pro nahrání adresy pomocí operace reference si vystačíme s uložením konstantní adresy návěští proměnné (které se poté v překladači assembleru přeloží na fyzickou adresu v paměti) `mload` a následné uložení této hodnoty z registru do paměti proměnné ukazatelového typu.

Podobně jednoduchá je i realizace jednoúrovňové dereference. Při uložení hodnoty z paměti do místa, kam ukazuje ukazatel, nejprve nahrajeme hodnotu pravého operandu do registru a následně tuto hodnotu nahrajeme do referencovaného místa. U komplementární operace načtení hodnoty se pomocí `iload` instrukce načte obsah ukazované paměti do dočasného registru, ve kterém se buď uchová nebo přesune do druhého účastnícího se operandu.

Problém nastává u vícenásobné dereference, kdy nám nestačí jediné paměťové místo a současně nám chybí instrukce pro práci s paměti podle adresy uložené v registru. Toto musíme řešit oklikou přes paměť, kdy se ke konci paměťového rozsahu adres alokuje jedna pomocná proměnná nazvaná jako `pointer_helper`. Nejprve provedeme úvodní dereferenci, kdy nahrajeme přes registr z počátečního ukazatele jeho hodnotu do pomocné proměnné a poté provedeme ještě tolik načtení z adresy uložené v pomocné paměti a následné uložení na tuto adresy podle úrovně dereference mínus dvě. Tímto následujeme řetěz dereferencí, než se dostaneme na jeho konec, kdy nám zbývá poslední jednoúrovňová dereference, kdy postupujeme stejně jako bylo zmíněno na začátku podkapitoly.

### 5.5.6 Implementace funkcí

Sémantika inline funkcí byla probrána v podkapitole 5.4.2, a v etapě generování cílového kódu nehraje žádnou roli. Funkce jsou reprezentovány svým tělem, které začíná návěštím pro skok na její začátek. Pozičně jsou pak umístěny ke konci assemblerovského kódu za tělem `main` funkce.

Volání a realizace funkcí, především vlastní a nevlastní rekurze, bylo největším výzvou celého projektu. Díky absenci hardwarového zásobníku bylo nutné provést jeho simulaci v paměti za pomoci dvou proměnných (začátek zásobníku a ukazatel do zásobníku, tzv. „stack pointer“) a ukazatelových operací.

Hlavním problémem je však rešení rekurze. Jednou z možností implementací funkcí by bylo úplně zakázat rekurzi. Alternativně lze definovat veškeré parametry a proměnné definované uvnitř funkčního těla do paměti. Při výskytu rekurze by však hrozilo přepsání již vypočtených hodnot v těle funkce a proto by bylo nutno všechny hodnoty proměnných, respektive registrů, uschovat v paměti na zásobníků. Z hlediska výsledné velikosti kódu a současně možnosti rekurze v programu jsem se proto rozhodl pro šetrnější řešení a veškeré funkční proměnné a parametry realizovat přes zásobník.

Všechny proměnné definované uvnitř těla funkce se tedy nachází na simulovaném zásobníku. V rámci překladače udržujeme vnitřně informaci o stavu zásobníku a v případě instrukcí pro práci s funkčními proměnnými získáme index proměnné uložené na zásobníku. Tedy jak hluboko se nachází a kterým číslem musíme zdrojovou adresu vystupujícího operandu upravit. Při neredukční práci se zásobníkem, tedy proměnnou ponecháme na svém místě, využijeme podobně jako u ukazatelů pomocného místa v paměti, do kterého uložíme výslednou adresu, která jest výsledkem odečtení indexu položky od aktuální adresy uložené v ukazateli na vrchol zásobníku. Podle vypočtené adresy poté buď uložíme nebo nahrajeme na/z patřičné/ho místo/a.

Překladač v rámci generování instrukcí mezikódu generuje instrukce pro volání procedury (funkce nevracející hodnotu) a funkce. Při generování cílové kódu probíhá režie spojená s voláním podprogramu identicky. Jediný rozdíl v instrukcích je v následném získání návratové hodnoty.

Při volání subrutiny se nejprve vypočítá návratová adresa a vloží se na vrchol zásobníku. K výpočtu návratové adresy můžeme přistoupit několika způsoby. Jednou z nich je použití aktuální adresy a vypočítat ji již v překladači nebo ji vypočítat v programu. Daleko jednodušší je však vytvořit nové návěští, které připojíme za všechny instrukce spojené s režií volání funkce, a adresu tohoto návěští vložit na zásobník.

Následuje zpracování skutečných parametrů předaných funkci, podle typu parametru se postupně uloží na zásobníku veškeré parametry. Posledním krokem je skok na začátek těla funkce. Za instrukcí skoku se poté v případě funkcí realizuje získání výsledku provedení funkce. Výsledek se buď uloží do paměti přiřazením do proměnné nebo alternativně v případě, že je funkce volána ve výrazu je ponechán v registru.

Návrat z podprogramu pomocí instrukce mezikódu `RoutineReturn` začíná uvolněním spotřebovaných prostředků alokovaných na simulovaném zásobníku. Toto je realizováno odečtením počtu proměnných definovaných v těle funkce. Ve skutečnosti se proměnné samozřejmě stále nacházejí v paměti a mohou být opět zpřístupněny. Odstraněním alokovaných proměnných ze zásobníku se dostane na vrchol návratová adresa, kterou načteme do registru a pomocí instrukce skoku podle adresy v registru provedeme návrat do místa volání funkce. V případě funkcí však před tímto skokem provedeme uložení návratové hodnoty na vrchol zásobníku. V prvotní implementaci však instrukce skoku podle obsahu registru nebyla zahrnuta a je proto nutno instrukční sadu o tutu operaci rozšířit.

### 5.5.7 Práce s porty

Součástí instrukční sady jsou také instrukce pro práci s porty specifické pro architekturu použitého FPGA procesoru. Samotný jazyk C neposkytuje žádné mechanismy pro práci

s porty, nebo externí pamětí, proto musíme tyto instrukce zpracovat samostatně.

Jednou z možností umožnění zápisu a čtení do/z portů by bylo rozšíření gramatiky jazyka C o nové syntaktické konstrukce specifické pro tuto komunikaci. Tímto bychom však ztratili nezávislost přední části překladače a v případě výměny nebo modifikace cílové architektury by bylo nutné veškeré moduly předělat. Pro zachování nezávislosti jednotlivých částí překladače je proto nutné sáhnout do prostředku poskytovaných samotným vstupním jazykem. Jinou možností se nabízí vyhrazení speciální funkce pro zápis a čtení z portu, avšak sémantika volání funkce neodpovídá zcela mechanismům práce s porty.

Jako cílové řešení jsem proto zvolil práci s porty stejně jako s poli. V případě, že požadujeme ve zdrojových souborech komunikaci s porty mikrokontroleru za pomocí instrukcí `inp` a `outp`, nadefinujeme v kódu externí pole integerů[6] s vyhrazeným jménem `port`. Toto řešení nikterak nezasahuje do vstupních a výstupních jazyků a přitom umožňuje práci s porty pomocí běžných instrukcí pro práci s polem.

Implementačně pak podobně jako u polí rozlišujeme jednotlivé toky dat (do portu, z portu do paměti, z portu do registeru) a pro čtení a zápis využíváme instrukcí `inp` a `outp`, které přijímají jako argumenty číslo portu a registr, ze kterého čteme nebo do kterého zapisujeme.

[6] `extern int port[256];`

# Kapitola 6

# Experimentální vyhodnocení a ověření navrženého překladače

Část testování překladače proběhla pomocí regresních testů sestávající z očekávaných výstupů porovnávaných se skutečnými výstupy. Velká čast však byla testována především pomocí visuální validace výsledků, vybrané úlohy pak byly nahrány do cílového procesoru a byla otestována jejich korektní funkce. Větší důraz byl však kladen na experimentování s mnou vytvořeným překladačem a nastavením různých kombinací parametrů překladače, jako jsou optimalizační techniky a podobně. Experimentováním a zkoumáním vlivu jednotlivých optimalizačních technik a parametrů na efektivitu generovaného kódu jsem ověřoval, zda vytvořený produkt splňuje počáteční stanovené cíle. Pro vyhodnocení jsem použil několik testovacích problemů, které budou podrobeny bližším experimentům – rozšíření procesoru, aplikace optimalizací a použití kontextového generování na místo slepého.

## 6.1 Rozšíření procesoru a jeho instrukční sady

Abychom mohli vyhodnotit vliv instrukční sady na velikost výsledného kódu, došlo k rozšíření cílového procesoru. Počet dostupných registrů byl zdvojnásoben na osm, byl přidán válcový posouvač implementující efektivnější operace posuvů a také hardwarový zásobník návratových adres umožňující přímou podporu volání a návratu z podprogramů. Přidané instrukce umožňují mnohem efektivnější implementaci volání funkcí a následný návrat a současně i dalších konstrukcí jazyka C jako je práce s ukazateli, poli a násobení.

Mezi nové instrukce pak patří práce s pamětí pomocí adresy uložené v registru (instrukce `irload` a `irstore`) usnadňující práci s ukazatelovými typy (i poli tedy). Repertoár aritmetických operací byl rozšířen o implementaci binárních operací `and`, `or` a `xor` a současně i všech typů binárních posuvů (aritmetických i logických).

Jak je známo, dostupný počet registrů může výrazně ovlivnit velikost výsledného kódu. Abychom mohli tuto skutečnost potvrdit, bylo zapotřebí rozšířit počet dostupných registrů. Pro adresování registru byly v prvotním návrhu instrukční sady vyhrazeny pouze dva bity (což dává možnost adresovat maximálně čtyři registry). Tento počet bitů však neumožňuje adresovat osm registrů a proto jsou registry sdruženy do dvou tzv. *banků*. Abychom nemuseli zvětšit šířku instrukčního slova a přesto měli možnost adresovat více registrů, byla přidána instrukce `swbank`, která mění zvolené banky pro všechny registrové části instrukční sady. Tato instrukce defacto určuje jak budou přidávány horní bity do těchto částí.

Možnost libovolně měnit instrukční sadu cílového procesoru v překladači prokázala fle-

xibilitu navrženého řešení. Řešení tohoto problému bylo dosaženo objektově. Využitím principů dědičnosti jsem vytvořil novou třídu pro rozšířený procesor dědící od původního procesoru. V nově vytvořené třídě pak stačilo pouze modifikovat metody vyžadující implementovat daný problém efektivnějším způsobem. Pomocí parametru lze snadno zvolit jednu z použitých architektur.

Rovněž ve spojení se zvýšením počtu registrů v procesoru a jejich sdružením do banků, byl i upraven algoritmus realizující kontextové generování. Při přidělování registrů se nejprve vybere seznam nejvhodnějších kandidátů (takové, které mají např. stejné pozdní použití). Každý z těchto kandidátu se poté ohodnotí číslem od nuly do tří, kdy v počátku je ohodnocen nulou a za každý momentálně nastavený bank pro jedno ze tří registrových míst instrukcí, který odpovídá banku registru, je zvýšeno ohodnocení o jedničku. Takto se vrátí první bank s nejlepším ohodnocením, který má také velkou pravděpodobnost, že bude v instrukci usazen na správné místo a nebude tak zapotřebí generovat instrukce pro přepnutí banku.

## 6.2 Průběh experimentování

Pro experimentální účely bylo vybráno pět demonstračních úloh. Omezení paměti a počtu instrukcí znemožnilo testovat složitější problémy, pro základní demonstrace nám tyto vstupní programy postačí. K procesoru jsou namapovány tři periférie – led dioda mapovaná na port číslo 1, LCD displej mapován na port číslo 2 a 3 a čítač. První dvě úlohy mají za úkol prověřit funkčnost těchto periferií. Další úlohy pak implementují metodu řazení posloupnosti X prvků *bubblesort*, test na dokonalé číslo (takové, které je součtem všech svých kladných dělitelů) a test základních aritmetických operací. Jejich kompletní zdrojový kód je uveden v příloze C. Získané výsledky shrnuje tabulka 6.1.

| **Úloha** | **Procesor INP [instrukce assembleru]** | | | |
|---|---|---|---|---|
| | **Slepé gen.** | | **Kontextové gen.** | |
| | **Bez opt.** | **S opt.** | **Bez opt.** | **S opt.** |
| `demo_diode_test.c` | 81 | 91 | 45 | 43 |
| `demo_lcd_test.c` | 207 | 207 | 133 | 133 |
| `demo_bubblesort.c` | 216 | 216 | 146 | 146 |
| `demo_operations.c` | 206 | 206 | 164 | 164 |
| `demo_perfect_number.c` | 219 | 193 | 157 | 125 |

| **Úloha** | **Procesor INP po rozšíření[instrukce assembleru]** | | | | | |
|---|---|---|---|---|---|---|
| | **Slepé gen.** | | | **Kontextové gen.** | | |
| | **Bez opt.** | **S opt.** | **Vynucení 4 registrů** | **Bez opt.** | **S opt.** | **Vynucení 4 registrů** |
| `demo_diode_test.c` | 97 | 112 | 91 | 54 | 51 | 43 |
| `demo_lcd_test.c` | 233 | 233 | 205 | 134 | 134 | 131 |
| `demo_bubblesort.c` | 239 | 239 | 209 | 150 | 150 | 139 |
| `demo_operations.c` | 195 | 195 | 158 | 134 | 134 | 116 |
| `demo_perfect_number.c` | 263 | 224 | 193 | 171 | 141 | 125 |

Tabulka 6.1: Vliv parametrů překladu na velikost výsledného kódu

## 6.3 Vyhodnocení testování a experimentů

V tabulce 6.1 lze sledovat vliv jednotlivých parametrů překladu na velikost výsledného kódu měřený v počtu instrukcí potřebných pro implementaci daného problému. Úlohy byly přeloženy vytvořeným překladačem a následně za pomocí grafického rozhraní programu QDevKit ve spojení s pluginem `cpuide`, pro překlad assembleru do cílového kódu, nahrány do projektu na FitKitu a visuálně verifikovány.

### 6.3.1 Vliv rozšíření procesoru

Předpokladem rozšíření procesoru bylo snížení velikosti cílového kódu. Současně zvýšení počtu registru umožňuje mít více dočasných uložišť a generovat tak méně instrukcí přesunů mezi pamětí a registry. Velikost instrukčního slova je však pevné délky (16-bit) a proto pro zdvojnásobení počtu registru bylo nutno zavést banky, mezi kterými můžeme měnit pomocí instrukce assembleru. Při běžném slepém generování sice dochází k redukci potřebných instrukcí assembleru, nicméně ji nahrazuje nutnost generovat instrukce pro změnu banku, čímž může docházet ke zvýšení velikosti výsledného kódu. Zvýšení počtu registrů procesoru proto nemusí být prioritní potřebou při rozšiřování procesoru s pevnou délkou instrukce. Při vynucení použití pouze čtyř registrů při slepém generování došlo k očekávanému snížení počtu operací.

Největší vliv na velikost výsledného kódu má hardwarová podpora aritmetických instrukcí, což lze především vidět na příkladech položených na těchto operacích. Současně tim umožnilo aktualizovat další softwarové algoritmy (např. násobení) a tím snížit výslednou velikost.

### 6.3.2 Vliv použití optimalizací

Je zcela patrné, že slepé generování kódu je méně efektivnější než jeho kontextový protějšek, který negeneruje tolik zbytečných operací. Slepé generování je stavěno tak, že generuje přesně podle vzorových šablon a nezískává žádné informace o dalším užití proměnných a dočasných uložišť a tudiž všechny mezivýsledky musíme uložit na simulovaném zásobníku.

Kontextové generování je však mnohem propracovanější, neboť si ukládá informaci o tom, kdy bude dále proměnná použita a nechává ji déle v registrech. V případě, že je registr s dočasnou proměnnou vyžádán a ta ještě bude v následujících instrukcích použita, pak se teprve uloží na zásobník a v případě potřeby z něj odebere.

Z tabulky 6.2 lze vidět, že díky kontextovému generování se generuje menší počet instrukcí načtení operandů, ale i jejich následné uložení. Jelikož se tabulka registrů v tomto typu generování souběžně snaží vrátit registr nejvhodnější i z hlediska aktuálních banků, snížil se i rapidně počet instrukcí `swbank` pro změnu banku registrů. Posledním výrazným snížením jsou instrukce inkrementu a dekrementu, které jsou využívány během práce se simulovaným zásobníkem, společně s instrukcemi pro práci s pamětí.

Protože se jednalo o krátké ukázky, často již ručně optimalizované ve zdrojových kódech, nelze sledovat výrazný vliv implementovaných optimalizačních technik na velikost výsledného kódu. Výrazný přínos optimalizací je však při psaní čitelnějších programů, kdy namísto magických konstant rozepíšeme počítané výrazy na více operací nebo v případě psaní programů laikem, které zaručí transformaci tohoto kódu do jeho kratšího ekvivalentu. Z těchto použitých optimalizací pak největším přínosem pro výsledný kód považuji aplikaci matematických výrazů, které zaručí čitelnost a především transformaci dělění a násobení mocninami dvojky na jejich ekvivalenty ve formě bitových posunů.

| Instrukce assembleru | Slepé generování | Kontextové generování |
|---|---|---|
| `mload` | 22 | 21 |
| `dload` | 25 | 7 |
| `iload` | 8 | 0 |
| `irload` | 2 | 2 |
| `dstore` | 11 | 5 |
| `istore` | 11 | 1 |
| `inc` | 13 | 5 |
| `dec` | 13 | 4 |
| `swbank` | 28 | 3 |

Tabulka 6.2: Srovnání počtu vybraných instrukcí assembleru při slepém a kontextovém generování na rozšířeném procesoru a demonstračním příkladu `demo_lcd_test.c`

### 6.3.3 Závěrečné zhodnocení

Experimentální výsledky ukazují význam a přednosti kontextového generování před běžným slepým podle pevných vzorových šablon. Současně ve všech úlohách, z pohledu velikosti kódu, zvítězila kombinace rozšířeného procesoru o nové instrukce s použitím optimalizací, generováním podle kontextu a vynucením pouze čtyř registrů. Lze vidět, že si v procesoru bohatě vystačíme s minimálním počtem registrů (kde minumum je rovno maximálnímu počtu potřebných registrů nejnáročnější operace, v našem překladači to jest 4 u dělení a násobení), za předpokladu že použijeme kontextové generování cílového kódu, které se svým způsobem snaží napodobit alokaci registrů prováděnou při manuální optimalizaci kódu na úrovni instrukcí. V hraničních případech dosahujeme redukce až o polovinu, což znamená u procesorů s omezeným prostorem pro program mnoho.

# Kapitola 7

# Závěr

Cílem práce bylo vytvořit překladač umožňující experimentovat s různými parametry překladu a demonstrovat jejich dopad na velikost výsledného kódu. Jeho hlavní význam spočívá v možnosti využití navrženého překladače při výuce pro propojení problematiky překladu zdrojových programů, jejich následné nahrání do procesoru a spuštění činnosti.

Přední část překladače podporuje většinu konstrukcí jazyka C, podle normy C99 [1] a jeho výsledný abstraktní syntaktický strom je tak univerzální pro použití u jiného překladače. Překladač korektně transformuje vstupní programy do instrukcí cílového assembleru podle možnosti, které poskytoval cílový procesor. Z hlediska typů podporuje pouze celočíselný typ `integer`, struktury, ukazatele a pole při jednodušších použití. Ve zdrojových programech psaných v jazyce C lze použit veškeré řídicí a podmíněné konstrukce jazyka a současně i tvořit podprogramy s vlastní i nevlastní rekurzí. Alternativní možností pro úsporu psaného kódu jsou pak *inline* funkce, jejichž těla se rozvinou v místě jejich volání. Výsledný kód lze dále přeložit do strojového kódu a spustit v procesoru běžícím na vývojovém přípravku FITkit.

S vytvořeným překladačem jsem blíže experimentoval a souběžně demonstroval vliv některých parametrů na velikost výsledného kódu. Hlavním přínosem je právě demonstrace jak se projeví rozšíření procesoru a jeho instrukční sady. Překladač lze takto využít při výuce, kdy ve spojení s procesorem ve formě FPGA projektu, kdy můžeme vidět, které instrukce má smysl doimplementovat do cílového procesoru, případně, které operace má smysl řešit hardwarovými jednotkami a nebo použit alternativní softwarové algoritmy. Toto má především význam u architektur s výrazným omezením zdrojů a prostředků.

Ačkoliv bylo věnováno mnoho úsilí, zůstává stále několik konstrukcí jazyka C, které nebyly implementovány. Jedná se například o některé z datových typů. Tento nedostatek však nikterak neomezuje použití překladače a jeho nasazení ve výuce. Abychom dále snížili počet generovaných instrukcí, bylo by vhodné rozšířit cílovou architekturu např. o FPU jednotku nebo o hardwarový zásobník nejen pro návratové adresy funkcí, čímž by odpadla nutnost simulace zásobníku v paměti, který zábírá relativně hodně paměti.

Interakce s procesorem probíhá přes prostředí `QDevKit`, které zajišťuje komunikaci s mikrokontrolerem, překlad FPGA projektů a jejich následné uložení do mikrokontroleru. `QDevKit` umožňuje tvorbu pluginů ve formě skriptů Pythonu. Realné by bylo vytvořit grafickou nadstavbu nad vytvořeným překladačem a tím umožnit psát zdrojové kódy v jazyce C a následně je nahrát do procesoru rovnou v tomto prostředí a tím usnadnit jejich spuštění.

# Literatura


[1] WG14/N1256 Committee Draft – September 7, 2007 ISO/IEC 9899:TC3 Contents.

[2] BARTSCH, H.-J.: *Matematické vzorce*, kapitola 12. Academia, čtvrté vydání, 2006, ISBN 80-200-1448-9, s. 774–776.

[3] BENEŠ, M.; ČEŠKA, M.; HRUŠKA, T.: *Překladače*. Skriptum VUT Brno, 1993, ISBN 80-214-0491-4, 262 str.

[4] ČEŠKA, M.: *Teoretická informatika: Učební texty*. FIT VUT v Brně, 2002.

[5] GAMMA, E.; HELM, R.; JOHNSON, R.; aj.: *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley Longman, Inc., 1997, ISBN 0-201-63442-2.

[6] HOWARTH, N.: Abstract Syntax Tree Design. Technická zpráva, ANSA, 1995.

[7] JONES, J.: Abstract Syntax Tree Implementation Idioms. In *Proceedings of the 10th Conference on Pattern Languages of Programs*, Illinois, 2003.

[8] MCGUIRE, P.: *Getting Started with Pyparsing*. O'Reilly Media, Inc., 2008, iSBN 9780596514235.

[9] MEDUNA, A.: *Elements of Compiler Design*. Auerbach Publications, 2008.

[10] NURMI, J.: *Processor Design: System-On-Chip Computing for ASICs and FPGAs*. Springer, 2007.

[11] PARR, T.: *The Definitive ANTLR Reference: Building Domain-Specific Languages*. Terence Parr, 2007, iSBN-13 978-09787392-4-9.

[12] RODEHEFFER, T.: Software Integer Division. Technická Zpráva MSR-TR-2008-141, Microsoft Research, August 2008.

[13] STALLMAN, R. M.; WEINBERG, Z.: *The C Preprocessor: For GCC verisou 4.3.2*. Mudranik Technologies Pvt Ltd / Pothi Books, 2007, iSBN-13 9788100004082.

[14] SUMMERFIELD, M.: *Programming in Python 3: A Complete Introduction to the Python Language*. Addison-Wesley, 2010, iSBN-13 978-0-321-68056-3.

# Příloha A

# Instrukční sada použitého procesoru

| Formát instrukce | | | | | | | | | | | | | | | | Instrukce | Popis instrukce |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 09 | 08 | 07 | 06 | 05 | 04 | 03 | 02 | 01 | 00 | | |
| 0 | 0 | 0 | 0 | - | - | - | - | - | - | - | - | 0 | 0 | 0 | 0 | **halt** | Zastav provádění programu |
| 0 | 0 | 0 | 0 | - | - | - | - | - | - | - | - | 1 | 1 | 1 | 1 | **nop** | Prázdná operace |
| 0 | 0 | 0 | 1 | a | a | v | v | v | v | v | v | v | v | v | v | **mload** rA, V | Nahrej do registru rA hodnotu V |
| 0 | 0 | 1 | 0 | a | a | d | d | d | d | d | d | d | d | d | d | **dload** rA, D | Nahrej do registru rA hodnotu uloženou na adrese D |
| 0 | 0 | 1 | 1 | a | a | d | d | d | d | d | d | d | d | d | d | **iload** rA, D | Nahrej do registru rA hodnotu uloženou na adrese M[D] |
| 0 | 1 | 0 | 0 | b | b | d | d | d | d | d | d | d | d | d | d | **dstore** D, rB | Ulož hodnotu z registru rB na adresu D |
| 0 | 1 | 0 | 1 | b | b | d | d | d | d | d | d | d | d | d | d | **istore** D, rB | Ulož hodnotu z registru rB na adresu M[D] |
| 0 | 1 | 1 | 0 | - | - | d | d | d | d | d | d | d | d | d | d | **branch** D | Skoč na adresu D |
| 0 | 1 | 1 | 1 | b | b | d | d | d | d | d | d | d | d | d | d | **brzero** rB, D | Skoč na adresu D, pokud je hodnota v registru rB rovna nule |
| 1 | 0 | 0 | 0 | b | b | d | d | d | d | d | d | d | d | d | d | **brpos** rB, D | Skoč na adresu D, pokud je hodnota v registru rB větší než nula |
| 1 | 0 | 0 | 1 | b | b | d | d | d | d | d | d | d | d | d | d | **brneg** rB, D | Skoč na adresu D, pokud je hodnota v registru rB menší než nula |
| 1 | 0 | 1 | 0 | a | a | b | b | - | - | - | - | 0 | 0 | 0 | 0 | **mov** rA, rB | Ulož do registru rA hodnotu z registru rB |
| 1 | 0 | 1 | 0 | a | a | b | b | - | - | - | - | 0 | 0 | 0 | 1 | **movn** rA, rB | Ulož do registru rA dvojkový doplněk hodnoty z registru rB |
| 1 | 0 | 1 | 0 | a | a | b | b | - | - | - | - | 0 | 0 | 1 | 0 | **inc** rA, rB | Ulož do registru rA inkrement hodnoty z registru rB |
| 1 | 0 | 1 | 0 | a | a | b | b | - | - | - | - | 0 | 0 | 1 | 1 | **dec** rA, rB | Ulož do registru rA dekrement hodnoty z registru rB |
| 1 | 0 | 1 | 0 | a | a | b | b | c | c | - | - | 0 | 1 | 0 | 0 | **add** rA, rB, rC | Ulož do registru rA součet hodnot z registru rB a rC |
| 1 | 0 | 1 | 1 | b | b | - | 0 | d | d | d | d | d | d | d | d | **outp** D, rB | Zapiš na port s adresou D hodnotu z registru rB |
| 1 | 0 | 1 | 1 | a | a | - | 1 | d | d | d | d | d | d | d | d | **inp** rA, D | Načti do registru rA hodnotu z portu na adrese D |

Tabulka A.1: Instrukční sada cílové architektury

# Příloha B

# Rozšíření instrukční sady použitého procesoru

| Formát instrukce | | | | | | | | | | | | | | | | | Instrukce | Popis instrukce |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 09 | 08 | 07 | 06 | 05 | 04 | 03 | 02 | 01 | 00 | | | |
| 0 | 0 | 0 | 0 | - | - | - | - | - | - | - | - | 0 | 0 | 0 | 0 | | **halt** | Zastav provádění programu |
| 0 | 0 | 0 | 0 | - | - | - | - | - | - | - | - | 0 | 0 | 0 | 1 | | **ret** | Návrat z podprogramu |
| 0 | 0 | 0 | 0 | a | a | - | - | b | b | - | 0 | 0 | 0 | 1 | 1 | | **irload** rA, rB | Nahrej do registru rA hodnotu z paměti na adrese určenou rB |
| 0 | 0 | 0 | 0 | a | a | - | - | b | b | - | 1 | 0 | 0 | 1 | 1 | | **irstore** rA, rB | Nahrej do paměti na adrese určené rB obsah registru rA |
| 0 | 0 | 0 | 0 | a | a | b | b | c | c | - | - | 0 | 1 | 0 | 0 | | **swbank** A, B, C | Změna banky pro jednotlivé registrové části instrukcí |
| 0 | 0 | 0 | 0 | a | a | b | b | c | c | - | - | 1 | 0 | 0 | 0 | | **and** rA, rB, rC | Logický součin registrů rA a rB s výsledkem v rC |
| 0 | 0 | 0 | 0 | a | a | b | b | c | c | - | - | 1 | 0 | 0 | 1 | | **or** rA, rB, rC | Logický součet registrů rA a rB s výsledkem v rC |
| 0 | 0 | 0 | 0 | a | a | b | b | c | c | - | - | 1 | 0 | 1 | 0 | | **xor** rA, rB, rC | Exkluzivní logický součet registrů ra a rB s výsledkem v rC |
| 0 | 0 | 0 | 0 | a | a | b | b | n | n | n | n | 1 | 0 | 1 | 1 | | **shr** rA, n, rB | Logický posun rB o N bitů doprava s výsledkem v rA |
| 0 | 0 | 0 | 0 | a | a | b | b | n | n | n | n | 1 | 1 | 0 | 0 | | **shl** rA, n, rB | Logický posun rB o N bitů doleva s výsledkem v rA |
| 0 | 0 | 0 | 0 | a | a | b | b | n | n | n | n | 1 | 1 | 0 | 1 | | **sra** rA, n, rB | Aritmetický posun rB o N bitů doprava s výsledkem v rA |
| 0 | 0 | 0 | 0 | a | a | b | b | n | n | n | n | 1 | 1 | 1 | 0 | | **sla** rA, n, rB | Aritmetický posun rB o N bitů doleva s výsledkem v rA |
| 1 | 0 | 1 | 0 | - | - | d | d | d | d | d | d | d | d | d | d | | **call** D | Volání rutiny na adrese D |

Tabulka B.1: Přidané instrukce cílového procesoru po rozšíření

# Příloha C

# Použité demonstrační příklady

```
/*
 * Demo for FitKit 1.2.
 * Implementation of simple bublesort
 */

// declaration for using ports
extern int port[256];

int main() {
  // sorted values for 1 to 5, all have +304 to get their
  // ascii value and with command for lcd write
  int pole[5] = {307, 308, 305, 309, 306};
  int pom;
  int index;

  // while port is busy
  while(port[2] != 0);
  port[2] = 128;

  // bubblesort part
  for(int j = 0; j < 4; j++) {
    for(int i = 0; i < 4; i++) {
      index = i + 1;
      if(pole[i] > pole[index]) {
        pom = pole[index];
        pole[index] = pole[i];
        pole[i] = pom;
      }
    }
  }

  // lcd output
  for(int k = 0; k < 5; k++) {
    while(port[2] != 0);
    port[2] = pole[k];
  }
}
```

Listing C.1: Bublinkové řazení

```
/*
 * Demo for FitKit 1.2.
 * Implementation of diode test, diode will be blinking
 */

// declaration for using ports
extern int port[256];

int main() {
```

```
  int test, a, b, c, d;
  // infinite cycle with some redundant computation
  while(1) {
    a = port[3];
    test = a;
    b = a;
    test += a;
    b = a + a;
    c = a + a + 1;
    c += a;
    d = c + test;

    // write to port with diode
    port[1] = d;
  }
}
```

Listing C.2: Test blikání diody D4

```
/*
 * LCD Demotest for FitKit 1.2
 */
extern int port[256];

// inline function used for waiting while busy
inline void wait() {
  while(port[2] != 0);
  return;
}

int main() {
  int login[8] = { 120, 102, 105, 101, 100, 111, 48, 49};
  // busy, waiting for 0
  wait();
  // set cursor to start
  port[2] = 128;

  for(int i = 0; i < 8; i++) {
    wait();
    // 256 is command for lcd otuput
    port[2] = login[i] + 256;
  }

  wait();
  port[2] = 167;

  for(int j = 8; j >= 0; j--) {
    wait();
    port[2] = login[j] + 256;
  }

  return 0;
}
```

Listing C.3: Test zobrazení textu na LCD FitKitu

```
/*
 * Demo for FitKit 1.2.
 * Computation of several operations
 */

int main() {
  int first_op = 120;
  int second_op = 15;
  int results[5];
  results[0] = first_op / second_op; // = 8 (8)
  results[1] = first_op % second_op; // = 0 (0)
  results[2] = first_op | second_op; // = 127 (7F)
```

```
  results[3] = first_op & second_op; // = 8 (8)
  results[4] = first_op ^ second_op; // = 119 (77)
}
```

Listing C.4: Zpracování několika aritmetických operací

```
/*
 * Demo for FitKit 1.2.
 * Implementation of test for perfect number
 */

// declaration for using ports
extern int port[256];

int main() {
  int tested = 6;
  int result = 1;

  // test all divisors of number
  for(int i = 2; i <= tested / 2; i++) {
    if(tested % i == 0) {
       result += i;
    }
  }

  /// set lcd cursor to 0
  while(port[2] != 0);
  port[2] = 128;

  // if result is same as sum of all positive divisors of
  // number then number is perfect and 1 is shown on lcd
  // else 0
  while(port[2] != 0);
  if(result == tested) {
    // 256 (command) + 48 (to ascii) + 1 = true
    port[2] = 305;
  } else {
    // 256 (command) + 48 (to ascii) + 0 = false
    port[2] = 304;
  }
}
```

Listing C.5: Test, zda je číslo dokonalé (tj. jest součtem svých kladných dělitelů)

# Příloha D

# Obsah přiloženého CD

- /**architectures** – cílové architectury
  - /**inp** – základní verze cílového procesoru
  - /**inp-ext** – rozšířená verze cílového procesoru
- /**demos** – sada demonstračních zdrojových souborů v jazyce C
- /**software** – implementace překladače jazyka C
  - /**common** – kolekce pomocných modulů
  - /**compiler** – moduly spojené s překladovou činnosti
  - /**libs** – použité moduly třetích stran
- /**thesis** – textová část bakalářské práce
- /**tools** – dodatečné nástroje
  - /**cpuide** – plugin do QDevKitu pro práci s assemblerem cílového procesoru
  - /**cpuide-ext** – rozšířená verze pluginu