VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

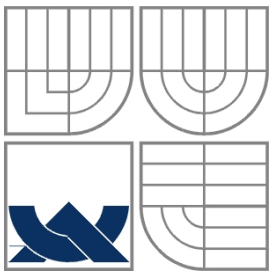
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

ZADNÍ ČÁST PŘEKLADAČE PODMNOŽINY JAZYKA C PRO 8-BITOVÝ PROCESOR

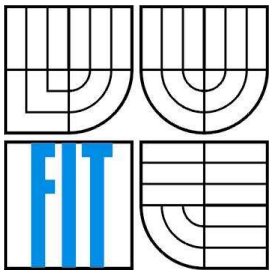
DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE Bc. JAKUB HORNÍK
AUTHOR

BRNO 2011



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

ZADNÍ ČÁST PŘEKLADAČE PODMNOŽINY JAZYKA C PRO 8-BITOVÝ PROCESOR

COMPILER BACK-END OF SUBSET OF LANGUAGE C FOR 8-BIT PROCESSOR

DIPLOMOVÁ PRÁCE
MASTER‘S THESIS

AUTOR PRÁCE
AUTHOR
Bc. JAKUB HORNÍK

VEDOUCÍ PRÁCE
SUPERVISOR
Ing. ZBYNĚK KŘIVKA, Ph.D.

BRNO 2011

Abstrakt

Překladač umožňuje programátorovi popisovat algoritmus ve vysokoúrovňovém programovacím jazyce s vyšší mírou abstrakce a strukturovaností, než poskytuje nízkoúrovňový strojový kód. Tato práce se týká návrhu zadní části překladače podmnožiny jazyka C pro 8bitový procesor Xilinx PicoBlaze-3, který je zde popsán od počátečního výběru vhodné přední části, návrhu architektury, až po samotnou implementaci. Jedním z důvodů této práce je, že není k dispozici uspokojující překladač pro tento procesor.

Abstract

A compiler allows us to describe an algorithm in a high-level programming language with a higher level of abstraction and readability than a low-level machine code. This work describes design of the compiler back-end of subset of language C for 8-bit soft-core microcontroller Xilinx PicoBlaze-3. Design is described from the initial selection of a suitable framework to the implementation itself. One of the main reasons of this work is that there is not any suitable compiler for this processor.

Klíčová slova

kompilátor, Low Level Virtual Machine Compiler, mezikód, překladač, PicoBlaze, PicoBlaze C Compiler, Small Device C Compiler, zadní část překladače

Keywords

back-end, compiler, intermediate code, Low Level Virtual Machine Compiler, PicoBlaze, PicoBlaze C Compiler, Small Device C Compiler

Citace

Horník Jakub: Zadní část překladače podmnožiny jazyka C pro 8-bitový procesor, diplomová práce, Brno, FIT VUT v Brně, 2011

Zadní část překladače podmnožiny jazyka C pro 8-bitový procesor

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením Ing. Zbyňka Křivky, Ph.D. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……
Jakub Horník
25. 5. 2011

Poděkování

Rád bych poděkoval Ing. Zbyňku Křivkovi, Ph.D. za poskytnuté konzultace a cenné rady, které byly stěžejní při tvorbě této práce.

Obsah

Úvod

V současné době se při vytváření softwarového produktu, ať už ve formě komplexního programu, nebo jednoduchého skriptu, využívá výhradně některý z vyšších (vysokoúrovňových) programovacích jazyků. Avšak, než je možné výsledný program spustit, je potřeba jej přeložit do formy, ve které může být vykonán na cílové platformě. Tento úkol má na starost program nazývaný *překladač* (či *kompilátor*). Překladač tedy umožňuje programátorovi popisovat algoritmus ve vysokoúrovňovém programovacím jazyce s vyšší mírou abstrakce a strukturovaností. Zároveň je zdrojový kód kratší a čitelnější.

Cílem této diplomové práce je vytvořit překladač jazyka C pro procesor PicoBlaze-3 firmy Xilinx s využitím již existující přední části[1] překladače. Procesor PicoBlaze je kompaktní, velmi jednoduchý a cenově nenáročný softwarový osmibitový RISC procesor, navržený firmou XILINX pro využití v jejich FPGA čipech.

První kapitola poskytuje čtenáři teoretické informace o překladačích, včetně popisu jednotlivých fází, na které bývá většina překladačů dělena.

Ve druhé kapitole je nejprve popsán samotný procesor PicoBlaze, do jehož jazyka symbolických instrukcí bude probíhat překlad. V druhé částí této kapitoly se rozebírá existující překladač, konkrétně PicoBlaze C Compiler (PCComp) a jeho hlavní nedostatky. Zastaralost a celkové nedostatky tohoto překladače byly jedním z hlavních motivů pro tvorbu této práce.

Ve třetí kapitole je popsán proces výběru vhodné přední části, pro kterou byly k dispozici dvě varianty. Ty byly porovnány na základě instrukcí jejich mezikódu, což bylo jedno z hlavních kritérií pro výběr.

Kompilátor, který byl vybrán ve třetí kapitole, je ve čtvrté kapitole detailněji rozepsán se zaměřením především na zadní část[2].

Pátá kapitola se týká samotného návrhu zadní částí překladače. Je zde popsáno mapování mezikódu vybraného kompilátoru na jazyk symbolických instrukcí procesoru PicoBlaze. V závěru kapitoly je navržena architektura zadní části.

Na základě návrhu popsaného v páté kapitole byla zadní část překladače implementována, čemuž se věnuje šestá kapitola. Implementace byla rozdělená do dvou modulů. První má na starost práci s registry, druhý modul sestává z funkcí pro generování cílového kódu.

Sedmá kapitola shrnuje vylepšení, které se podařilo s tímto překladačem dosáhnout oproti staršímu PCComp. Byla provedena celá řada testů překladu, jejichž výsledky se nacházejí na přiloženém datovém nosiči. Na závěr je provedeno shrnutí této diplomové práce a nastíněny možné rozšíření.

[1] Přední část (tzv. front-end) překladače – část závislá na vstupním jazyce
[2] Zadní část (tzv. back-end) – část závislá na cílové architektuře

1 Úvod do překladačů

Následující text čerpá, a více informací lze zjistit, z [2, 5, 6]. Překladač je zjednodušeně řečeno program, který čte program v jednom jazyku (tzv. zdrojový jazyk) a překládá jej do ekvivalentního programu v jiném jazyce (tzv. cílový program). Výstupním jazykem nejčastěji bývá nízkoúrovňový strojový kód cílové platformy. Překladač tedy umožňuje programátorovi popisovat algoritmus ve vysokoúrovňovém programovacím jazyce s vyšší mírou abstrakce a strukturovaností. Zároveň je zdrojový kód kratší a čitelnější. Další důležitou vlastností překladače je, že informuje o veškerých chybách zdrojového programu, na které se v průběhu překladu narazí. Pro překladač se také užívá označení *kompilátor*, který označuje překladač, jehož vstupem je vysokoúrovňový jazyk a výstupem je jazyk nízkoúrovňový.

Prozatím jsme na překladač nahlíželi jako na černou skříňku, která pro zadaný vstupní program produkuje výstupní. Pokud se na něj zaměříme detailněji, můžeme jej rozdělit na dvě části: *analýzu* a *syntézu*.

Účelem analýzy je rozdělit zdrojový program na logicky oddělené části a sestavit z nich gramatický strom. Z něj se pak sestaví *vnitřní reprezentace* zdrojového programu (také označována jako *mezikód*). Pokud se v rámci analýzy zjistí, že zdrojový program je syntakticky nebo sémanticky nesprávný, informuje o tom uživatele, na kterém potom je, aby provedl patřičné zásahy do zdrojového programu, vedoucí k opravení těchto zjištěných nedostatků. Dalším úkolem analýzy je sbírat informace o zdrojovém programu jako názvy proměnných, jejich datové typy apod., do struktury označované *tabulka symbolů*. Ta je spolu s vnitřní reprezentací zdrojového programu (mezikódem) předána syntéze.

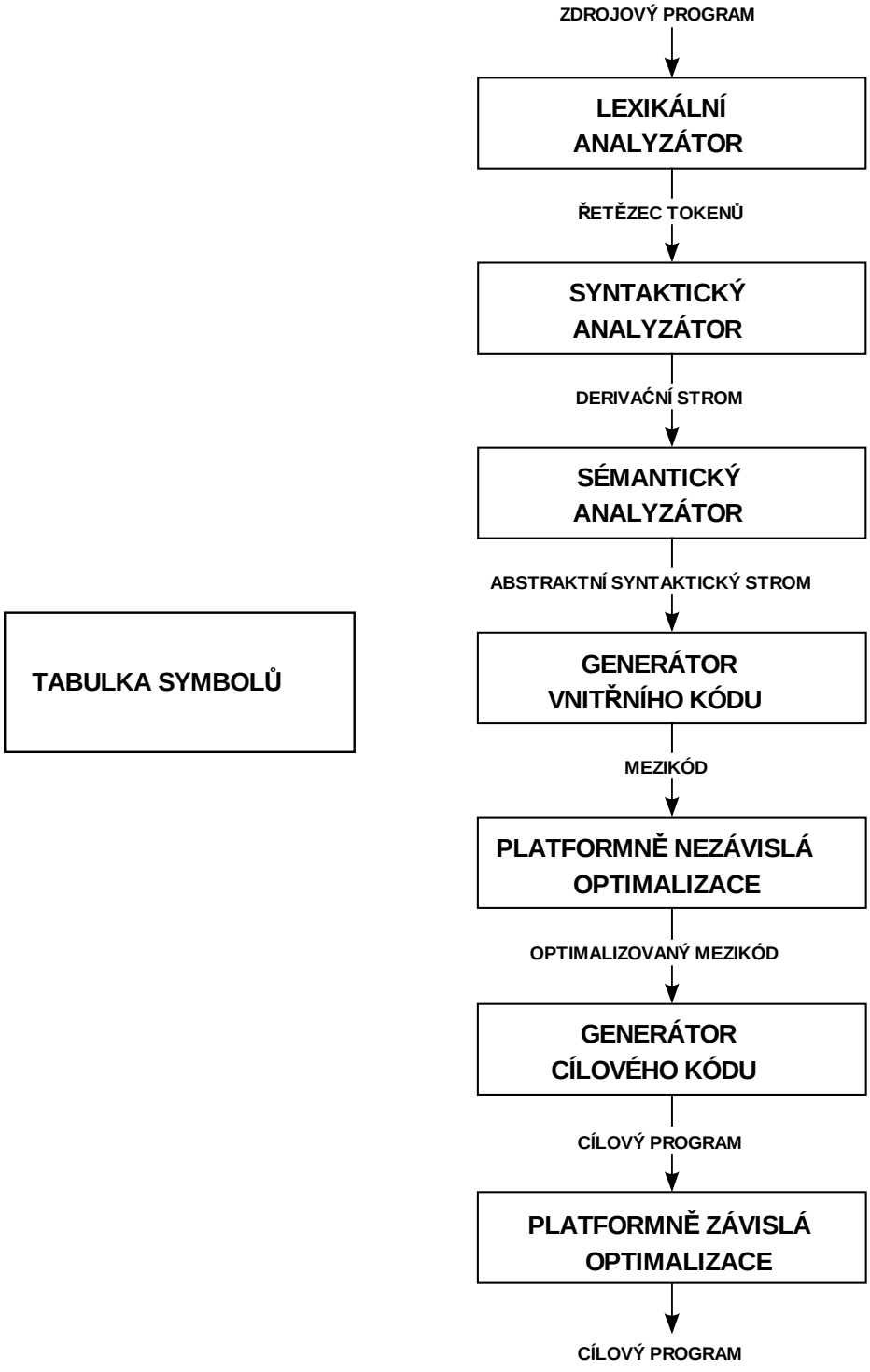
Úkolem syntézy je sestavit cílový program z mezikódu a tabulky symbolů, které jsou výstupem analýzy. Analýza se také označuje jako přední část překladače, syntéza jako zadní část překladače.

Z hlediska závislosti na zdrojovém a cílovém jazyku můžeme o tomto rozdělení tvrdit:

- **Přední část** (anglicky front-end) – část závislá na vstupním jazyce.
- **Zadní část** (anglicky back-end) – část závislá na cílové platformě procesoru.

Výhodou této architektury je, že se kompilátor rozdělí na dvě relativně nezávislé jazykové transformace, přičemž překlad ze zdrojového jazyka do mezikódu je společný pro více cílových platforem, a až typem zadní části se určuje cílová platforma. Navíc záměnou přední části, při zachování struktury mezikódu, docílíme podpory pro jiný zdrojový jazyk.

Pokud se zaměříme na samotný proces překladu detailněji, můžeme jej z logického hlediska rozdělit do určitých podprocesů, které nazýváme *fáze*. Na obrázku 1 vidíme typické rozdělení překladače na jednotlivé fáze. Tabulka symbolů, která obsahuje informace o celém zdrojovém programu je přístupná všem dílčím fázím překladu. Většina překladačů obvykle provádí mezi přední a zadní části platformově nezávislou optimalizaci. Jejím úkolem je zoptimalizovat mezikód tak, aby zadní část produkovala lepší cílový kód než bez této optimalizace.



Obrázek 1: Fáze překladače

1.1 Lexikální analýza

První fází překladu je lexikální analýza, v angličtině označována *scanner*. Lexikální analyzátor čte řetězec znaků zdrojového programu, ze kterých tvoří lexikální jednotky, zvané *lexémy*. Lexikální jednotka odpovídá klíčovému slovu, identifikátoru, operátoru, číselné hodnotě, oddělovacímu znaku či komentáři (ten se většinou v této fázi zahazuje, jelikož pro samotný překlad nemá uplatnění). Všechny možné lexémy a případná struktura zdrojového jazyka bývají popsány pomocí regulárních

výrazů. Scanner bývá nejčastěji konstruován jako konečný automat, který na základě vstupního znaku a aktuálního stavu provádí odpovídající akci, ať už je to přechod do jiného stavu, nebo generování výstupu.

Pro každý lexém produkuje lexikální analyzátor výstup, tzv. *token*, který má tvar

```
<jméno_tokenu, hodnota_atributu>
```

a je předán následující fázi, nazvané syntaktická analýza. První složka *jméno_tokenu* je abstraktní symbol, který se používá u syntaktické analýzy, druhá část *hodnota_atributu* je ukazatel do tabulky symbolů na tento token.

1.2 Syntaktická analýza

Druhou fází je syntaktická analýza, v angličtině označována *parser*. Úlohou syntaktické analýzy je rozpoznat, zda zdrojový program je syntakticky správně napsaný, tzn. například v jazyce C, zda jsou všechny složené závorky {} v páru, ale také určuje prioritu matematických operátoru apod. Z tohoto důvodu využívá syntaktický analyzátor první složky tokenů, produkované lexikální analýzou, a vytváří z nich rozklad zdrojového programu ve formě *derivačního (syntaktického) stromu*[3], kde každý vnitřní uzel reprezentuje operaci a potomci uzlu argumenty dané operace.

1.3 Sémantická analýza

Sémantický analyzátor zpracovává derivační strom z předešlého kroku a kontroluje, zda jednotlivé výrazy vstupního programu jsou sémanticky korektní. Zároveň přidává informace o datových typech proměnných do tabulky symbolů i do derivačního stromu.

Důležitou součástí sémantické analýzy je *typová kontrola*, kdy se kontroluje, zda každý operátor má operandy platných datových typů. Nebo se také kontrolují výrazy na levé a pravé straně přiřazovacího příkazu, aby se zabránilo přiřazení nekompatibilních typů (např. číslo s plovoucí desetinnou čárkou do proměnné s pevnou řádovou čárkou).

Sémantický analyzátor může provádět i *typové konverze* tam, kde je to namístě. Například u binárních operací, kdy jedním z operandů je celé číslo a druhým operandem je číslo s plovoucí desetinnou čárkou.

Výstupem je abstraktní syntaktický strom, což je ve své podstatě derivační strom z předchozího kroku doplněný o typové informace, případné typové konverze apod.

1.4 Generátor vnitřního kódu

Smyslem této fáze je generovat lineární posloupnost instrukcí ze stromové struktury překládaného programu, která vznikla v předešlých fázích. Tato vnitřní reprezentace zdrojového kódu (nebo taky nazývána mezikód) musí splňovat dvě důležité vlastnosti:

- Musí být snadno generovatelná z derivačního stromu.
- Musí být jednoduše přeložitelná do cílového jazyka.

[3] Derivační strom je orientovaný, kořenový strom, který reprezentuje syntaktickou strukturu slovního řetězce podle formální gramatiky

Vnitřní reprezentace bývá nejčastěji ve formě *postfixové* (reverzní Polské) notace, *dvoj-adresného* (*trojice*) nebo *tří-adresného* kódu (*čtveřice*). Valná většina jazyků symbolických instrukcí procesorů je tvořena ze tří částí: *operátor*, *zdrojový operand* a *cílový operand*. Například instrukce procesoru PicoBlaze `ADD s0, s1` tvoří trojici, která sečte obsah registrů `s0` a `s1`, a výsledek uloží do registru `s0`. Reprezentace této trojice v jazyce C by mohla být:

```
a += b;
```

a typická matematická reprezentace stejné operace:

```
(+=, a, b)
```

Čtveřice neboli tří-adresná instrukce se skládá ze čtyř částí. Následující čtveřice má dva zdrojové operandy, operátor a cílový operand:

```
a = b1 + b2
```

Matematická reprezentace této čtveřice by mohla vypadat:

```
(+, a, b1, b2)
```

Jen pro doplnění, některé čtveřice, jako například přiřazení, mají pouze tři části a čtvrtá je volná. Ta se obvykle značí pomlčkou:

```
(=, a, b, -)
```

Dvoj-adresný kód má jednu výhodu oproti tří-adresnému, a to tu, že je velmi podobný cílovým jazykům symbolických instrukcí, takže generování cílového kódu je jednodušší. Na druhou stranu, tří-adresný kód má výhody dvě. Pro stejný zdrojový kód produkuje kompaktnější kód, než dvoj-adresný. Například čtveřice `(+, a, b1, b2)` vyžaduje dvě trojice, aby se dosáhlo stejného efektu:

```
( =, a, b1)
(+=, a, b2)
```

Rovněž je se čtveřicemi možné provádět určité optimalizace mnohem snáze, než s trojicemi, které jsou více pozičně závislé. Například s předešlou dvojicí se musí zacházet jako s nedělitelným blokem při jakékoliv optimalizaci, která přeskupuje kód kvůli větší efektivnosti.

Poslední z běžných možností, jak reprezentovat mezikód, je postfixová notace. Ta se rovněž vyznačuje některými výhodami. Vzhledem k absenci závorek je vyhodnocování výrazů mnohem snazší než u ostatních typů. Zároveň překladač nemusí alokovat dočasné proměnné pro vyhodnocování postfixových výrazů, místo toho se využívá zásobník. Všechny operandy jsou vloženy na zásobník, a jsou nad nimi prováděny operace v definovaném pořadí, které ovlivňují vždy pouze horní operandy. Například následující výraz:

```
(1 + 2) * (3 + 4)
```

se v postfixové notaci zapíše jako:

```
1 2 + 3 4 + *
```

1.5 Optimalizace kódu

Smyslem optimalizace je vylepšit vnitřní kód či samotný výstupní kód tak, aby jeho provádění bylo co možná nejefektivnější. Nejde tedy o nalezení nejoptimálnější varianty, jejíž nalezení může být výpočetně nemožné, nebo alespoň velmi časově náročné.

Optimalizovat můžeme mezikód (označováno jako strojově nezávislá optimalizace) nebo generovaný kód (strojově závislá optimalizace), kde se využívá lineární (peephole) optimalizace. Tu si můžeme představit jako plovoucí okno (odtud peephole = kukátko, skulina), které se pohybuje po generovaném kódu, prozkoumává se jeho aktuální obsah, a kdykoliv to je vhodné, nahradí sekvenci instrukcí uvnitř okna za efektivnější nebo rychlejší.

Mezi další optimalizační metody patří například:

- Zabalení konstanty
- Šíření konstanty
- Kopírování proměnné
- Eliminace mrtvého kódu
- Rozbalení cyklu

Jednotlivé metody jsou podrobněji popsané v kapitole 4.2, která se týká optimalizací mezikódu, které provádí přední část překladače Small Device C Compiler.

1.6 Generování cílového kódu

Cílem této fáze je vygenerovat z mezikódu, v tomto okamžiku již optimalizovaném, výstupní kód v generovaném cílovém jazyce. Tento blok má definováno mapování instrukcí mezikódu na výstupní jazyk. Tato fáze je jedna z hlavních částí implementovaných v rámci řešení této diplomové práce a je podrobně popsána v pozdějších kapitolách.

2 Procesor XILINX PicoBlaze

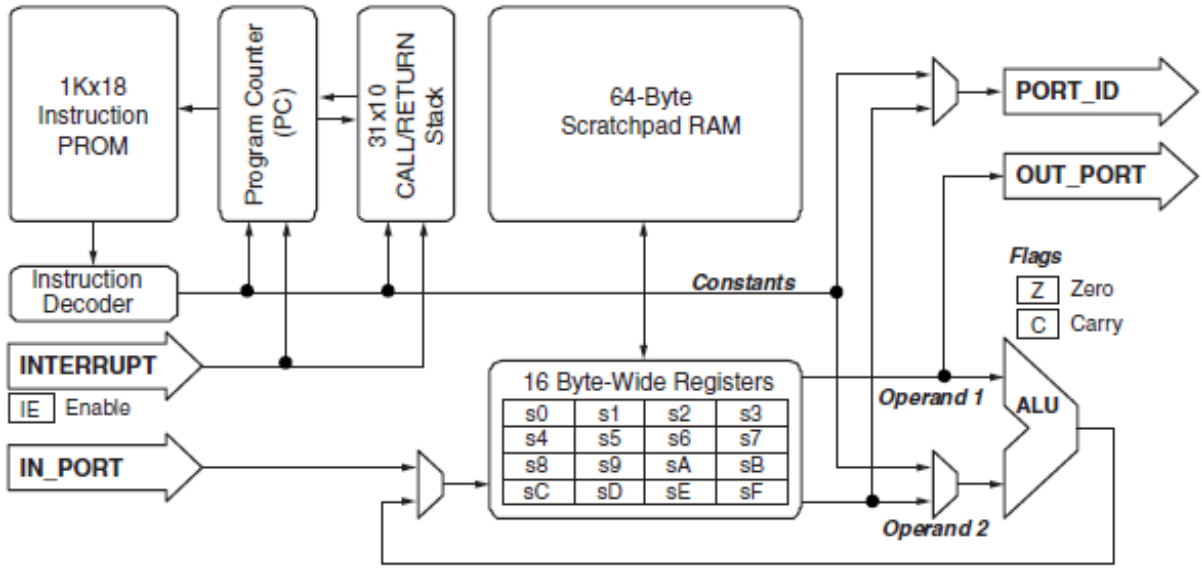
PicoBlaze-3 (dále v textu bude uváděn pouze jako PicoBlaze) je kompaktní, velmi jednoduchý a cenově nenáročný softwarový osmibitový RISC procesor, navržený firmou XILINX pro využití v FPGA čipech rodiny Spartan-3 a Virtex. Hlavní výhodou je jeho velikost, jelikož v případě čipu Spartan-3 zabírá pouze 96 FPGA bloků[4]. Rychlost procesoru se pohybuje v závislosti na zvoleném FPGA v rozmezí 44 až 100 MIPS (milion instrukcí za sekundu).

Procesor PicoBlaze je plně začlenitelný do cílového FPGA a nevyžaduje žádné dodatečné externí zdroje. Základní funkcionalitu lze jednoduše rozšířit připojením dodatečné FPGA logiky na jeho vstupní a výstupní porty. [1]

2.1 Architektura procesoru

Základní architektura procesoru PicoBlaze je znázorněna na obrázku 2. Mezi základní rysy patří:

- 8bitová aritmeticko-logická jednotka (ALU)
- příznaky CARRY a ZERO
- 16 osmibitových registrů pro všeobecné použití
- paměť pro 1024 instrukcí
- 64bajtů datové paměti (scratchpad RAM)
- jeden vstup pro externí přerušení
- 256 I/O portů pro rozšíření
- automatický zásobník pro 31 návratových adres
- reakce na přerušení do 5 taktů



Obrázek 2: architektura procesoru PicoBlaze [1]

[4] blok (neboli slice) – programovatelný element uvnitř FPGA jako např. multiplexor, blok paměti RAM apod.

2.2 Instrukční sada

Každá z dedikovaných instrukcí procesoru PicoBlaze se provede v jednom strojovém cyklu. Pro přístup k paměti dat slouží instrukce `STORE` a `FETCH`, které mají jako parametr zdrojový/cílový registr a adresu paměti. Procesor umožňuje provádět základní aritmetické operace, avšak neobsahuje žádné dedikované instrukce pro násobení a dělení. Téměř všechny instrukce mají jako parametry registry, popř. konstanty. Veškeré aritmetické operace jsou osmibitové a znaménkové. Avšak některé instrukce jako `ADDCY` a `SUBCY` využívají příznak `CARRY`, což umožňuje, při jejich vhodném uspořádání, výpočet výrazů delších než 8 bitů. Zde se však už jedná o skupinu příkazů, a takový výpočet poté zabere několik strojových cyklů (2 pro 16bitové hodnoty, 4 pro 32bitové hodnoty).

Kompletní instrukční sada procesoru PicoBlaze je vložena v příloze A.

2.3 Přerušení

Procesor PicoBlaze poskytuje jeden zdroj externího přerušení. Po resetu je přerušení standardně zakázáno. Jeho povolení se provede instrukcí `ENABLE INTERRUPT` a opětovné zakázání pomocí instrukce `DISABLE INTERRUPT`. Při příchodu přerušení se dokončí právě prováděná instrukce, aktuální obsah PC (Program Counter – programový čítač) se uloží na zásobník a provede se skok na adresu 0x3FF (tzv. vektor přerušení). Zde obvykle bývá skoková instrukce na obslužnou rutinu. Návratová instrukce z podprogramu přerušení má dvě možnosti, a to pokud při návratu chceme opětovně povolit přerušení - `RETURNI ENABLE`, nebo ho ponechat zakázané - `RETURNI DISABLE`.

2.4 Překladač jazyka C pro PicoBlaze - PCComp

PCComp (PicoBlaze C Compiler) je jednoduchý překladač jazyka C na jazyk symbolických instrukcí procesoru PicoBlaze. Tento překladač se již nevyvíjí a jeho poslední verze je z roku 2005. Zároveň byla z projektové stránky odstraněna možnost jeho stáhnutí a nalezení jiného zdroje na Internetu je velmi obtížné, ne-li nemožné. Jelikož nejsou k dispozici zdrojové kódy kompilátoru, je jediným možným způsobem k určení jeho charakteristik uživatelský manuál, vydaný autorem programu (viz [7]).

2.4.1 Základní nedostatky

- Jak již zmiňuje sám autor, překladač PCComp neprovádí žádný druh optimalizace a výsledný kód je velmi neefektivní, v některých případech pak zcela nefunkční.
- Generování probíhá na principu zásobníkového kódu, což je vzhledem k absenci zásobníku a velikosti paměti dat neefektivní.
- Jednoduchý preprocesor podporuje pouze základní direktivy:
  `#asm, #endasm, #ifdef, #ifndef, #else, #endif (#include, #define)`
  Neprovádí kontrolu, zda např. každému `#asm` odpovídá `#endasm`. Tato kontrola je ponechána na uživateli.
- Maximální datový typ je celočíselný 16bitový typ.
- Pouze částečná podpora explicitních typových konverzí.

- Doporučuje se používat pouze globální proměnné, jelikož práce s lokálními proměnnými je velmi neefektivní a má za následek rychlé spotřebování veškeré volné paměti.
- Podpora pouze jednorozměrných polí.
- Není podporována ukazatelová aritmetika.
- Parametrem funkce nemůže být ukazatel nebo pole.

Některé uvedené nedostatky však mohou být limitovány samotným procesorem PicoBlaze a jejich případná implementace může být velmi náročná, popř. nemožná.

2.4.2 Formulace cíle

Z výše uvedených nedostatků můžeme sestavit množinu, které lze vylepšit v rámci řešení této diplomové práce a to zejména:

- Optimalizace.
- Podpora vícerozměrných polí.
- Ukazatele na proměnné a pole.
- Částečná podpora ukazatelové aritmetiky.
- Preprocessing a plná podpora syntaxe C99.

Částečná optimalizace, preprocessing a podpora syntaxe jazyka C dle C99 je splnitelná výběrem vhodné přední části překladače. Zbylé body je potřeba implementovat v rámci zadní části překladače.

2.4.3 Ukázka překladu

Zdrojový kód v jazyku C:

```
int a, b;
a = b+1;
```

Výstup překladače - jazyk symbolických instrukcí procesoru PicoBlaze:

```
FETCH XL, _b_low
FETCH XH, _b_high

SUB YL, 01
STORE XH,(YL)
SUB YL, 01
STORE XL,(YL)
LOAD ZL,01
LOAD ZH,00
FETCH XL,(YL)
ADD YL, 01
FETCH XH,(YL)
ADD YL, 01
ADD XL, ZL
ADDCY XH, ZH

STORE XL, _a_low
STORE XH, _a_high
RETURN
```

Na výše uvedeném výstupním kódu z kompilátoru PCComp je patrný zásobníkový princip generování. To s sebou nese celou řadu instrukcí pro přístup k datové paměti. I když se jedná o procesor architektury RISC, a tedy přístup do paměti trvá stejně cyklů jako k registrům, je tato volba generování značně neefektivní.

3 Výběr přední části překladače

K volbě vhodné přední části se nabízejí dva volně dostupné překladače, které jsou založeny na principu jednotné přední části překladače generující mezikód společný pro všechny cílové architektury a zadní částí, která je závislá na cílové architektuře. Těmito překladači jsou SDCC (Small Device C Compiler) a LLVM (Low Level Virtual Machine Compiler).

Základním, a rozhodujícím kritériem při výběru vhodné přední části překladače je generovaný mezikód. Vzhledem k jednoduchosti instrukční sady procesoru PicoBlaze musí být co nejjednodušší.

3.1 Small Device C Compiler

Small Device C Compiler (SDCC) je volně dostupný (open-source) překladač jazyka C vyvinutý především pro 8bitové procesory. I když existuje celá řada volně dostupných překladačů jazyka C pro obecné procesory (např. GCC a níže uvedené LLVM), tak SDCC je v současné době jediný volně dostupný překladač zabývající se specifickými potřebami 8bitových procesorů. Většina těchto procesorů je založena na Harvardské architektuře, což znamená, že programová a datová paměť je fyzicky oddělena a mají rozdílný adresový prostor. To s sebou přináší jiný postup při samotné konstrukci překladače, který SDCC řeší. Další velkou výhodou SDCC je velice efektivní optimalizace a vestavěný debugger [4].

Struktura SDCC je dělena na přední (platformě nezávislou) a zadní (platformě závislou) část a umožňuje přidávat další části (v terminologii SDCC označované jako port). V současné době podporuje SDCC překlad do těchto portů [8]:

- Intel 8051
- Maxim 80DS390
- Zilog Z80
- Microchip PIC16
- Freescale (dříve Motorola) HC08
- částečná podpora rodiny Microchip PIC

Překlad probíhá ve dvou krocích. V prvním kroku je generován mezikód jazyka, který je v SDCC označován jako `iCode` (popř. `BBlock` v jeho blokové variantě). Ten obsahuje veškeré důležité informace, které jsou potřeba pro efektivní generování výsledného jazyka symbolických adres ve druhém kroku překladu [8].

3.2 Low Level Virtual Machine Compiler

LLVM je kolekce nástrojů a knihoven určena pro tvorbu překladačů, jejíž vývoj započal v roce 2000 na univerzitě Illinois. Původně se jednalo pouze o výzkumný projekt, který měl prozkoumat techniky dynamického kompilování programovacích jazyků. Ten se postupně rozrůstal a v současné době se jedná o rozsáhlou infrastrukturu podporující širokou škálu programovacích jazyků (C, C++, Objective-C, Fortran, Ada, Haskell, Java, Python, Ruby, ActionScript, GLSL, atd.).

Architektonicky vychází LLVM z klasické struktury překladačů, dělící se na přední a zadní část, s mezikódem jako spojovacím článkem. Tato vnitřní reprezentace instrukcí je platformě nezávislá a je založena na principu SSA (Static single assignment), což znamená, že každá proměnná je přiřazena

právě jednou. Tím je velmi zjednodušena analýza závislostí mezi proměnnými. LLVM také umožňuje velmi důkladné optimalizace, které mají za následek velice efektivní generovaný výstupní kód [9, 10].

3.3 Porovnání mezikódu

V následující kapitole jsou porovnány jednotlivé mezikódy překladačů SDCC a LLVM. Instrukce jsou podle jejich použití rozděleny do několika kategorií.

3.3.1 Binární operace

V této části jsou porovnány binární operace, ke kterým patří klasické aritmetické a binární operace.

SDCC

Následující instrukce mají 2 operandy a výsledek - `IC_LEFT, IC_RIGHT, IC_RESULT`. První sloupec značí, jak instrukce pojmenovány v samotném mezikódu.

'+'	**sčítání**
'-'	**odčítání**
'*'	**násobení**
'/'	**dělení**
'%'	**modulo**
'>'	**větší než**
'<'	**menší než**

Bitové binární operace

'^'	**exkluzivní OR (XOR)**
'|'	**bitová nonekvivalence**
EQ_OP	**operace rovno**
AND_OP	**logické AND**
OR_OP	**logické OR**
BITWISEAND	**bitové AND**
LEFT_OP	**levý posuv**
RIGHT_OP	**pravý posuv**

Výslednou sémantika těchto instrukcí v C zápisu lze zapsat ve tvaru:
`IC_RESULT = IC_LEFT <operace> IC_RIGHT`.

LLVM

Instrukce mají dva operandy `<op1>, <op2>`, parametr určující typ číselné hodnoty `<ty>` a výsledek `<result>`. Operandy mohou reprezentovat vektor hodnot daného typu. Syntaxi instrukcí lze zapsat ve tvaru `<result> = <operace> <ty> <op1>, <op2>`.

add	**celočíselné sčítání**
fadd	**sčítání čísel s pohyblivou řádovou čárkou (FP)**
sub	**celočíselné odčítání**

fsub	**odčítání FP**
mul	**celočíselné násobení**
fmul	**násobení FP**
udiv	**celočíselné dělení, unsigned**
sdiv	**celočíselné dělení, signed**
fdiv	**dělení FP**
urem	**zbytek po celočíselném dělení, unsigned**
srem	**zbytek po celočíselném dělení, signed**
frem	**zbytek po dělení, FP**

Bitové binární operace LLVM

shl	**posuv doleva o daný počet bitů**
lshr	**logický posuv doprava o daný počet bitů**
ashr	**aritmetický posuv doprava o daný počet bitů**
and	**bitové AND**
or	**bitový OR**
xor	**bitový XOR**

Binární instrukce, pokrývající základní matematické operace, jsou u obou překladačů téměř totožné. LLVM dělí instrukce na celočíselné, a na operace nad čísly s pohyblivou desetinnou čárkou (FP). Zároveň je u každé instrukce specifikována velikost operandů (i8, i16, …). U SDCC jsou instrukce definovány obecněji a velikost i typ operandů jsou specifikovány v parametru konkrétního operandu. Implementaci v LLVM komplikuje možnost počítání s vektory.

Zároveň instrukční sada procesoru PicoBlaze neumožňuje násobení a dělení. Tyto instrukce je proto nutné simulovat pokud jsou potřeba (např. iterativní přičítání/odčítání, posuv pří dělení nebo násobení dvěma) [3].

Mezikód SDCC obsahuje jak bitové, tak i logické operace. Operace pro posuv jsou pouze základní (aritmetický posuv doleva/doprava o daný počet kroků). Naproti tomu LLVM obsahuje instrukce pro aritmetický i logický posuv a opět s možností specifikovat o kolik bitů dojde k posuvu.

3.3.2 Instrukce řízení toku

Následující instrukce slouží k řízení toku programu.

Instrukce SDCC

`CALL`	volání funkce reprezentované `IC_LEFT` `IC_RESULT = IC_LEFT();`
`PCALL`	volání funkce přes ukazatel `IC_RESULT = (*IC_LEFT)();`
`RETURN`	návrat z funkce
`LABEL`	návěští
`GOTO`	skok na návěští
`JUMPTABLE`	skok na dané návěští dle podmínky
`IFX`	podmínkový skok `if (IC_COND) goto IC_TRUE else goto IC_FALSE;`

SDCC obsahuje instrukci pro volání podprogramu přes ukazatel. S tím může být při implementaci problém, protože PicoBlaze neumožňuje explicitně manipulovat s instrukčním ukazatelem.

Instrukce LLVM

ret	návrat z podprogramu
br	skok na návěští (základní blok)
switch	skok na dané návěští dle podmínky (C switch)
indirectbr	nepřímý skok na danou adresu
invoke	volání funkce
call	volání funkce

Obě implementace obsahují instrukce pro přímý skok (skok na návěští), volání podprogramu a návrat z něj. Skokové instrukce u SDCC jsou řešeny pomocí návěští, u LLVM na začátek základního bloku, což je však v praxi to samé (návěští je jednou z možností, kdy dochází ke vzniku nového bloku). LLVM dále umožňuje nepřímý skok na danou adresu.

3.3.3 Paměťové instrukce

Tyto instrukce slouží pro práci s pamětí jako načtení hodnoty z paměti a podobně.

Instrukce SDCC

`IPUSH`	uložení operandu na zásobník
`IPOP`	získání operandu z vrcholu zásobníku
`GET_VALUE_ AT_ADDRESS`	načte hodnotu ze zadané adresy `IC_RESULT = (*IC_LEFT);`
`POINTER_SET`	ulož hodnotu na zadanou adresu `(*IC_RESULT) = IC_RIGHT;`
`ADDRESS_OF`	získání adresy operandu `IC_RESULT = &IC_LEFT();`

SDCC poskytuje standardní množinu instrukcí. Instrukci `ADDRESS_OF` je potřeba řešit na úrovni překladače.

Instrukce LLVM

load	načtení operandu (hodnoty) z paměti
store	uložení operandu (hodnoty) do paměti
getelementptr	získání adresy operandu

alloca	alokuje paměť na zásobníku pro vykonávanou funkci, automaticky uvolní při RET, vrací ukazatel na alokovanou paměť
	`<result> = alloca <type>`
	`[, <ty> <NumElem>][, align <alignment>]`

Paměťový model SDCC je mnohem jednodušší než u LLVM. SDCC má klasické instrukce *push* a *pop* pro uložení jedné hodnoty na datový zásobník. Práce s LLVM je v tomto případě daleko složitější. LLVM umožňuje alokovat na zásobníku bloky dat, instrukce *alloca* vrací ukazatel na tento

blok. Zásobník se uvolňuje až při návratu z funkce. Tento model, kdy každá funkce má svůj zásobník, je vzhledem k velikosti paměti procesoru PicoBlaze nevhodný. Navíc má PicoBlaze pro návratové adresy vlastní paměť.

3.3.4 Konverzní operace

Následující podkapitola představuje možnosti přetypování operandu v jednotlivých mezikódech.

Instrukce SDCC

`CAST`	konverzní funkce, typ konverze je dán typem IC_LEFT

Mezikód SDCC má pouze jedinou konverzní funkci, typ konverze je dán jedním parametrem a konvertovaná hodnota druhým. To lze zapsat pomocí sémantiky v C zápisu následovně:

```
IC_RESULT = (typeof IC_LEFT) IC_RIGHT;
```

Instrukce LLVM

trunc	ořeže (konvertuje) hodnotu s větším rozsahem na menší
zext	rozšíří hodnotu s menším rozsahem na větší
sext	znaménkové rozšíření
fptrunc	ořeže číslo s pohyblivou řádovou čárkou (FP)
fpext	rozšíří FP hodnotu na větší rozsah
fptoui	konverze FP na bezznaménkový celočíselný typ
fptosi	konverze FP na znaménkový celočíselný typ
uitofp	konverze bezznaménkového celočíselného typu na FP
sitofp	konverze znaménkového celočíselného typu na FP
ptrtoint	konverze ukazatele na celočíselný typ
inttoptr	konverze celočíselného typu na ukazatel
bitcast	změna interpretace bitové reprezentace dané hodnoty

Konverzní funkce u LLVM jsou v porovnání s SDCC značně rozsáhlé a komplikovanější. Zároveň každý typ konverze má svou vlastní instrukci.

3.3.5 Ostatní instrukce

Následují zbylé instrukce, které nešlo zařadit do výše uvedených tříd.

SDCC

`'='`	přiřazení
`UNARYMINUS`	unární mínus (`IC_RESULT = - IC_LEFT;`)
`GETHBIT`	nejvyšší bit v proměnné `IC_RESULT = (IC_LEFT >> (sizeof(IC_LEFT)*8 -1));`
`SEND`	předávání parametrů funkci
`RECV`	získaní předaných parametru ve funkci

Mezi zajímavé instrukce mezikódu SDCC zde patří `SEND` a `RECEIVE`, které umožňují efektivní předávání parametrů do funkce.

LLVM

icmp	porovnání celočíselných hodnot, typ určen parametrem (větší, menší, rovno,…)
fcmp	porovnání hodnot s plovoucí desetinnou čárkou
select	výběr hodnoty na základě podmínky, bez skoku (z hodnot nebo vektoru)
va_arg	zpracování argumentů předávaných přes pole va_list

LLVM dále obsahuje několik instrukcí pro práci s vektory a poli.

3.4 Volba překladače

Vzhledem k zjištěným okolnostem je výhodnější jako překladač pro procesor PicoBlaze zvolit SDCC. Tato platforma je primárně vytvořena pro vestavěná zařízení, a proto je generovaný mezikód více uzpůsoben pro využití v této skupině procesorů. Zároveň díky instrukcím SEND a RECEIVE pro předávání parametrů funkcím lze aspoň částečně optimalizovat práci s pamětí, a pro předávání parametrů využít registry, kterých má PicoBlaze dostatečné množství. Paměť, popř. zásobník, lze využít až po obsazení určitého množství registrů.

Naproti tomu LLVM je psáno obecněji, a pro širokou škálu procesorů. Vzhledem k nutnosti simulovat instrukce pro operace se zásobníkem a velikosti datové paměti procesoru PicoBlaze by byla práce s paměťovým modelem LLVM daleko obtížnější než SDCC. Další výhodou SDCC oproti LLVM je celkový počet instrukcí mezikódu (viz konverzní a aritmetické funkce).

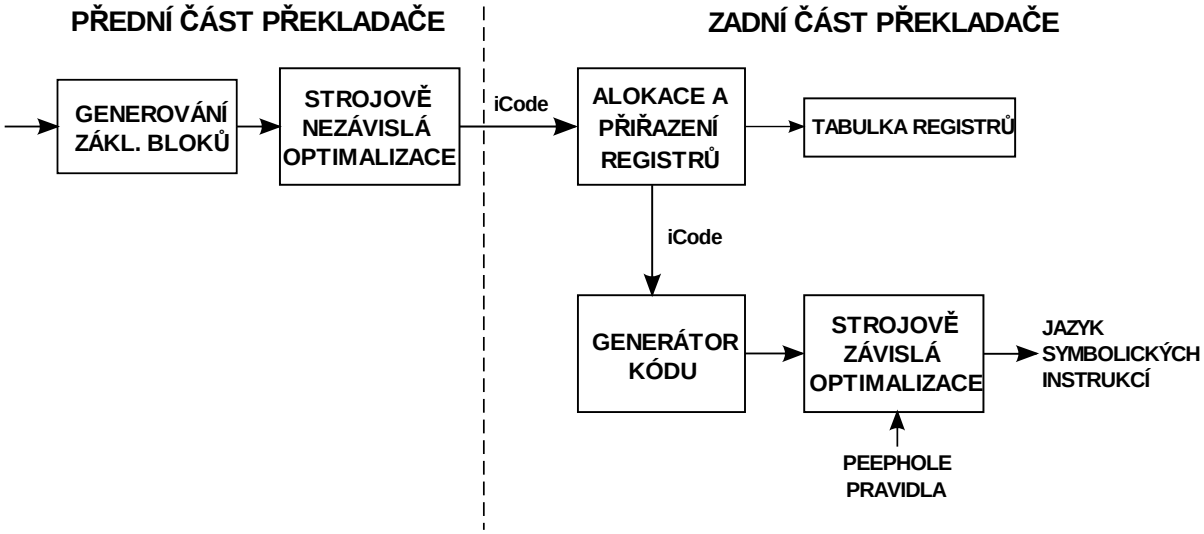
4 Architektura překladače SDCC

Small Device C Compiler, jak již bylo řečeno v kapitole 3.1, se drží základních praktik tvorby překladačů, a je rozdělen na přední a zadní část, což jej umožňuje rozšířit o podporu dalších procesorů (portů). Obrázek 3 znázorňuje architekturu SDCC včetně jednotlivých funkčních bloků, ze kterých se jednotlivé části skládají.

Přední část překladače SDCC můžeme zjednodušeně rozdělit na dva moduly. Prvním je *Generátor základních bloků*, který pro zadaný vstupní soubor provede lexikální analýzu následovanou syntaktickou a sémantickou kontrolou. Pokud se zjistí, že kód nevyhovuje pro některé z pravidel těchto kontrol, vypíše se na standardní chybový výstup odpovídající chybová hláška, včetně řádku zdrojového souboru, kde byla nalezena. Výstupem tohoto modulu je interní reprezentace abstraktního syntaktického stromu (AST) `eBBlock`. Z tohoto AST se generuje mezikód ve formě tří-adresných instrukcí, který je v názvosloví SDCC pojmenovaný `iCode`.

Druhým modulem je *Strojově nezávislá optimalizace.* Ten provádí určité druhy optimalizace, které jsou nezávislé na cílovém portu, a mají za úkol zefektivnit mezikód, než se z něj vygeneruje posloupnost instrukcí cílového jazyka symbolických instrukcí.

Tím končí zpracování zdrojového programu prováděné přední částí překladače, které je společné pro všechny porty. Následující zpracování je již unikátní pro každou cílovou platformu, ale principiálně vychází z jednoho postupu, který rovněž můžeme rozdělit do několika kroků. Pokud nebereme v potaz optimalizaci, můžeme generování cílového programu z mezikódu rozdělit do dvou kroků.



Obrázek 3: Architektura překladače SDCC

Prvním krokem, a v blokovém schématu pojmenovaným, je *Alokace a přiřazení registrů*. Jeho smyslem je přidělovat a spravovat registry daného procesoru v daném portu. V tomto modulu jsou přiřazeny jednotlivým operandům instrukcí mezikódu registry ještě před tím, než je provedeno mapování z mezikódu na cílový jazyk, což se děje až v dalším kroku. Díky tomu je možné efektivně zacházet s registry procesoru, a v konečném důsledku to má výrazný vliv na kvalitu výstupního kódu. Pro procesory se složitější architekturou se jedná o jedno z nejvýhodnějších řešení.

Druhým funkčním modulem zadní části překladače je *Generátor kódu*. Tento modul na základě vstupní instrukce mezikódu a přidělených registrů vygeneruje odpovídající sekvenci instrukcí

cílového jazyka. Na implementační úrovni si tuto fázi můžeme představit jako rozsáhlý rozhodovací prvek z konstrukce switch jazyka C, který dle identifikátoru instrukce mezikódu volá funkci, která se postará o vygenerování odpovídajících instrukcí cílového jazyka.

V poslední fázi zadní části překladače se provádí peephole optimalizace dle definovaných optimalizačních pravidel. Výstupem je program přeložený do cílového jazyka, který se linkerem, a následně asemblerem převede na strojový kód cílového procesoru.

4.1 Mezikód překladu (iCode)

Pro vnitřní reprezentaci abstraktního syntaktického stromu ve formě tří-adresných instrukcí slouží mezikód pojmenovaný *iCode*. I když je tento pomocný jazyk překladu bez specifické vazby na konkrétní typ procesoru, má již k nim velice blízko, aby bylo možné co nejsnadněji z jeho instrukcí vygenerovat cílový kód. *iCode* předpokládá existenci neomezeného počtu registrů, což má za následek generování značného množství pomocných proměnných. Dále předpokládá neomezenou datovou paměť a neomezený zásobník, se kterým se pracuje pomocí operací IPUSH a IPOP. Mezikód programu tvoří obousměrně provázaný seznam *iCode* instrukcí.

Každá instrukce *iCode* je popsána strukturou stejného jména, která obsahuje veškeré informace potřebné pro generování cílového kódu. Mezi ně patří například:

- Identifikátor instrukce `op`
- Sekvenční číslo v rámci funkce `seq`
- Hloubka zanoření `depth`
- Ukazatele na operandy (`operand`) instrukce `lrr` (případně podmínkový operand a k němu odpovídající návěští při splnění/nesplnění podmínky `cnd`)
- Registry přiřazené instrukci `rUsed`
- Registry používané během zpracování instrukce `rMask`
- Příznaky specifické pro generování
- Ukazatel na předchozí a následující instrukci (`prev, next`)

4.1.1 Operand

Každý operand instrukce je popsán strukturou stejného jména `operand`. V této struktuře je definováno zejména, zda se jedná o symbolický operand (`symbol`), nebo o hodnotu (`value`). Dalšími příznaky této struktury se určuje, zda je operand:

- adresa
- globální proměnná
- volatile proměnná
- ukazatel

4.1.2 Symbol

Struktura `symbol` popisuje operand instrukce, který je ve formě symbolické proměnné. Každý symbol má definované jméno, které je pro globální proměnné shodné s tím, jak bylo uvedené ve zdrojovém souboru. Pro dočasné proměnné je ve formě `iTemp<číslo>`. Kromě toho se zde nalézají informace o životi proměnné, jejichž analýza se provádí v přední části (`liveFrom`, `liveTo`). Díky těmto hodnotám je možné efektivně zacházet s přidělováním registrů operandům.

Pokud jsou proměnné přiřazeny registry, jsou jejich ukazatele uloženy v poli `regs`. Dále obsahuje tato struktura celou řadu příznaků, které ovlivňují vlastnosti a práci s danou proměnnou.

4.1.3 Hodnota

Struktura `value` slouží pro uchování číselné hodnoty, která je jednou z možností operandu. Kromě číselné hodnoty je uloženo i o jaký typ hodnoty se jedná – celočíselná, bezznaménková, číslo s plovoucí desetinnou čárkou, apod.

4.1.4 Ukázka instrukce

Pomocí parametru kompilátoru (např. `–dumpall`) lze vypisovat instrukce mezikódu v řádkovém formátu.

Například následující instrukce byla vygenerována při překladu ze zdrojového souboru s názvem `test.c` a její zdrojová instrukce byla na řádku 16. V rámci funkčního bloku se jedná o třetí iCode instrukci, která přiřadí do globální proměnné `gch` (`liveFrom = liveTo = 0`) typu `char` číselnou hodnotu 10 (v šestnáctkové soustavě 0xA).

```
test.c(l6:s3:k3:d0:s0) _gch [k2 lr0:0 so:0]{ ia0 a2p0 re0 rm0 nos0 ru0
dp0}{char fixed} := 0xa {char literal}
```

4.2 Optimalizace mezikódu

Jak již bylo zmíněné, překladač SDCC provádí celou řadu optimalizací na mezikódu překladu. Mezi nejdůležitější druhy optimalizací například patří [8]:

Eliminace mrtvého kódu

Cílem je odstranit operandy, které jsou nedostupné, nepoužívají se, nebo je jim zbytečně přiřazena hodnota. Následující příklad ukazuje provedení eliminace mrtvého kódu na zdrojový kód uvedený v levé části. Výstupem je kód uvedený vpravo.

```
int global;

void f () {
  int i;      /* bez použití */
  i = 1;      /* mrtvé přiřazení */
  global = 1; /* mrtvé přiřazení */
  global = 2;
  return;
  global = 3; /* nedostupné */
}
```

->

```
int global;

void f () {
  global = 2;
}
```

Propagace a kopírování konstanty

Účelem je optimalizovat kód takovým způsobem, že se místo přiřazení proměnné použije číselná hodnota tam, kde je to zřejmé.

```
int f() {
  int i, j;
  i = 10;
  j = i;
  return j;
}
```

Ve funkci uvedené výše se proměnné `i` přiřadí konstanta `10`. Hned další instrukce přiřadí hodnotu proměnné `i` do proměnné `j`. V této době, během překladu, ale hodnotu proměnné `i` známe. Tuto instrukci proto můžeme nahradit za přímé přiřazení konstanty. To samé nastává i u návratové hodnoty, kde taky můžeme propagovat konstantu. Po provedení optimalizace tedy dostáváme následující funkci:

```
int f() {
  int i, j;
  i = 10;
  j = 10;
  return 10;
}
```

Vytknutí výrazů

Při výpočtu několika po sobě následujících výrazů může dojít k situaci, že určité podvýrazy jsou u několika výpočtů totožné a zbytečně se vícekrát počítají. Aby se tomu zamezilo, provede optimalizátor vytknutí tohoto výrazu, který se vypočte předem, a proměnná s jeho hodnotou se poté dosadí na původní pozice. Tuto optimalizaci znázorňuje následující transformace zdrojového kódu.

```
i = x + y + 1;
j = x + y;
```

->

```
iTemp = x + y;
i = iTemp + 1;
j = iTemp;
```

Optimalizace cyklů

Překladač SDCC provádí dva druhy optimalizace cyklů, a to vyjmutí neměnného výrazu z cyklu (angl. *Loop-invariant code motion*) a nahrazení složitější a dražší operace za jednodušší (angl. *Strength reduction*).

Pokud se ve zdrojovém kódu nachází cyklus, ve kterém se stále počítá určitý výraz, jehož operandy i výsledná hodnota jsou po celou dobu neměnné, můžeme výpočet tohoto výrazu přesunout před cyklus. Výsledek se uloží do pomocné proměnné, která se poté dosadí na původní pozici výrazu. Tato optimalizace má velmi blízko k vytknutí výrazu, kde se ale výraz vyskytoval v kódu několikrát. V tomto případě se může v cyklu vyskytovat i jen jednou, avšak musí být splněna podmínka neměnnosti operandů výrazu uvnitř cyklu. Následující transformace znázorňuje provedení této optimalizace.

```
for (i = 0; i < 100; i++)
  f += k + l;
```

->

```
iTemp = k + l;
for (i = 0; i < 100; i++)
  f += iTemp;
```

Druhý typ optimalizace *Strength reduction* například nahradí dělení konstantou za násobení s odpovídající konstantou, to se týká zejména čísel s plovoucí desetinnou čárkou. Další možností je nahrazení násobení za cyklus s opakovaným přičítáním, což znázorňuje následující transformace.

```
for (i = 0; i < 100; i++) {
  array[i*5] = i*3;
}
```

->

```
iTemp1 = 0;
iTemp2 = 0;
for (i = 0; i < 100; i++) {
  array[iTemp1] = iTemp2;
  iTemp1 += 5;
  ITemp2 += 3;
}
```

Algebraické zjednodušení

Překladač SDCC provádí celou řadu algebraických zjednodušení konkrétního výrazu nezávisle na ostatních. Můžeme jmenovat například:

- Přičítání nuly
  `i = j + 0;  ->  i = j;`
- Odčítání stejné hodnoty
  `i = j - j;  ->  i = 0;`
- Násobení číslem 0
  `i = j * 0;  ->  i = 0;`
- Násobení číslem 1
  `i = j * 1;  ->  i = j;`
- Dělení číslem 1
  `i = j / 1;  ->  i = j;`
- Dělení čísla mocninou 2 (2, 4, 8, 16, …)
  `i /= 2;  ->  i >>= 1;`
  `i /= 4;  ->  i >>= 2;`

4.3 Peephole optimalizace

Překladač SDCC využívá mechanismus pro vyhledávání a nahrazení vzorů založený na definovaných pravidlech. Tyto pravidla mohou být definována pro každý port v souboru s příponou `def`, ze kterého jsou poté přidány do překladače. Druhou možností je definovat soubor s pravidly jako parametr příkazové řádky `--peep-file <filename>` při spouštění překladu. Následující kód ilustruje příklad takového pravidla.

```
replace {
     FETCH %1, %2
     LOAD  %1, %3
} by {
     LOAD  %1, %3
}
```

Aplikováním tohoto pravidla na následující posloupnost instrukcí:

```
FETCH s0, 10
LOAD  s0, s2
```

dostaneme:

```
LOAD  s0, s2
```

Číselné hodnoty s procentem u přepisujícího pravidla reprezentují zástupné symboly, které peephole optimalizátor ztotožní s odpovídajícími parametry dané instrukce. Pravidlo uspěje jen tehdy, pokud na pozicích se stejným zástupným symbolem jsou i shodně pojmenované parametry.

Pokud se za klíčové slovo `replace` uvede `restart`, je při úspěšném nahrazení prohledáváno znovu od začátku kódu, jinak se pokračuje od pozice ihned za provedenou změnou. Vzor pro nahrazení nemůže být prázdný, ale můžeme vkládat i komentáře, které se uvozují středníkem.

4.4 Port

Slovem *'Port'* se v názvosloví SDCC označuje rozšíření překladače specifické pro konkrétní procesor a jeho architekturu. V podstatě se jedná o vlastní implementaci zadní části kompilátoru, která má za úkol přeložit vstupní program v mezikódu do cílového jazyka.

Volba cílového portu se provádí přepínačem `-m<port>` příkazové řádky kompilátoru. Každý procesor může mít několik modelů, které jsou zpracovávány jedním portem. Pro tento účel existuje ještě parametr `-p`, který slouží pro přesnější specifikaci portu. Rozdíly mezi jednotlivými modely jsou potom popsány například pomocí konfiguračního souboru, čímž je zajištěno rozšíření o další podporované modely bez nutnosti rekompilace. Příkladem může být port procesoru PIC16, který podporuje několik desítek cílových platforem.

Pro popis portu slouží struktura `PORT` v souboru `src/port.h`, ve které jsou popsány nejdůležitější aspekty cílové architektury. Mezi ně patří například:

- Jméno procesoru a jeho rodiny
- Ukazatel na funkci generující cílový kód, pokud se nemá použít defaultní (`do_glue`)
- Pokyny pro spuštění assembleru
- Pokyny pro linker
- Odkaz na soubor s optimalizačními pravidly
- Velikosti základních datových typů (`char, short, int, long, float`, …)
- Informace o paměťových regionech
- Konfigurace zásobníku
- Volba endianity (`little_endian, big_endian`)
- Podpora dělení, násobení a posunů
- Ukazatel na funkci pro přiřazování registrů, což je vstupní funkce zadní části a je volána přední části po sestavení mezikódu z funkce (`assignRegisters`)
- Ukazatele na funkce pro generování inicializačních instrukcí procesoru, tabulky vektoru přerušení apod.
- Seznam klíčových slov specifických pro daný port (`keywords`)

5 Návrh překladače

Tato kapitola popisuje návrh zadní části překladače, která rozšiřuje kompilátor SDCC popsaný v předešlé kapitole o podporu překladu do jazyka symbolických instrukcí procesoru PicoBlaze. Důležitou částí návrhu je kromě definice samotné architektury i určení vazeb mezi instrukcemi mezikódu a cílového jazyka. Pro každou instrukci mezikódu je proto v této kapitole definována odpovídající sekvence instrukcí cílového jazyka.

5.1 Definice datových typů vstupního jazyka

Tato kapitola definuje velikosti datových typů podmnožiny jazyka C tak, jak jsou popsány u každého rozšíření překladače SDCC. Ke každému datovému typu existuje i jeho bezznaménková (**unsigned**) varianta.

- **char** - 1 Bajt (standardní velikost)
- **short** - 2 Bajty
- **int** - 2 Bajty
- **long** - 4 Bajty
- **ptr** - 1 Bajt (pro adresaci datové paměti procesoru PicoBlaze stačí 6 bitů, kdy se bere nižších 6 dané proměnné, horní 2 se ignorují)

Uvedené velikosti datových typů se berou jako konstantní, avšak je možné je změnit v konfiguračním souboru portu SDCC. Vzhledem k tomu, že přední část překladače SDCC je stavěna pro maximální velikost proměnné 4B, bere se tato velikost jako hraniční.

Čísla s plovoucí desetinou čárkou a operace s nimi se nebudou implementovat, jelikož nejsou podporovány ze strany procesoru PicoBlaze a jejich implementace by byla náročná. Zároveň práce s nimi by byla krajně neefektivní a zdlouhavá.

5.2 Mapování mezikódu na jazyk symbolických adres

Tato kapitola definuje převod mezikódu na jazyk symbolických adres procesoru PicoBlaze. Pro zjednodušení popisu je předpokládáno, že levý operand se již nachází v registru označeném `rL` a případný pravý operand v registru `rRy` a tyto registry není možno modifikovat. Toto označení registrů koreluje s operandy mezikódu SDCC označených `IC_LEFT` pro levý a `IC_RIGHT` pro pravý. Výsledek operace se uloží do registru označeného `rRes`. Instrukce jsou rozděleny na fáze načtení (fetch) a vykonání (execute) s případným uložením (writeback).

5.2.1 Instrukce součtu '+'

V závislosti na datových typech operandů může nastat několik možností. Veškeré operace jsou znaménkové. U bezznaménkových operací se ignoruje příznak přetečení u nejvyššího bajtu. Níže použité zkratky `rL_MSB` (`rR_MSB`) znamenají vyšší (významnější) bajt a `rL_LSB` (`rR_LSB`) méně významový (nižší) bajt daného operandu.

Součet 8 bitů + 8 bitů = 8 bitů (char + char = char)

```
LOAD  rRes, rL
ADD   rRes, rR
```

Součet 16 bitů + 16 bitů = 16 bitů (short, int)

```
LOAD rRes_MSB, rL_MSB
LOAD  rRes_LSB, rL_LSB

ADD   rRes_LSB, rR_LSB
ADDCY rRes_MSB, rR_MSB
```

Pokud dojde k situaci, kdy jeden nebo oba operandy jsou typu `char` (velikost 1 bajt) a výsledek typu `int` (velikost 2 bajty), je potřeba provést přetypování a provést součet druhou možností. O to se však stará jiná instrukce.

Součet větších hodnot lze řešit rozšířením 16bitového výpočtu o další instrukce `ADDCY` v pořadí bajtů datového typu od nižší po vyšší.

5.2.2 Instrukce rozdílu '-'

Operace rozdílu je podobná jako součet.

Rozdíl 8 bitů – 8 bitů = 8 bitů

```
LOAD  rRes, rL
SUB   rRes, rR
```

Rozdíl 16 bitů – 16 bitů = 16 bitů

```
LOAD  rRes_MSB, rL_MSB
LOAD  rRes_LSB, rL_LSB

SUB   rRes_LSB, rR_LSB
SUBCY rRes_MSB, rR_MSB
```

Jedná se o odčítání znaménkových hodnot stejně jako u sčítání. Pro bezznaménkové platí, co již bylo zmíněno výše.

5.2.3 Negace UNARYMINUS

Instrukční sada PicoBlaze neposkytuje dedikovanou instrukci negace, lze ji však simulovat pomocí nonekvivalence a přičtení 1.

```
LOAD  rRes, rL

XOR   rRes, FF
ADD   rRes, 01
```

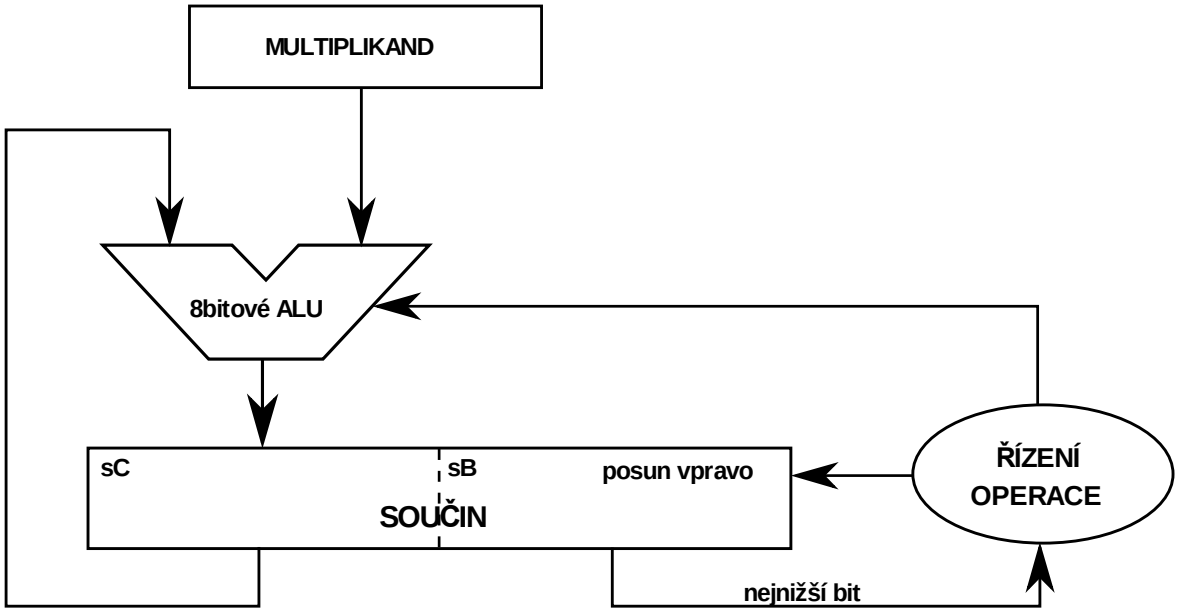
5.2.4 Násobení

Operace násobení také nemá dedikovanou instrukci. Lze využít algoritmu „přičti a posunuj“. Princip tohoto algoritmu je popsán v [3]. Menší komplikací jsou operace se znaménkovými čísly, kdy je třeba před samotnou operací zkontrolovat, zda je operandem záporné číslo a případně jej převést na kladné, s tím, že se tato změna poznamená. Po provedení samotné operace se poté přihlédne ke znaménkům vstupních operandů, a na základě znaménkových pravidel pro násobení a dělení se patřičně upraví znaménko výsledku operace. Jelikož pro operace násobení a dělení je potřeba větší množství registrů, budou označovány stejně jako v procesoru PicoBlaze.

5.2.4.1 Algoritmus násobení

Obrázek 4 graficky znázorňuje obvod, který umožňuje vynásobit dvě kladná čísla, aniž by k tomu použil dedikovanou instrukci pro násobení. Obvod sestává z aritmeticko-logické jednotky, řídicí logiky a tří registrů, kdy dva z nich jsou spojeny do jednoho registru o dvojnásobné délce. Na začátku obsahuje spodní polovina (`sB`) tohoto složeného registru násobenec, horní polovina (`sC`) je vynulována. Třetí samostatný registr obsahuje násobitel. Pro jejich lepší rozlišení budeme tento registr dále označovat pouze jako násobitel, kdežto posuvným registrem budeme označovat zbylý složený registr.

Pro výpočet součinu potřebuje obvod *X* kroků, kde *X* je bitová šířka registru. Na konci každého tohoto kroku se provede aritmetický posun spojeného registru doprava o jeden bit. Pokud je ale aktuální hodnota nejnižšího bitu log. 1, přičte se napřed k horní polovině registru násobitel. Po ukončení výpočtu se výsledný součin nachází v posuvném registru.



Obrázek 4: Princip algoritmu pro násobení

5.2.4.2 Implementace algoritmu pomocí instrukcí procesoru PicoBlaze

Následující funkce, která je přepisem algoritmu z předešlé kapitoly do instrukcí procesoru PicoBlaze, vypočte součin dvou bezznaménkových 8bitových čísel, které jsou uloženy v registrech `sB` a `sC`.

Výsledek má 16bitů a po návratu bude uložen ve stejných registrech, kdy `sB` bude obsahovat nižší bajt a `sC` vyšší. Algoritmus navíc ještě vyžaduje dva pomocné registry `sD` a `sE`.

Násobení bezznaménkových čísel 8 bitů * 8 bitů = 16 bitů

```
_muluchar:
           LOAD  sE, 08     ; 8 cyklů
           LOAD  sD, sC
           LOAD  sC, 00

_loop:     TEST  sB, 01     ; nejnižší bit == 1
           JUMP  Z, _noadd
           ADD   sC, sD

_noadd:    SR0   sC         ; posuv doprava
           SRA   sB
           SUB   sE, 01
           JUMP  NZ, _loop

           RETURN
```

Jak již bylo zmíněno v odstavci výše, před samotným násobením znaménkových čísel se musí zjistit znaménka obou operandů, a pokud jsou záporné, převést je na kladné, s tím, že se tato změna poznamená. Po provedení operace násobení se na základě původní kombinace znamének nastaví odpovídající výsledné znaménko tak, jak je definováno v pravidlech násobení čísel (viz Tabulka 1). Znaménko operandu nejsnadněji zjistíme testováním jeho nejvyššího bitu, kdy log. 0 odpovídá kladnému a naopak.

Levý operand	Pravý operand	Výsledné znaménko
+	+	+
-	+	-
+	-	-
-	-	+

Tabulka 1: Vztah mezi znaménky operandů a výsledku u násobení

Násobení znaménkových čísel 8 bitů * 8 bitů = 16 bitů

```
_mulschar:
           LOAD sA, 00
           TEST sB, 80          ; test prvního operandu
           JUMP Z, _neg1
           XOR sB, FF
           ADD sB, 01
           LOAD sA, 01          ; poznamenání změny
                                                    ->
```

```
_neg1:      TEST sC, 80           ; test druhého operandu
            JUMP Z, _neg2
            XOR sC, FF
            ADD sC, 01
            XOR sA, 01            ; poznamenání změn\

_neg2:      CALL _muluchar
            TEST sA, 01           ; test zda je nutno negovat
            JUMP Z, _end          ; výsledné znaménko
            XOR sB, FF
            XOR sC, FF
            ADD sB, 01
            ADDC sC, 00
_end:       RETURN
```

Násobení větších datových typů (16bitů, 32bitů) je principiálně stejné, jako výše uvedené násobení 8bitových čísel, pouze je náročnější na výpočetní čas i prostor.

Následující algoritmus pro násobení dvou 16bitových čísel již vyžaduje 7 registrů. Registry `sB` až `sE` tvoří jeden větší 32bitový posuvný registr, jež se v každém cyklu posouvá o jedno místo doprava. Polovina tohoto registru (`sD` a `sE`) je na začátku vynulována, druhá polovina (`sB` a `sC`) obsahuje násobitel. V registrech `s8` a `s9` je uložen násobenec, `s7` slouží k odpočítání cyklů. Vzhledem k tomu, že z výsledku bereme pouze nižších 16 bitů (`sB` a `sC`), není třeba rozlišovat znaménkovou a bezznaménkovou variantu.

Násobení 16 bitů * 16 bitů = 32 bitů (16 bitů znaménkově)

```
_mulint:
            LOAD s7, 10           ; 16 cyklů
            LOAD s8, sD
            LOAD s9, sE
            LOAD sD, 00
            LOAD sE, 00

_loop:      TEST sB, 01           ; nejnižší bit == 1
            JUMP Z, _noadd:
            ADD sD, s8
            ADDCY sE, s9

_noadd:     SR0 sE                ; posuv doprava
            SRA sD
            SRA sC
            SRA sB

            SUB s7, 01
            JUMP NZ, _loop
            RETURN
```

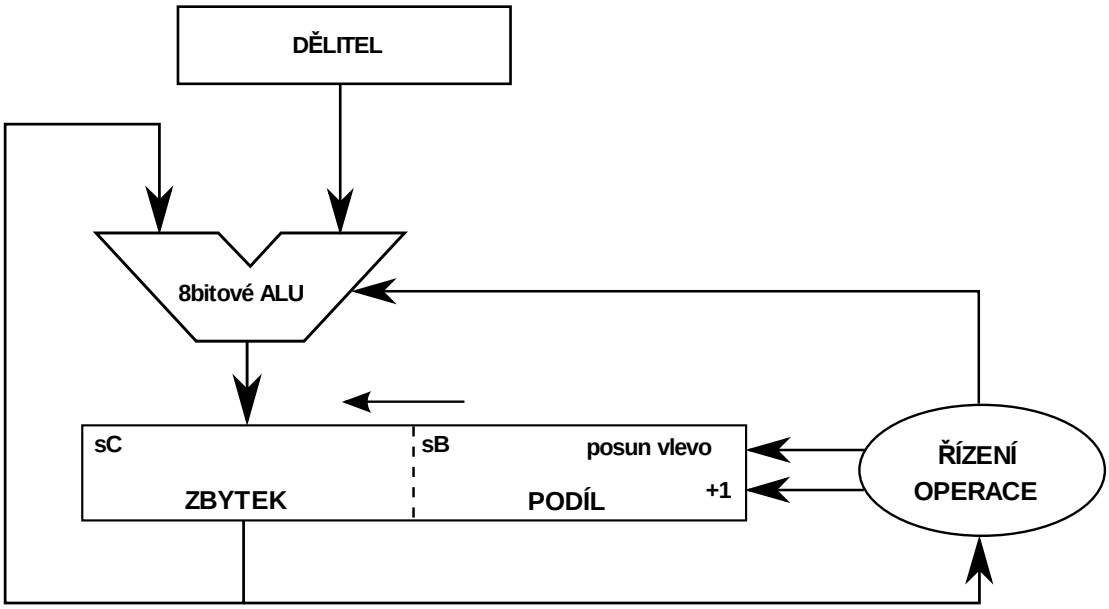
5.2.5 Operace dělení a modulo

Operace dělení a modulo rovněž nemají dedikovanou instrukci a je nutné je provádět dělícím algoritmem popsaném v [3].

5.2.5.1 Algoritmus dělení

Grafické znázornění obvodu, který vydělí dvě kladná čísla bez dedikované instrukce podílu je na obrázku 5. Tento obvod je velmi podobný tomu pro násobení. Rovněž se zde vyskytují 2 registry, kdy jeden je složený ze dvou a na počátku obsahuje jeho nižší polovina dělenec. Druhý, samostatný registr, obsahuje hodnotu dělitele.

Princip obvodu je následující. Na začátku každého kroku se provede aritmetický posuv složeného registru doleva. Pokud je následně hodnota v horní polovině (zbytku) větší nebo rovna děliteli, je od ní odečten a k dolní polovině (podílu) je přičtena 1. Po *X* krocích, kde *X* je bitová šířka registru, je výpočet ukončen s tím, že dolní polovina (`sB`) obsahuje výsledný podíl a horní polovina (`sC`) zbytek po dělení (modulo).



Obrázek 5: Princip algoritmu dělení

5.2.5.2 Implementace algoritmu pomocí instrukcí procesoru PicoBlaze

Dle principu uvedeného v předešlé kapitole byla implementována funkce, která vydělí dvě kladná čísla. Pokud chceme dělit znaménková čísla, musíme postupovat obdobně jako u násobení, a pokud jsou záporné, převést dělenec i dělitel na kladné hodnoty. Podle pravidel dělení (viz Tabulka 2) se poté nastaví výsledná hodnota podílu.

Dělenec	Dělitel	Podíl
+	+	+
+	-	-
-	+	-
-	-	+

Tabulka 2: Vztah mezi znaménky operandů u dělení

Dělení bezznaménkových čísel 8 bitů / 8 bitů = 8 bitů

Následující funkce předpokládá hodnotu dělence v registru `sB`, dělitele v registru `sC`. Dále vyžaduje jeden pomocný registr pro uchování hodnoty dělitele a jeden pro počítání cyklů. Výsledný podíl je v registru `sB` a zbytek po dělení v registru `sC`.

```
_divuchar:
           LOAD sE, 08          ; 8 cyklů
           LOAD sD, sC
           LOAD sC, 00

_loop:     SL0 sB               ; shift left
           SLA sC
           COMPARE sC, sD       ; test sC <= dělitel
           JUMP C, _nosub
           SUB sC, sD
           ADD sB, 01           ; zvětši podíl o 1

_nosub:    SUB sE, 01
           JUMP NZ, _loop
           RETURN
```

Dělení znaménkových čísel 8 bitů / 8 bitů = 8 bitů

Znaménková čísla je nutno nejprve zkontrolovat, zda nejsou záporná, a v tomto případě je převést na kladná. Poté se zavolá funkce pro bezznaménkové dělení celých čísel uvedená výše. Po návratu z této funkce je výsledný podíl v registru `sB` a registr `sC` obsahuje zbytek po dělení. Výsledné znaménko se nastaví dle tabulky 2.

```
_divschar:
           LOAD sA, 00
           TEST sB, 80          ; test dělěnce
           JUMP Z, _nosign1
           XOR sB, FF
           ADD sB, 01
           OR sA, 01

_nosign1:  TEST sC, 80          ; test dělitele
           JUMP Z, _nosign2:
           XOR sC, FF
           ADD sC, 01
           OR sA, 02

_nosign2:  CALL _divuchar       ; dělení bezznaménkových čísel
           TEST sA, 03          ; nastavení výsledného znaménka
           JUMP NC, _end
           XOR sB, FF
           ADD sB, 01
_end:      RETURN
```

Vzhledem k tomu, že funkce pro dělení uvedené v této kapitole produkují jako druhý výsledek i zbytek po dělení (modulo), není třeba pro tuto operaci vytvářet zvlášť funkce. Tím ve výsledku docílíme úspory generovaných instrukcí v cílovém programu, jelikož se modulo často vyskytuje i s dělením. Další výhodou je, že pokud dělení a modulo mají být vypočteny pro stejné vstupní operandy, kdy tyto operace bývají velmi často za sebou, stačí provést pouze jeden výpočet, a tím získáme obě výsledné hodnoty. Ve výsledku pouze stačí prohodit podíl za zbytek, aby bylo modulo v registru `sB` a výš.

Modulo bezznaménkových čísel 8 bitů / 8 bitů = 8 bitů

```
_moduschar:
           CALL _divuchar
           XOR sB, sC      ;swap sB & sC
           XOR sC, sB
           XOR sB, sC
           RETURN
```

U operace modulo se znaménkovými celými čísly je opět potřeba nastavit znaménko výsledku. Závislost mezi znaménky dělence, dělitele a výsledného zbytku po dělení je zobrazena v tabulce 3. Z ní je patrné, že výsledné znaménko je totožné se znaménkem dělence.

Dělenec	Dělitel	Modulo
+	+	+
+	-	+
-	+	-
-	-	-

Tabulka 3: Závislost mezi znaménky operandů u operace modulo

Modulo znaménkových čísel 8 bitů / 8 bitů = 8 bitů

```
_modschar:
           CALL _divschar
           TEST sA, 01
           JUMP Z, _end
           XOR sC, FF
           ADD sC, 01

_end:      XOR sB, sC
           XOR sC, sB
           XOR sB, sC
           RETURN
```

Dělení větších čísel je založeno na stejném algoritmu jako pro 8 bitové čísla. Pouze dochází ke zvětšení registrů na dvojnásobek, respektive čtyřnásobek původních. Zároveň s tím roste počet cyklů nutných pro provedená výpočtu.

Dělení bezznaménkových čísel 16 bitů / 16 bitů = 16 bitů

Následující funkce předpokládá hodnotu dělence v registrech `sB` a `sC`, dělitele v registrech `sD` a `sE`. Dále vyžaduje dva pomocné registry pro uchování hodnoty dělitele a jeden pro počítání cyklů. Výsledný podíl je v registrech `sB` a `sC`, zbytek po dělení v registrech `sD` a `sE`.

```
_divuint:
          LOAD s7, 10           ; 16 cyklů
          LOAD s8, sD
          LOAD s9, sE
          LOAD sD, 00
          LOAD sE, 00

_loop:    SL0 sB                ; shift left
          SLA sC
          SLA sD
          SLA sE

          COMPARE sE, s9        ; zbytek >= dělitel
          JUMP C, _noadd
          COMPARE sD, s8
          JUMP C, _noadd

          SUB sD, s8
          SUBCY sE, s9
          ADD sB, 01

_noadd:   SUB s7, 01
          JUMP NZ, _loop
          RETURN
```

Dělení znaménkových čísel 16 bitů / 16 bitů = 16 bitů

Znaménková čísla je nutno nejprve zkontrolovat, zda nejsou záporná, a v tomto případě je převést na kladná. Poté se zavolá funkce pro bezznaménkové dělení celých 16bitových čísel. Po návratu z této funkce je výsledný podíl v registrech `sB` a `sC`. Výsledné znaménko se nastaví dle tabulky 2.

```
_divsint:
          LOAD sA, 00
          TEST sC, 80           ; test dělence
          JUMP Z, _nosign1
          XOR sB, FF
          XOR sC, FF
          ADD sB, 01
          ADDCY sC, 00
          OR sA, 01

_nosign1: TEST sE, 80           ; test dělitele
          JUMP Z, _nosign2
          XOR sD, FF
          XOR sE, FF
          ADD sD, 01
          ADDCY sE, 00
          OR sA, 02                                          ->
```

```
_nosign2:  CALL _divuint          ; bezznaménkové dělení

           TEST sA, 03            ; nastavení výsledného znaménka
           JUMP NC, _end
           XOR sB, FF
           XOR sC, FF
           ADD sB, 01
           ADDCY sC, 00
_end: RETURN
```

Modulo bezznaménkových čísel 16 bitů /16 bitů = 16 bitů

Operace modulo se liší jen tím, že je potřeba prohodit dvě dvojice registrů.

```
_moduschar:
           CALL _divuint
           XOR sB, sD      ;swap sB & sD
           XOR sD, sB
           XOR sB, sD

           XOR sE, sC      ;swap sC & sE
           XOR sC, sE
           XOR sE, sC

           RETURN
```

Modulo znaménkových čísel 16 bitů /16 bitů = 16 bitů

Operace modulo u znaménkových čísel se také liší jen tím, že je potřeba prohodit dvě dvojice registrů. Výsledné znaménko se nastaví dle tabulky 3.

```
_modsint:
           CALL _divsint
           TEST sA, 01
           JUMP Z, _end
           XOR sD, FF
           XOR sE, FF
           ADD sD, 01
           ADDCY sE, 00

_end:      XOR sB, sD
           XOR sD, sB
           XOR sB, sD

           XOR sE, sC
           XOR sC, sE
           XOR sE, sC
           RETURN
```

Vzhledem k tomu, že dělení 32bitových čísel je principiálně totožné, pouze je zapotřebí více registrů, není třeba jej uvádět.

5.2.6 Větší než - '>' (rL > rR)

Procesor PicoBlaze má pro porovnání dedikovanou instrukci COMPARE, která dle výsledku porovnání odpovídajícím způsobem nastaví příznaky ZERO a CARRY (viz Tabulka 4). Jelikož mezikód očekává výsledek v proměnné `IC_RESULT`, je potřeba nastavit registr `rRes` na 1, pokud je levý operand větší než pravý, v opačném případě nastavit na 0.

Příznak	Příznak = 0	Příznak = 1
ZERO	rL != rR	rL = rR
CARRY	rL >= rR	rL < rR

Tabulka 4: Nastavení příznaků při provedení instrukce COMPARE rl, rR

```
LOAD       rRes, 00

COMPARE    rR, rL
ADDCY      rRes, 00
```

Větší datové typy se již neobejdou bez instrukcí skoku. Porovnávají se odpovídající bajty operandů od nejvyššího po nejnižší s tím, že při nastavení příznaku CARRY na 1 se porovnání muže ukončit a výsledek se nastaví na 1. V opačném případě po ukončení porovnávání zůstane výsledek vynulován.

Následující kód je pro porovnání 16bitových čísel. Pro 32bitová se porovnání provede celkem čtyřikrát.

```
      LOAD    rRes, 00

      COMPARE rR_MSB, rL_MSB
      JUMP C, _gt
      COMPARE rR_LSB, rL_LSB
      JUMP C, _gt

      JUMP    _lt
_gt:  LOAD    rRes, 01
_lt:
```

5.2.7 Menší než - '<' (rL < rR)

Převod se provede analogicky jako u operace **větší než**, Pouze dojde k prohození operandů `rL` a `rR`. Výsledek je uložen do registru `rRes`.

5.2.8 Operace rovno - EQ_OP

Opět se využije instrukce COMPARE, avšak získání příznaku ZERO a uložení do registru `rRes` již vyžaduje instrukci skoku.

```
      LOAD       rRes, 00

      COMPARE    rL, rR
      JUMP NZ, _nz
      LOAD       rRes, 01
_nz:
```

Pro větší datové typy je řešením zopakovat dvojici instrukcí COMPARE a JUMP na všechny bajty operandu. Stačí nalézt jedinou nesouhlasnou dvojici a výsledkem je 0.

5.2.9 Bitové operace AND, OR a XOR

Využívá dedikované instrukci AND, OR a XOR. Aby nedošlo k modifikaci hodnoty `rL`, je potřeba ji před provedením operace načíst do `rRes`. Pro vícebitové datové typy stačí opakovat pro každý bajt.

```
LOAD  res, rL

<instrukce> rRes, rR
```

5.2.10 Logické AND - AND_OP

Tato instrukce reprezentuje logické AND známé z jazyka C, popsané podmínkou:
`IF rL > 0 AND rR > 0 THEN rRes = 1 ELSE rRes = 0.`

```
            LOAD  rRes, 00

            COMPARE rL, 00
            JUMP Z, false

            COMPARE rR, 00
            JUMP Z, false
            LOAD  rRes, 01
false:
```

Výše uvedený kód obsahuje zkrácené vyhodnocování. Pokud neuspěje první podmínka, dojde k přeskočení vyhodnocení druhé podmínky.

Vícebajtové datové typy je třeba před provedením logického AND převést na jeden bajt. Tento převod je velmi jednoduchý, a v podstatě stačí provést bitové OR mezi jednotlivými bajty datového typu a výslednou hodnotu uložit do pomocného registru. Následující kód provede tuto operaci pro dvoubajtové číslo.

```
            LOAD  rRes, rL_MSB
            OR    rRes, rL_LSB
```

5.2.11 Logické OR - OR_OP

Podmínka log. OR: `IF rL > 0 OR rR > 0 THEN rRes = 1 ELSE rRes = 0`

```
            LOAD  rRes, 01

            COMPARE rL, 00
            JUMP NZ, true

            COMPARE rR, 00
            JUMP NZ, true
            LOAD  rRes, 00
true:
```

Stejně jako u operace logické AND obsahuje kód zkrácené vyhodnocování. Pro větší datové typy je postup rovněž stejný jako u logického AND.

5.2.12 Rotace doleva – RLC a doprava - RRC

Lze využít nativní instrukci pro rotaci doleva (RL) a doprava (RR).

```
LOAD  rRes, rL

RL    rRes            (RR rRes)
```

U větších datových typů je nutné zařídit přenos bitů mezi jednotlivými bajty. Pro to lze s výhodou využít instrukcí SRA (SLA) – posun přes CARRY. Před samotnou sekvencí posuvů se musí nastavit CARRY na hodnotu nejnižšího (resp. nejvyššího) bitu datového typu. K tomu lze využít instrukci TEST.

```
TEST  rRes_LSB, 1           (TEST rRes_MSB, 80)

SRA   rRes_LSB              (SLA rRes_LSB)
SRA   rRes_MSB              (SLA rRes_MSB)
```

5.2.13 Levý posuv - LEFT_OP a pravý posuv – RIGHT_OP

Následující popis je pro posuv operandu doleva. Posuv doprava lze provést analogicky záměnou instrukcí za opačnou variantu. Operaci lze rozdělit na dvě možnosti, které mohou nastat:

Známe velikost posuvu

Nejjednodušším řešením za předpokladu, že známe velikost posuvu, je opakovat v cyklu instrukci procesoru PicoBlaze SL0. Pokud máme operand větší než 1B, tak cyklus naroste ještě o instrukce SLA pro každý vyšší bajt datového typu.

```
LOAD  rRes_LSB, rL_LSB
LOAD  rRes_MSB, rL_MSB

; posuv o jedno místo doleva
SL0   rRes_LSB        ; první bajt operandu
SLA   rRes_MSB        ; další bajty operandu pokud > 1B
```

Tento způsob je možné vylepšit, čímž se omezí celkový počet cyklů. Pokud máme operand o velikosti 1B a velikost posuvu je větší než čtyři, lze použít rotaci doprava o počet odpovídající rozdílu osmi a tohoto posuvu. Nakonec se vynuluje stejný počet nejnižších bitů, jako se těchto rotací provedlo.

```
; příklad posuvu o 6 míst doleva -> rotace o 2 doprava
LOAD  rRes, rL

RR    rRes
RR    rRes
AND   rRes, FC        ; maska 11111100
```

U větších datových typů lze omezit počet opakování na maximální hodnotu sedmi. Za každých 8 posuvů lze provést přesun bajtů operandu pomocí instrukce LOAD a vynulování odpovídajícího množství nejnižších bajtů. Pokud velikost posuvu překročí osminásobek celkového počtu bajtů, stačí operand vynulovat.

Například posuv 32bitového operandu o 17 míst doleva může vypadat následovně:

```
; operand je již načten v registrech rRes_0 (LSB) až rRes_3 (MSB)
LOAD rRes_3, rRes_1
LOAD rRes_2, rRes_0
LOAD rRes_1, 00
LOAD rRes_0, 00

SL0 rRes_0
SLA rRes_1
SLA rRes_2
SLA rRes_3
```

Neznáme velikost posuvu

Zde nelze využít kopírování instrukce a je potřeba vytvořit iteraci. Předpokladem je nezáporná velikost `rR`, jinak by bylo potřeba tuto skutečnost kontrolovat. Pro větší datové typy stačí zopakovat instrukci SLA pro každý bajt operandu.

```
            LOAD  rRes, rL
            LOAD  c, rR

loop:       COMPARE c, 00
            JUMP  Z, end

            AND   rRes, rRes      ; nastaveni CARRY na 0
            SLA   rRes            ; posun do leva

            SUB   c, 01
            JUMP loop
end:
```

Pravý posuv se provede podobně jako levý posuv, pouze se zamění instrukce levého posuvu SLA (SL0) za instrukci SRA (SR0). Rovněž lze využít kopírování instrukce při známem posuvu nebo iterativně.

5.2.14 CALL

Volání podprogramu (podprocedury) začínajícího na adrese `rL` (resp. návěští).

```
CALL rL
```

5.2.15 PCALL

Volání podprogramu přes ukazatel. U této instrukce se naráží na jednoduchost instrukční sady a samotného procesoru PicoBlaze. U tohoto procesoru není možné přímo přistupovat k hodnotě instrukčního ukazatele (anglicky Instruction Pointer – IP) ani ho explicitně měnit, což je pro implementaci ukazatele na funkci nezbytné. Proto tato instrukce mezikódu zůstává bez odpovídající implementace, která by přinejmenším vyžadovala modifikaci samotného procesoru.

5.2.16 LABEL

Vygenerování návěští v cílovém asembleru. Jeho název je dán položkou IC_LABEL. Protože dochází ke generování návěští jednak pomocí této instrukce mezikódu a zároveň i u některých složitějších instrukcí při mapování na jazyk symbolických adres, je potřeba hlídat, aby nedošlo ke jmenné kolizi. Návěští se budou generovat podle předem daných pravidel (prefix + číselná hodnota inkrementovaná po vygenerování).

```
_label:
```

5.2.17 GOTO

Tato instrukce provede přímý skok na návěští dané parametrem `_label.`

```
JUMP _label
```

5.2.18 JUMPTABLE

Instrukce mezikódu se chová jako klasická konstrukce switch jazyka C, kterou je skok na návěští dle podmínky. Tato instrukce naráží na stejné problémy jako u PCALL a bez možností explicitního nastavení instrukčního ukazatele ztrácí její implementace smysl. Tuto instrukci lze nahradit sekvencí instrukcí IFX.

5.2.19 IFX

Instrukce mezikódu reprezentující úplný podmíněný příkaz. Pokud je podmínka `cond` splněna (hodnota různá od 0), provede se skok na návěští `_true`, jinak na návěští `_false`.

```
          COMPARE cond, 00
          JUMP Z, zer
          JUMP _false
zer:      JUMP _true
```

5.2.20 Instrukce zásobníku IPUSH a IPOP

Procesor PicoBlaze nemá implementován zásobník a pro něj dedikované instrukce, které ale naštěstí lze simulovat. Operandy je možné ukládat do paměti dat dvěma způsoby:

1. Operandy ukládat sestupně od poslední adresy paměti. Zde může dojít ke kolizi s globálními daty, která se mohou nacházet na nižších adresách. Na druhou stranu je možné v extrémním případě využít plnou velikost paměti pro zásobník.
2. Pro zásobník vyhradit pevnou část paměti. Zde odpadá starost s možnou kolizí s klasickými daty, která se budou nacházet ve svém vlastním paměťovém prostoru. Avšak na druhou stranu je tento způsob paměťově neefektivní, pokud se nevhodně určí velikost paměti pro zásobník. Pro správné určení velikosti zásobníku je navíc nutné program staticky zanalyzovat již při překladu, což nemusí být úplně nejjednodušší, zvláště pokud se v programu vyskytují rekurze.

První způsob se zdá vzhledem k velikosti paměti dat vhodnější. Samotnou implementaci zásobníku lze řešit dvěma způsoby. U prvního způsobu je potřeba vyhradit jeden registr pro ukazatel vrcholu zásobníku a určit druhý, se kterým se bude operovat při načítání/ukládání hodnoty. Samotná

implementace poté spočívá ve volání podprocedur, které provedou načtení/uložení hodnoty z/na adresy dané ukazatelem vrcholu zásobníku. Pro tento ukazatel je vyhrazen registr `sF`, lze jej však případně změnit ve zdrojovém souboru překladu.

```
; inicializace ukazatele na vrchol zásobníku
LOAD sF, 3F

; uložení hodnoty na vrcholu zásobníku
push:
     STORE s0, sF
     SUB sF, 01
RETURN

; získání hodnoty z vrcholu zásobníku
pop:
     ADD sF, 01
     FETCH s0, 01
RETURN
```

Procedura pro získání hodnoty vrcholu zásobníku uvedená výše však neošetřuje možnost volání při prázdném zásobníku, stejně tak vložení hodnoty na plný zásobník (sF = 0), to lze však ošetřit drobnou modifikací:

```
; uložení hodnoty na vrcholu zásobníku
push:
     COMPARE sF, 00
     JUMP Z, _full
     STORE s0, sF
     SUB sF, 01
_full:RETURN

; získání hodnoty z vrcholu zásobníku
pop:
     COMPARE sF, 3F
     JUMP Z, _emp
     ADD sF, 01
     FETCH s0, 01
_emp: RETURN
```

Při provedení procedury nad prázdným zásobníkem tak dostáváme nedefinovanou (původní) hodnotu registru. Pokud se bude hlídat použití operací PUSH a POP, a tím se zamezí vzniku těchto nežádaných událostí, lze použít procedury dle první definice. Tuto kontrolu je možné implementovat s tím, že se při překladu na základě parametru zvolí, zda se má provádět, či nikoliv.

Druhou možností je explicitně vkládat dvojice instrukcí STORE a SUB (respektive ADD a FETCH) přímo do výstupního kódu, aniž by se provádělo odpovídající volání podprogramu. Za cenu možného celkového nárůstu instrukcí tím docílíme efektivnějšího kódu a odpadá nutnost vyhradit druhý registr pro operování se zásobníkem. Této variantě postačuje jediný registr, který slouží jako ukazatel vrcholu zásobníku.

5.2.21 GET_VALUE_AT_ADDRESS

Načte hodnotu ze zadané adresy paměti uložené v registru `rL` do registru `rRes`.

```
FETCH rRes, (rL)
```

Pokud je adresa paměti zadaná konstantou `mem`, tak se načtení provede následovně.

```
FETCH res, mem
```

5.2.22 POINTER_SET

Uloží hodnotu obsaženou v registru `rR` na paměťové místo adresované registrem `rRes`.

```
STORE rR, (rRes)
```

Adresa paměti je zadaná konstantou `mem`.

```
STORE rR, mem
```

5.2.23 ADDRESS_OF

Tato instrukce slouží pro získání adresy paměťového místa. Procesor PicoBlaze nemá žádnou instrukci pro získání adresy dané proměnné. Je potřeba řešit při generování. Překladač musí spravovat adresy globálních a lokálních proměnných.

5.2.24 CAST

Tato instrukce provádí konverze datových typů.

char → short (int)
Pro rozšíření datového typu je potřeba vyhradit pro proměnnou novou paměťovou buňku (registr) a vynulovat. Proměnná se tak skládá ze dvou paměťových buněk (celkem 16 bitů), kdy původní hodnota tvoří nižší bajt. Jelikož se jedná o znaménkovou hodnotu, je potřeba přenést znaménkový bit. V registru `rL` se nachází původní hodnota, v registrech `rRes_msb` a `rRes_lsb` výsledná.

```
LOAD  rRes_msb, 00
LOAD  rRes_lsb, rL

TEST  rL,80
SUBCY rRes_msb, 00
```

short (int) → char
Při této konverzi dochází k ořezání 2bajtové hodnoty o horní (vyšší) bajt. Ve výsledku se jedná o uvolnění tohoto registru. Není potřeba provádět žádné speciální operace. Do výsledného registru pouze stačí přenést nižší bajt.

V případě konverze bezznaménkových (unsigned) hodnot není potřeba řešit přenesení znaménkového bitu, jinak princip zůstává stejný. Princip konverze mezi jinými datovými typy (se stejnou znaménkovostí) je stejný.

znaménkové→bezznaménkové
Po provedení této konverze se výsledná hodnota bere jako bezznaménková a není třeba provádět žádné úpravy.

bezznaménkové → znaménkové
Zde může dojít k situaci, že se výsledné číslo do datového typu nevleze a dojde k jeho přetečení.

5.2.25 Přiřazení '='

Přiřazení hodnoty `val`. Může se jednat o přiřazení hodnoty do registru nebo do konkrétní paměťové buňky. Při větší proměnné než jeden bajt je potřeba operaci iterativně opakovat.

Do registru

```
LOAD res, val
```

Do paměti programu

```
STORE val, (res)
```

5.2.26 GETHBIT

Uloží do registru res nejvyšší bit v dané proměnné. Pokud se jedná o vícebajtový operand, stačí provést pouze na nejvyšším bajtu.

```
LOAD        rRes, 00

COMPARE     rL, 80
ADDCY       rRes, 00
```

5.2.27 SEND a RECV

Instrukce mezikódu SEND a RECV představují v SDCC jedinou možnost, jak předávat parametry funkcím a záleží jen na nás, jak jej implementujeme. Pro předávání parametrů lze využít registry, datový zásobník nebo jejich kombinaci. Předávání přes registry se zdá být efektivnější, vzhledem k tomu, že známe celkový počet a velikost proměnných předávaných funkcím, a vzhledem k nutnosti simulovat zásobník u procesoru PicoBlaze. Na druhou stranu je práce se zásobníkem daleko elegantnější a implementačně jednodušší.

Otázkou tedy je, zda pro předávání parametrů využívat registry či datový zásobník. U zásobníku je potřeba dále vyřešit, zda paměťové místo pro zásobník vyhradit staticky, čímž dojde ke zmenšení již tak malé paměti, či dynamicky.

Nejvýhodnější se zdá pro předávání parametrů vyhradit určitý počet registrů a až dojde k jejich obsazení, tak teprve poté využít datové paměti. Tento způsob je nejefektivnější, jelikož ve většině případů dochází k předávání pouze jednoho, nejvýše dvou parametrů menších datových typů *char* a *int*, na které je v registrech spolehlivě dostatek místa. Pro tento účel je vhodné vyhradit tyto registry speciálně pro tento účel. Nejlepším řešením jsou čtyři registry `sB` až `sE` (vzhledem k tomu, že `sF` bude vyhrazen pro ukazatel vrcholu zásobníku), které jsou rovněž nutné jako návratové a větší datový typ než 4 bajty není podporován. Výhodou je, že tyto operace se nebudou nijak ovlivňovat a lze použít stejnou čtveřici registrů.

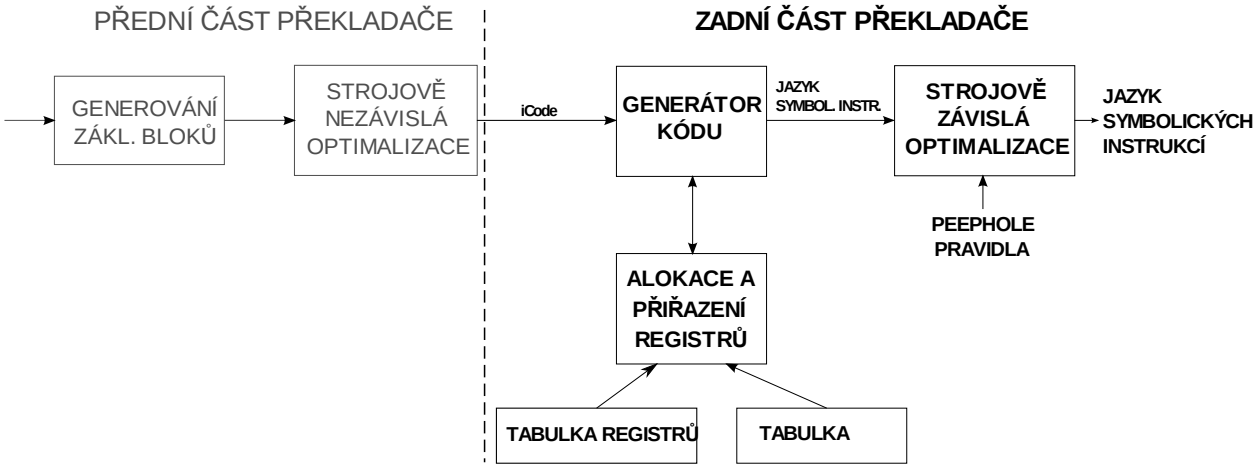
Princip pro jeden parametr typu char:

```
; SEND par (40)
LOAD  sA, 40

; volání funkce CALL f
CALL       f
…

      ; funkce f
      ; RECV par
      FETCH rL, sA
      …
      RETURN
```

5.3 Návrh architektury překladače

Jak již bylo řečeno v dřívějších kapitolách, bude překladač založen na struktuře sestávající se z přední části (front-end), závislé na vstupním jazyce a zadní části (back-end), závislé na cílové architektuře. Překlad neprobíhá přímo, ale je rozdělen do dvou kroků. Nejprve je zdrojový text překládán do mezikódu, a až poté z mezikódu do cílového jazyka. Výhodou této architektury je, že se kompilátor rozdělí na dva jednodušší kompilátory, přičemž překlad ze zdrojového jazyka do mezikódu je společný pro více cílových architektur a až typem zadní části se určuje cílová architektura. Navíc záměnou přední části, při zachování struktury mezikódu, docílíme podporu pro jiný zdrojový jazyk [2].

Návrh struktury zadní části překladače je na obrázku 6, který znázorňuje navázání přední části SDCC (zobrazeno šedou barvou) na navrhovanou zadní část (zobrazeno černou barvou). Spojovacím článkem je mezikód SDCC, označovaný jako iCode (popř. BBlock v implementaci jeho blokové varianty). Samotnou zadní část překladače lze rozdělit na několik dílčích modulů tak, jak je doporučeno v [5]. Vzhledem k jednoduchosti instrukční sady i samotného procesoru PicoBlaze bylo možné modifikovat architekturu zadní části oproti již implementovaným portům u překladače SDCC, a to tak, že se generování kódu a alokace registrů provádí v jednom kroku.



Obrázek 6: architektura zadní části překladače a její navázání na SDCC

Prvním modulem je ‘Generátor kódu‘. Ten na základě typu instrukce definované mezikódem provede vygenerování odpovídající posloupnosti instrukcí jazyka symbolických adres dle převodních pravidel, definovaných v kapitole 5.2.

Druhým modulem je ‘Alokace a přiřazení registrů‘. Jeho smyslem je udržovat informace o obsazenosti registrů a paměti dat konkrétními proměnnými, a na základě těchto poznatků efektivně využívat registry v instrukcích generovaných prvním blokem. Případně při nedostatečném množství registrů provést jejich uvolnění tak, aby byl co možná nejmenší dopad na výslednou efektivitu a výkonnost.

Posledním modulem je ‘Optimalizace cílového kódu‘. Ten má za úkol na základě definovaných optimalizačních pravidel provádět zefektivnění cílového kódu. Tato část je již ve větší míře implementována v rámci SDCC a stačí pouze vhodně zvolit optimalizační pravidla.

Vzhledem k tomu, že je překladač SDCC implementován v jazyce C, bude i zadní část pro procesor PicoBlaze napsána v témže jazyku.

5.3.1 Alokace a přiřazení registrů

Smyslem tohoto modulu je přiřazovat registry operandům instrukcí cílového jazyka, které jsou generované na základě vstupní instrukce mezikódu.

Při přiřazování registrů pro daný operand instrukce může nastat několik situací (vztaženo na 1B operand, při větší velikosti je potřeba provést pro každý bajt operandu):

Operand se již nachází v registru

- Nejlepší varianta, při které není nutné řešit alokaci registrů.

Operand dané instrukce se nenachází v registru, ale všechny registry nejsou obsazeny

- Vybere se první volný registr a provede se nahrání patřičné hodnoty z paměti, nebo registru, případně přiřazení dané hodnoty.

Operand dané instrukce se nenachází v registru, všechny registry jsou obsazeny

- Nejhorší varianta vyžadující provedení analýzy živosti všech registrů a vybrání vhodného kandidáta pro odstranění (hodnota v registru je již mrtvá). Pokud se takový kandidát nenajde, je potřeba vybrat registr, jehož hodnota bude nejdéle nepoužita, či pomocí jiné metody, a jeho hodnotu odložit do paměti dat, čímž dojde k jeho uvolnění pro nový operand.

Aby tyto kroky bylo možné řešit, musí modul udržovat základní informace o registrech, a to v tzv. *Tabulce registrů*, konkrétně:

- Název
- Zda je volný
- Ukazatel na operand
- Zda je rezervovaný pro jiný účel (např. ukazatel vrcholu zásobníku apod.)
- Zda obsahuje globální proměnnou a její hodnota byla modifikována

Informace, které je třeba spravovat pro paměť dat, se shodují s těmi pro registry. Tato struktura se nazývá *Tabulka adres* a obsahuje informace o všech paměťových buňkách.

Další důležitou součástí tohoto modulu by měla být funkce, které se předají jako parametry libovolný operand a jeho konkrétní offset. Na základě těchto parametrů se provede analýza tabulky registrů a adres na situace, které mohou vzniknout (viz výše) a provede se odpovídající reakce:

- Pokud se operand nachází v registru, nic se nestane.
- Pokud se operand nikde nenachází, dojde k jeho alokování.
- Pokud se operand (popř. jeho část) nachází v paměti dat, dojde k přenesení do registru (vygenerují se patřičné instrukce pro načtení dat z paměti).

Funkce vrátí registr s aktuálním umístěním operandu.

5.3.2 Generátor kódu

Funkcí tohoto modulu je na základě mezikódu vygenerovat instrukce jazyka symbolických instrukcí. Základem pro výběr typu zpracovávané instrukce je mezikód. Podle identifikátoru mezikódu se zavolá podprogram, který vygeneruje odpovídající instrukce výstupního jazyka na základě mapování mezikódu na jazyk symbolických instrukcí definovaném v kapitole 5.2. K doplnění registrů za operandy instrukce slouží funkce, kterou poskytuje modul ‘Alokace a přiřazení registrů‘.

6 Implementace zadní části překladače

Následující kapitola popisuje samotnou implementaci zadní části překladače jazyka C pro procesor PicoBlaze, která vychází z návrhu zpracovaného v předešlé kapitole. Implementaci můžeme rozdělit na modul mající na starost alokaci a přiřazení registrů a modul pro generování kódu.

6.1 Vytvoření nového portu

Ještě než bylo možné implementovat zadní část překladače, bylo třeba vytvořit nový port, což se neobešlo bez modifikace konfiguračních souborů a zdrojového kódu přední části.

Prvním krokem bylo vytvořit novou složku v adresáři `sdcc/src` a pojmenovanou podle vytvářeného portu, v našem případě `pblaze`. V té se nacházejí zdrojové soubory zadní části, které jsou dle zvyklostí ostatních SDCC portů pojmenovány následovně a obsahují:

main.c, main.h (sdcc/src/pblaze/main.c)

- Globální struktura typu PORT, která popisuje cílovou platformu (viz kapitola 4.4)
- Funkce pro zpracování parametrů příkazové řádky, které jsou specifické pro konkrétní port
- Inicializační funkce
- Generování tabulky přerušení

ralloc.c, ralloc.h (sdcc/src/pblaze/ralloc.c)

- Funkce pro správu registrů
- Hlavní funkcí, kterou volá přední část pokaždé, když sestaví AST funkce zdrojového programu je:
  ```
  void pblaze_assignRegisters (ebbIndex * ebbi)
  ```

gen.c, gen.h (sdcc/src/pblaze/gen.c)

- Funkce pro generování instrukcí cílového jazyka z mezikódu
- Hlavní funkcí je:
  ```
  void genPBLAZECode (iCode * lic)
  ```

peeph.def (sdcc/src/pblaze/peeph.def)

- Soubor s optimalizačními pravidly

Dalším krokem je modifikace souboru `port.h`, kde se definuje nově vytvářený port, a to na třech místech.

- Začátek souboru, číslo 12 je další v pořadí:
  ```
  #define TARGET_ID_PBLAZE     12
  ```

- Přidání makra pro testování portu
  ```
  #define TARGET_IS_PBLAZE   (port->id == TARGET_ID_PBLAZE)
  ```

- Definice na konci souboru

```
#if !OPT_DISABLE_PBLAZE
extern PORT pblaze_port;
#endif
```

Poslední modifikace zdrojového kódu SDCC je v souboru `SDCCmain.c`. Zde se do struktury `*_ports[]` přídá kód:

```
#if !OPT_DISABLE_PBLAZE
  &pblaze_port,
#endif
```

Nakonec modifikace konfiguračních souborů, kde místa pro modifikaci již nejsou tak patrná. Následující úpravy platí pro verzi `sdcc-src-20110219-6227`, u novějších verzí se již mohou lišit. V souboru `configure` to jsou tyto změny:

- Volby `Optional Features`

```
--disable-hc08-port     Excludes the HC08 port
--disable-pblaze-port   Excludes the PicoBlaze port
--enable-avr-port       Includes the AVR port (disabled by default)
```

- Modifikace přibližně na řádku 610

```
OPT_DISABLE_AVR
OPT_DISABLE_PBLAZE
OPT_DISABLE_HC08
```

- Modifikace přibližně na řádku 710

```
enable_hc08_port
enable_pblaze_port
enable_avr_port
```

- Skript pro testování povolení portu (`# Supported targets`)

```
# Check whether --enable-pblaze-port was given.
if test "${enable_pblaze_port+set}" = set; then :
   enableval=$enable_pblaze_port;
fi

if test "$enable_pblaze_port" = "no"; then
   OPT_DISABLE_PBLAZE=1
else
   enable_pblaze_port="yes"
   OPT_DISABLE_PBLAZE=0
fi

cat >> confdefs.h <<_ACEOF

#define OPT_DISABLE_PBLAZE $OPT_DISABLE_PBLAZE
_ACEOF
echo pblaze >>ports.all

if test $OPT_DISABLE_PBLAZE = 0; then
   echo pblaze >>ports.build
fi
```

- Přibližně na řádku 8000 `# Generating output files`

```
if test $OPT_DISABLE_PBLAZE = 0; then
    ac_config_files="$ac_config_files src/pblaze/Makefile"
fi
```

- Blok `# Handling of arguments`.

```
"device/lib/hc08/Makefile")CONFIG_FILES="$CONFIG_FILES
device/lib/hc08/Makefile" ;;
"src/pblaze/Makefile")CONFIG_FILES="$CONFIG_FILES
src/pblaze/Makefile" ;;
"src/mcs51/Makefile")CONFIG_FILES="$CONFIG_FILES
 src/mcs51/Makefile" ;;
```

- Na konci souboru `ENABLED Ports:`

```
mcs51               ${enable_mcs51_port}
pblaze              ${enable_pblaze_port}
pic14               ${enable_pic14_port}
```

V souboru `configure.in` byly provedeny tyto změny:

- Na konci souboru `# Generating output files`

```
if test $OPT_DISABLE_PBLAZE = 0; then
   AC_CONFIG_FILES([src/pblaze/Makefile])
fi
```

Tím dostáváme nový port, který je defaultně v konfiguračním souboru povolen a spuštěním konfigurace SDCC se pro něj automaticky vytvoří Makefile soubory. Port je poté klasicky dostupný přes parametr `–m<název_portu>`.

6.2 Alokace a přiřazení registrů

Funkce pro správu registrů jsou implementovány v souboru `ralloc.c` a jeho hlavičkového souboru `ralloc.h`. Kromě funkcí jsou zde definovány i struktury pro uchovávání informací o registrech (tabulka registrů) a datové paměti (tabulka adres).

6.2.1 Paměťový model

Paměťový model procesoru PicoBlaze můžeme rozdělit na registry a datovou paměť

6.2.1.1 Registry

Šestnáct registrů procesoru PicoBlaze bylo rozděleno na 3 části:

Registry s0 – sA

Tato skupina tvoří registry pro všeobecné použití (GPR – General Purpose Register), které se účastní všech operací a mohou být přiřazeny konkrétnímu operandu během vykonávání operace. Pokud se vyčerpají volné registry, je to právě z tohoto prostoru, ze kterého se uvolní registry do datové paměti.

Registry sB - sE

Tato skupina registrů plní několik rolí během provádění programu. Hlavním účelem je předávání parametrů volané funkci pro první argumenty, které celkově toto množství čtyř registrů nepřekročí a není nutné je rozdělit (například při předávání dvoubajtového datového typu následovaného čtyřbajtovým se přes registry předá pouze první jmenovaný). Pokud ani toto množství nestačí, je využíván zásobník.

Druhým využitím je předávání návratové hodnoty volající funkci. Vzhledem k tomu, že větší datový typ než 4bajty není podporován, je toto množství dostačující. Návratová hodnota se plní od registru sB, kde se ukládá nejméně významný bajt, až po sE, v případě čtyřbajtové hodnoty s nejvýznamnějším bajtem.

Posledním účelem, kde se tato čtveřice využívá je u hodnot, které se nacházejí pouze v datové paměti. Tato situace nastane, pokud nejsou volné registry a není možné některý z nich uvolnit. V tomto případě se operand alokuje pouze v datové paměti a pro práci s ním se využívají registry právě z této čtveřice. Aby se zamezilo možným konfliktům, je mezi nimi rotováno.

Registr sF

Tento registr slouží pouze jako ukazatel na vrchol zásobníku.

6.2.1.2 Datová paměť

Do datové paměti se zapisuje od nejnižší adresy (0x0) a je typu big-endian. To znamená, že na paměťové místo s nejnižší adresou se uloží nejvíce významný bajt a za něj se ukládají ostatní bajty až po nejméně významný bajt na konci. Vzhledem k velikosti datové paměti je snaha ji využívat co nejméně. Kromě globálních hodnot a operandů s nutností adresace (pole apod.) se využívá pouze při nedostatku registrů pro jejich uvolnění a až v krajním případě pro uložení normálního operandu.

Při simulaci datového zásobníku se s pamětí pracuje od poslední adresy (0x3F) a s každou operací PUSH pro uložení hodnoty na vrchol zásobníku je tato hodnota zmenšena o 1. A naopak při odebrání hodnoty z vrcholu zásobníku (POP) je adresa vrcholu inkrementována.

Díky tomuto modelu je datová paměť využívána co nejefektivněji, ale může nastat situace, že se klasická paměť dostane do konfliktu se zásobníkem. Bez důkladné simulace je zjištění tohoto stavu velice složité (vzhledem k tomu, že datový tok programu může sestávat z několika cyklů a větvení, kdy každý blok může něco ukládat do paměti, případně na zásobník). Tato kontrola je prováděna alespoň staticky, kdy je ověřováno, zda aktuální hodnota vrcholu zásobníku nezasahuje do prostoru, který je již obsazen operandy.

Ještě než se začne generovat výsledný kód, analyzuje se na výskyt globálních proměnných a datových typů s nutností adresace, pro které se rezervuje místo v paměti. Globální proměnné jsou v paměti po celou dobu, dočasné pouze po dobu své životnosti.

6.2.2 Odložení registrů do paměti

Odkládání registrů do paměti je krajní situace, která nastává při jejich nedostatečném množství, kdy je potřeba některý registr vhodně vybrat a přesunout jej do datové paměti.

Tento „vhodný výběr“ byl implementován pomocí strategie nejdéle nepoužitého registru a provádí jej funkce `spillRegsIntoMem`. Vzhledem k tomu, že se odkládá z registrového prostoru během toho, co se již generuje výsledný kód, je potřeba brát ohled na některé možné konflikty. Proto není možné vybírat z celé množiny registrů, ale musí se redukovat o registry, jejichž odložení může konflikt způsobit.

Tyto možné konflikty vznikají zejména při užívání instrukcí pro modelování řízení toku (*if-else*, *for*, *while*). Jedna z možností nastává v případě cyklu, kdy se s operandem pracuje na dvou a více místech, například na začátku a ke konci. Když mezi nimi dojde k situaci, že se registry přiřazené na začátku tomuto operandu odloží do paměti a na konci se pro ten samý operand alokují jiné registry, pak dojde s velkou pravděpodobností v následujícím kroku ke konfliktu, jelikož operand je již v jiných registrech, než se původně nacházel. Podobná situace nastává i při větvení, když se pracuje se stejnými operandy v bloku při splnění podmínky i v bloku při nesplnění podmínky.

Výběr vhodného registru je proto rozdělen na několik kroků. Nejprve se algoritmus snaží nalézt vhodný registr, který vzniká, a případně i zaniká, uvnitř těchto bloků bez řídících instrukcí toku. Pokud algoritmus uspěje, registr se odloží do datové paměti a funkce se ukončí.

Jestliže se v první fázi nenajde vhodný registr, je nutné jej hledat i přes modelovací instrukce toku již v celém programu. Zde se pro každý operand v registru zkontroluje, zda jeho odložení v aktuální pozici generování kódu nezpůsobí nekonzistence, jejichž princip byl popsán v odstavci výše. Ze všech zbylých registrů, které osahují pouze operandy, které tento konflikt nezpůsobí, se poté vybere ten, který bude nejdéle nepoužitý.

Pokud ani po druhé fázi není k dispozici požadované množství registrů, je nutná část operandu alokována v datové paměti a nastaven příznak `isOnlyInMem`, který říká, že daný offset operandu se nachází pouze v datové paměti. Přístup k této proměnné již není optimalizován, a pokud je vyžadována jako jeden z operandů instrukce, je před provedením načtena do jednoho z registru sB až sE a poté, pokud došlo k její modifikaci, ihned uložena do datové paměti zpátky.

6.2.3 Správa registrů a datové paměti

Aby bylo možné zjistit, který registr je volný a který obsahuje některý z operandů, je na globální úrovni vytvořena tabulka registrů, kde má každý registr svou strukturu `regs`. Ta je definována následovně:

```
typedef struct regs
  {
    short rIdx;            /* index do tabulky registrů */
    char *name;            /* jméno registru */
    operand *currOper;     /* aktuální operand v registru */
    short offset;          /* offset tohoto operandu */
    short changed;         /* globální hodnota změněna v registru */
    unsigned isFree:1;     /* registr je volný */
    unsigned isReserved:1; /* registr je rezervovaný */
  } regs;
```

V této struktuře je každému registru nastaveno jméno, které se vkládá do generovaných instrukcí. Dále příznaky volného registru (`isFree`) nebo zda je registr rezervovaný pro jiný účel, než jako GPR registr (`isReserved`). Pokud má operand přiřazen registr, je jeho ukazatel uložen v `currOper` a jeho offset v proměnné stejného jména. Poslední proměnnou je `changed`, která se používá ve spojení s globální proměnnou. Pokud je hodnota globálního operandu přenesena do registru, kde je poté změněna, nastaví se příznak `changed`. Před návratem z funkce, nebo před voláním jiné funkce je její hodnota v datové paměti aktualizována. Pokud ke změně nedojde, nenastaví se ani příznak a hodnotu není třeba aktualizovat. Tento příznak bude detailně popsán v kapitole týkající se generování globálních operandů.

Pro datovou pamět je implementována téměř identická tabulka a princip zůstává stejný jako u tabulky registrů. Příznak changed je zde zbytečný, a není tedy obsažen. Oproti tabulce registru se zde nachází pomocná hodnota `nextPart`, která slouží pro rychlejší přístup k jednotlivým offsetům operandu. Pokud se v datové paměti nachází offsety operandu, které přímo sousedí, je mezi nimi vytvořen jednosměrný seznam až po nejnižší položku. V hodnotě `nextPart` je proto adresa nižšího offsetu, nebo -1 pokud se jedná o nejnižší offset, anebo se v paměti nenachází.

6.2.4 Hlavní funkce modulu

Následující výčet představuje hlavní funkce modulu `ralloc.c`, který má na starost práci s registry. Tyto funkce využívá modul, který má na starost samotné generování výstupního kódu.

aopGetRegName(iCode * ic, operand * op, int offset)

Funkce vrátí jméno registru, kde se nachází zadaný offset operandu `op`. V modulu existuje i podobná funkce `aopGetReg`, která vrací přímo odkaz na registr. Při zpracování nastává jedna ze tří možností, popsaných v kapitole 5.3.1.

- Pokud je offset operandu již v registru, vrací pouze jeho jméno.
- Zjistí-li se, že offset operandu je v datové paměti (`isOffsetInMem`), tak mohou nastat tři možnosti:
  - **Operand byl do paměti přesunut z důvodu nedostatku registrů**
    V tomto případě se vygeneruje instrukce FETCH a operand se přemístí do volného registru, jehož jméno se poté vrátí. Do struktury registru se poznamená, že se v něm operand aktuálně nachází. Místo v paměti, kde se operand původně nacházel, se označí jako volné. Pokud se nepodaří najít volný registr, začne se s operandem pracovat v režimu umístění pouze v datové paměti (`isOnlyInMem`).
  - **Operandem je globální proměnná**
    Zde nastává podobná situace jako v předchozím případě, ale proměnná se zkopíruje a zůstává stále v datové paměti.
  - **Operand je v režimu umístění pouze v datové paměti**
    Tyto operandy se do registrového prostoru přesouvají pouze dočasně, po dobu zpracování instrukce, a ještě do jiné skupiny registrů (sB až sE). S každou instrukcí FETCH se v rámci této skupiny rotuje registr pro načtení hodnoty z datové paměti, čímž je zajištěno, že i kdyby byly všechny tři operandy instrukce v tomto režimu, nevznikne nekonzistence z důvodu opětovného použití registru. I když by stačily pouze tři registry, využívají se všechny čtyři z této skupiny.
- Offset operandu není ani v registrech, ani v datové paměti, a musí být alokován. Tato situace nastává pouze u dočasných proměnných. Pokud je volný registr, nebo se jej povede uvolnit, přiřadí se offset k tomuto registru a je na něj vrácen ukazatel. Pokud se volný registr nenajde, je offset alokován v datové paměti a pracuje se s ním pouze v tomto prostoru a je nastaven příznak `isOnlyInMem`.

aopPutReg(iCode *ic, operand *result, regs *rFrom, int offset)

Tato funkce uloží obsah registru `rFrom` na místo, kde se nachází offset operandu `result`. Pokud se jedná o normální datový typ, zjistí se jeho umístění voláním funkce `aopGetReg` a vygeneruje se instrukce LOAD, která provede požadovanou operaci.

Další možností je, že operand `result` je ukazatelem do datové paměti. V tomto případě se vygeneruje instrukce STORE, která obsah registru `rFrom` na referované místo uloží. Instrukce STORE se vygeneruje i v případě, že se `result` nachází pouze v datové paměti.

aopPutVal(iCode *ic, operand *result, char *val, int offset)

Tato funkce je principiálně shodná s předchozí, pouze s tím rozdílem, že se neukládá obsah registru, ale přímo číselná hodnota `val`.

aopMoveReg(iCode * ic, operand *result, regs *rFrom, int offset)

Funkce asociuje registr `rFrom` konkrétnímu offsetu operandu `result`. Pokud se jedná o registr z GPR skupiny, nastaví se pouze nový operand. Pokud ne, pracuje funkce obdobným způsobem jako `aopPutReg`.

aopUpdateOpInMem(iCode * ic, operand * op, int offset)

Tato funkce je z důvodu ošetření operandů, které jsou pouze v datové paměti, a volá se po každé modifikaci registru během zpracovávání instrukce. Pokud se zjistí, že se jedná o offset operandu s nastaveným příznakem `isOnlyInMem`, vygeneruje se instrukce STORE pro aktualizování její hodnoty v datové paměti. Pro klasické operandy nemá funkce žádný efekt.

6.3 Generátor kódu

Funkce implementované v souboru `gen.c` jsou určeny pro generování posloupnosti instrukcí jazyka symbolických instrukcí procesoru PicoBlaze na základě vstupní instrukce mezikódu.

Hlavní funkcí tohoto bloku je `genPBLAZECode`, která očekává jako parametr seznam instrukcí iCode, pro které se bude provádět generování. Tuto funkcí tvoří cyklus procházející přes všechny předané instrukce, a na základě typu instrukce se zavolá odpovídající funkce, která se postará o generování.

Jak již bylo řečeno, SDCC je navrženo pro komplexnější procesory jako např. PIC16, HC08 nebo MCS51 a samotné generování je zde rozděleno do dvou fází. Prvním průchodem se pouze přiřadí registry jednotlivým instrukcím mezikódu, a až při druhém průchodu se provede vygenerování výstupních instrukcí. Tyto průchody se provádí pro každou funkci jazyka C zvlášť v pořadí shodném s jejich uvedením ve zdrojovém kódu, nikoliv tedy pro zdrojový kód jako celek. To je ovšem při řešení námi navrhnutým způsobem nežádoucí, jelikož už od počátku postrádáme informace o všech globálních proměnných a znemožňuje to efektivně rozvrhnout datovou paměť, které má procesor PicoBlaze k dispozici velmi omezené množství. Řešením bylo při volání zadní části překladače pouze ukládat sekvence *iCode* instrukcí, patřících vždy k jedné samostatné funkci, do obousměrně vázaného seznamu a samotné generování spustit až po zpracování celého zdrojového programu. Bez nutného zásahu do implementace přední části ale informaci, že se jedná o poslední sekvenci *iCode*, nemáme nijak k dispozici. Samotné volání generátoru bylo proto přesunuto až do konečné fáze SDCC (nazvané *glue*), která spojuje všechny vygenerované bloky kódu jazyka symbolických instrukcí do výsledného souboru, čímž jsme se vyhnuli modifikaci přední části překladače.

Zároveň modul *glue*, který je definován v souboru `pbglue.c`, byl značně modifikován a zjednodušen. Generovaný kód původních portů překladače SDCC je rozdělen na více oblastí (*.area*), s čímž si neporadil asembler KCPSM3, který slouží pro finální překlad do strojového kódu.

Dále bylo potřeba specifikovat volání funkcí pro generování operací MUL, DIV a MOD, a pro generování vektoru přerušení.

6.3.1 Volací konvence

Algoritmy psané ve strukturovaných programovacích jazycích a v jejich dalších vývojových stupních, jako například modulární jazyky, kam patří i jazyk C, jsou často děleny na dílčí úlohy neboli funkce. Nastává tedy situace, že v rámci zpracovávání jedné funkce se může pro získání dílčího výpočtu volat jiná. Pokud tuto situaci převedeme na náš problém generování cílového kódu, může dojít k situaci, že registry, a v nich obsažené hodnoty, používané před voláním podprogramu, jsou požadovány i po návratu z něj. Vzhledem k tomu, že každá funkce předpokládá, že má k dispozici celý registrový prostor, je nutné registry dočasně uchovat v datové paměti, na což existují dvě strategie:

Volající funkce spravuje parametry volané funkce

V této strategii (angl. caller-saves) se postará o uchování registrů v datové paměti volající funkce. Vzhledem k tomu, že během generování známe u každé funkce život jejich operandů a obsazenost jednotlivých registrů, můžeme do datové paměti přesunout jen ty, u kterých to je nutné (život operandu začala před voláním funkce a končí až po návratu z ní).

V implementované zadní části překladače je tato možnost implicitní. Před vygenerováním instrukce pro volání podprogramu se projde tabulka registrů a všechny registry, jejichž doba životi přesahuje aktuální, jsou uloženy na zásobník. Následně je vygenerována instrukce CALL, po níž je obnoven stav registrů do původní podoby načtením operandů v opačném pořadí ze zásobníku.

Volaná funkce spravuje své parametry

V této strategii (angl. callee-saves) se o uchování registrů stará až volaná funkce. V rámci implementovaného překladače je třeba tento způsob vynutit pomocí parametru příkazové řádky `--all-callee-saves` pro všechny funkce, nebo existuje i pragma v SDCC, které dokáže také individuálně nastavovat u funkcí tuto vlastnost.

Pokud je tato volba nastavena pro danou funkci, kontroluje se před každým použitím registru, zda ještě nebyl použit, a pokud nebyl, vygeneruje se instrukce pro uložení tohoto registru na zásobník. Zároveň se tato informace poznamená. Před návratem do volané funkce se použité registry vrátí do výchozího stavu.

6.3.2 Generování instrukcí MUL, DIV a MOD

Vzhledem k tomu, že procesor PicoBlaze, jak již bylo řečeno, neposkytuje žádné dedikované instrukce pro násobení, dělení a operaci modulo, jsou tyto operace řešeny pomocí algoritmů využívajících sčítání, odčítání a posuv. Princip těchto algoritmů již byl nastíněn v kapitole 5.2. Tyto instrukce jsou generovány jako volání podprogramu, který vrací výsledek požadované operace. Pro každý podporovaný celočíselný datový typ byla vytvořena hierarchie funkcí (bezznaménkový a znaménkový datový typ), které tyto podprogramy vygenerují do cílového programu. Všechny funkce se nachází v souboru `genmuldiv.c`.

Aby se předešlo zbytečnému vygenerování, je během zpracovávání programu zaznamenáváno, které operace, a s jakými datovými typy nastaly, a úplně na závěr se na konec souboru připojí odpovídající funkce.

Jelikož se jedná o volání funkce, je nutné s ní zacházet tak, jak bylo popsáno v předešlé kapitole 6.3.1, a tedy zachovat registry s operandy, které se využívají i po návratu z podprogramu. Vzhledem

k tomu, že je známo, které registry se účastní výpočtu, stačí zkontrolovat pouze je. Registry jsou přiřazovány v opačném pořadí, než se to děje u běžných funkcí, proto často není nutné zachovávat ani jeden registr, i když se budou používat i po provedení aritmetické operace.

6.3.3 Generování typů s modifikátorem

V této kapitole je popsán princip generování datových typů se speciálními modifikátory *global* a *volatile*.

6.3.3.1 Globální typ

Globální proměnná je speciální datový typ, který je dostupný v jakémkoliv místě zdrojového programu. Tento typ je deklarován a případně i definován mimo tělo funkce. V inicializační fázi zadní části se analyzuje mezikód a vyhledají se všechny globální proměnné, pro které se vyhradí patřičné místo v datové paměti. Pokud se na globální úrovni proměnná i definuje, děje se tak přímo v datové paměti pomocí instrukcí STORE. Jinak je snaha s globální proměnnou pracovat v registrech, a přistup do datové paměti omezit na minimum.

Práce s globální proměnnou je na lokální úrovni uvnitř funkce optimalizována. Pokud první instrukcí, která s touto proměnnou pracuje, je přiřazení hodnoty, není nutné ji načíst z datové paměti, jelikož se tím stává tato hodnota zastaralá. Dokud není nutné globální hodnotu uložit zpět do datové paměti, pracuje se s ní v rámci její lokální kopie v registrovém prostoru. Výjimkou je globální proměnná s modifikátorem *volatile*, který je popsán v následující podkapitole.

Aby překladač věděl, v jakém prostoru se nachází aktuální hodnota, má každá globální proměnná přiřazen příznak `changed`, pomocí kterého se definuje stav globální proměnné:

- Hodnoty jsou stejné.
- V registrech je aktuální hodnota.

Při načtení do registru je implicitně nastaven příznak totožných hodnot. Pokud v lokálním registrovém prostoru dojde ke změně, je odpovídajícím způsobem tento příznak upraven. Před návratem z funkce, nebo voláním jiné funkce, je tento příznak zkontrolován, a pokud došlo ke změně, je vygenerována posloupnost instrukcí pro aktualizování hodnoty v datové paměti. Pokud jsou hodnoty stále totožné, můžeme operand v registrech bezpečně zahodit.

Existuje ještě třetí možnost, kdy je aktuální hodnota v datové paměti. K této situaci může dojít pouze u rutiny obsluhy přerušení, kdy je nezávisle na pozastavené funkci načtena a modifikována hodnota v datové paměti, která v registrovém prostoru nebyla změněna. Kontrola, aby tato situace nenastala, není na překladači, ale na programátorovi. Atomičnost operací nad takovou globální proměnnou si musí programátor zařídit ručně, například přes dočasné vypnutí možnosti přerušení.

6.3.3.2 Volatile

Klíčovým slovem *volatile* označujeme operandy, u kterých nechceme, aby překládač prováděl jakýkoliv druh optimalizace. To znamená, že u globálních proměnných s tímto modifikátorem se neprovádí optimalizace popsána výše.

Překladač se ke *globálním volatile* proměnným chová paranoidně, a do registrů je přesouvá pouze na dobu nutnou k provedení výpočtu. Pokud dojde k její modifikaci, je ihned zapsána zpět do datové paměti. U lokálních *volatile* proměnných se pouze neprovádí jakékoliv optimalizace, a pokud je k dispozici dostatečný počet registrů, ani není třeba s těmito proměnnými pracovat v datové paměti

6.3.4 Generování obsluhy přerušení

Procesor PicoBlaze poskytuje jeden vstup pro externí přerušení. Tomu odpovídá i vektor přerušení, který se nachází na poslední adrese paměti programu (0x3FF). Pokud vznikne událost přerušení, nastaví se instrukční ukazatel právě na tuto adresu, kde poté nejčastěji bývá instrukce CALL pro volání rutiny obsluhy přerušení.

Pokud se během generování narazí na tuto rutinu, je na ní poznamenán odkaz. Úplně na konec generování se poté volá funkce `genIVT`, která vygeneruje na poslední adrese paměti programu skok do poznamenané rutiny, která je typu *callee-saves*. Pokud žádná nebyla definována, nemá funkce žádný efekt. Pokud programátor definuje více rutin přerušení, vybere se ta, která má nejvyšší číslo přerušení (`INTNO`) a zbylé se ignorují. Nejvyšší prioritu má rutina obsluhy přerušení, kde číslo přerušení není uvedeno. Jestliže programátor ve zdrojovém kódu uvede více rutin se shodným číslem přerušení, bere se pouze první uvedená ve zdrojovém programu.

6.3.5 Generování dle zvoleného dialektu

Překladač implementuje možnost volby dvou dialektů jazyka symbolických instrukcí. Prvním je dialekt, který vyžaduje simulátor pBlazeIDE, a zvolíme jej pomocí parametru příkazové řádky `--dialect=pblazeide`. Tato volba je zároveň implicitní, a pokud nezvolíme ani jeden, generuje se kód právě pro tento dialekt. Druhou možností je dialekt pro asembler KCPSM3. Ten zvolíme parametrem příkazové řádky `--dialect=kcpsm3`.

Testování, zda jedním z parametrů příkazové řádky byla volba dialektu, se provádí ve funkci `_pblaze_parseOptions`, která podle výsledku porovnání mezi možnými variantami nastaví globální hodnotu `pblaze_options.dialect`. Rozdíly mezi jednotlivými dialekty jsou popsány v příloze B.

6.3.6 Generování vloženého asembleru

Překladač umožňuje vkládat instrukce vloženého asembleru ze zdrojového programu na odpovídající místo generovaného cílového programu. S instrukcemi se neprovádí žádné úpravy ani kontrola správnosti. Je proto na samotném programátorovi, aby si ověřil, že vložením instrukcí nezpůsobí nekonzistence či úplnou nefunkčnost programu. V tomto vloženém bloku instrukcí se lze odkazovat na globální proměnné v datové paměti, pokud se jednobajtový operand instrukce pojmenuje ve tvaru `_jmeno`, a vícebajtový jako `_jmeno_offset`, kde offset je číslo bajtu operandu.

Následující zdrojový kód demonstruje oba případy globálních proměnných.

```
char g1 = 20;
int g2 = 2000;

void main( void)
{
   while(1) {
#asm
      FETCH s1, _g1
      STORE s1, _g2_0
#endasm
   }
}
```

6.3.7 Zápis ladících informací

Překladač umožňuje pomocí volby `--debug` zapnout zápis ladících informací. Tento zápis probíhá jak do cílového souboru, tak i do pomocného souboru s příponou `adb`. Z něho poté spojovací program (angl. linker) a asembler, které však v rámci této diplomové práce nebyly implementovány, sestaví pomocný ladící soubor s příponou `cdb`. Struktury obou souborů si jsou velmi podobné. Soubor `cdb` má navíc definováno, na jakých adresách v datové paměti se nacházejí konkrétní symboly programu a další pomocné poziční informace, které jsou známy až po sestavení cílového programu. Na tuto oblast nebyl při řešení diplomové práce kladen důraz, proto případné vyřešení problematiky převodu souboru `adb` na `cdb` vyžaduje další a podrobnější výzkum.

Takto vygenerovaný program lze poté ladit pomocí ladících nástrojů a simulátoru *SDCDB*, který je součástí SDCC. Struktura tohoto souboru je detailně popsána v [8].

7 Testování a porovnání

S překladačem, implementovaným v rámci této diplomové práce, byla provedena celá řada testů, které se nacházejí na přiloženém datovém nosiči ve složce /`test`. Testy byly zaměřeny na ověření funkčnosti překladu a to zejména korektnost:

- Aritmetických operací.
- Bitových operací.
- Typových konverzí.
- Instrukcí pro modelování toku (IF-ELSE, cykly).
- Zacházení s globálními a volatile proměnnými.
- Práce s ukazateli.
- Odkládání registrů do paměti.
- Generování obsluhy přerušení.

7.1 Srovnání s překladačem PCComp

V kapitole 2.4.1 byly sepsány základní vlastnosti starší verze překladače PCComp včetně jeho nedostatků. Úkolem této diplomové práce bylo eliminovat alespoň některé analyzované nedostatky. Toto zadání bylo splněno a podařilo se vylepšit či odstranit většinu těchto nedostatků.

V následující tabulce jsou srovnány klíčové vlastnosti obou překladačů:

PCComp	**SDCC + PicoBlaze port (PBCC)**
Bez optimalizace	Optimalizace mezikódu i generovaného cílového programu
Podpora celočíselných datových typů: `char` (1B), `int` (2B)	Podpora celočíselných datových typů: `char` (1B), `short` a `int` (2B), `long` (4B)
Pouze částečná podpora typových konverzí	Úplná podpora typových konverzí mezi podporovanými datovými typy
Podpora pouze jednorozměrných polí	Bez omezení dimenzí polí
Není podporována ukazatelová aritmetika	Částečná podpora ukazatelové aritmetiky (ukazatel na funkce nebylo možno implementovat)
Parametrem funkce nemůže být ukazatel nebo pole	Bez omezení
Globální proměnné je možné definovat pouze uvnitř funkce	Globální funkce je možné deklarovat i definovat na globální úrovni

Závěr

Překladač (či kompilátor) je důležitým prostředníkem během vytváření programů a umožňuje programátorovi popisovat algoritmy pomocí vysokoúrovňových jazyků, které se vyznačují vyšší mírou abstrakce.

Cílem této diplomové práce bylo implementovat překladač jazyka C pro procesor PicoBlaze, konkrétně jeho zadní část. Pro volbu přední části navrhovaného překladače byly uvažovány dva produkty, a to SDCC a LLVM. Na základě porovnání mezikódu a jejich vlastností byl jako vhodný kandidát vybrán SDCC.

Pro procesor PicoBlaze již existuje překladač PCComp, který byl v rané fázi této práce analyzován, a byly zjištěny jeho zásadní nedostatky. Můžeme jmenovat například nemožnost používání ukazatelů, pouze částečná podpora typových konverzí, čí žádná optimalizace.

Překladač implementovány v rámci této diplomové práce tyto nedostatky odstraňuje, v některých případech alespoň částečně, když se narazilo na strop samotné architektury procesoru PicoBlaze a jeho instrukční sady. To se týká například práce s ukazateli na funkce, což nebylo možné, protože procesor neumožňuje explicitní přístup k instrukčnímu ukazateli, včetně nastavení požadované hodnoty.

I když se během překladu provádí celá řada optimalizaci, jedná se pouze o zlomek všech možných. Jedním z možných rozšíření této práce je proto implementace dalších optimalizačních procedur nejen pro mezikód, ale také na výstupní program v rámci peephole optimalizace. Jelikož je peephole již implementována v samotném SDCC, stačí pouze vhodně definovat optimalizační pravidla.

Další možné rozšíření této práce se týká algoritmu výběru vhodného registru pro uvolnění do datové paměti. V této práci byl zvolen princip nejdéle nepoužitého registru, což jistě není nejvýhodnější. Například se může implementovat metoda barvení registrů. Této oblasti se týká i práce s operandy, které jsou umístěny pouze v paměti. Přístup k nim není téměř nijak optimalizovaný a vede k velkému množství zbytečných operací pro přístup do paměti. Tento způsob byl zvolen s ohledem na fakt, že k této situaci dochází pouze občas v krajních případech, když již není možné žádný registr uvolnit. Proto tato oblast nebyla prozatím uspokojivě vyřešena, takže vyžaduje delší výzkum.

Posledním navrženým rozšířením je možnost přístupu z vloženého asembleru na lokální i globální proměnné. Nejlepším řešením je implementovat konečný automat, který rozloží instrukci vloženého asembleru na klíčové slovo operace a jednotlivé operandy. Pokud je některý z operandů zadán jménem, místo konkrétním registrem, ověří se v tabulce symbolů jeho umístění a instrukce se před vygenerováním modifikuje na správný tvar jména operandu, popř. se i změní instrukce načtení z registru na načtení z datové paměti a naopak, podle aktuálního umístění operandu.

Literatura

[1] XILINX: PicoBlaze 8-bit Embedded Microcontroller User Guide v2.0, dostupné na http://www.xilinx.com/picoblaze [online].

[2] Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools, Second Edition. Pearson Education, 2007, ISBN 03-215-4798-5.

[3] Patterson D.A., Hennessy, J.L.: Computer Organization and Design (The Hardware/Software Interface), Fourth Edition. Morgan Kaufmann, 2008, ISBN 01-237-4493-8.

[4] Leupers, R., Marwedel, P.: Retargetable Compiler Technology for Embedded Systems (Tools and Applications). Kluwer, 2001, ISBN 978-1-4419-4928-8.

[5] Holub, A.I.: Compiler Design in C. Prentice Hall, 1990, ISBN 01-315-5045-4.

[6] Meduna, A.: Elements of compiler design. Boca Raton : Auerbach Publications, 2008, ISBN 14-200-6323-5.

[7] PicoBlaze C Compiler User's manual v1.1, dostupné na http://www.docstoc.com/docs/22421511/Picoblaze-C-Compiler [online], [cit. 2011-01-02].

[8] Small Device C Compiler, dostupné na http://sdcc.sourceforge.net [online], [cit. 2011-05-06].

[9] LLVM Compiler, dostupné na http://llvm.org/docs [online], [cit. 2011-01-02].

[10] Wikipedia, The free encyclopedia: Low Level Virtual Machine, dostupné na en.wikipedia.org/wiki/Low_Level_Virtual_Machine, [online], [cit. 2011-01-02].

Příloha A
Instrukční sada procesoru PicoBlaze

Kompletní instrukční sada procesoru PicoBlaze, převzatá z [1].

Instrukce	Popis	Funkce
ADD sX, kk	sečte registr sX s hodnotou kk	sX ← sX + kk
ADD sX, sY	sečte registry sX a sY	sX ← sX + sY
ADDCY sX, kk	sečte registr sX s hodnotou kk a přičte bit přenosu	sX ← sX + kk + CARRY
ADDCY sX, sY (ADDC)	sečte registry sX a sY a přičte bit přenosu	sX ← sX + sY + CARRY
AND sX, kk	bitový AND registru sX a hodnoty kk	sX ← sX AND kk
AND sX, sY	bitový AND registrů sX a sY	sX ← sX AND sY
CALL aaa	nepodmíněné volání podprocedury aaa	TOS← PC PC← aaa
CALL C, aaa	volání podprocedury aaa pokud je nastaven příznak CARRY	If CARRY=1, {TOS ← PC, PC← aaa}
CALL NC, aaa	volání podprocedury aaa pokud není nastaven příznak CARRY	If CARRY=0, {TOS ← PC, PC← aaa}
CALL NZ, aaa	volání podprocedury aaa pokud není nastaven příznak ZERO	If ZERO=0, {TOS ← PC, PC← aaa}
CALL Z, aaa	volání podprocedury aaa pokud je nastaven příznak ZERO	If ZERO=1, {TOS ← PC, PC← aaa}
COMPARE sX, kk (COMP)	porovnání sX s hodnotou kk, dle výsledku nastaví příznaky ZERO a CARRY	If sX=kk, ZERO ← 1 If sX<kk, CARRY← 1
COMPARE sX, sY	porovnání sX s sY, dle výsledku nastaví příznaky ZERO a CARRY	If sX=sY, ZERO ← 1 If sX<sY, CARRY← 1
DISABLE INTERRUPT (DINT)	zakázání přerušení	INTERRUPT_ENABLE ← 0
ENABLE INTERRUPT (EINT)	povolení přerušení	INTERRUPT_ENABLE ← 1
Interrupt Event	asynchronní přerušení	
FETCH sX, (sY) (FETCH sX, sY)	načte obsah scratchpad RAM adresované registrem sY do registru sX	sX← RAM[(sY)]
FETCH sX, ss	načte obsah scratchpad RAM na pozici ss do registru sX	sX← RAM[ss]
INPUT sX, (sY) (IN sX, sY)	načte hodnotu vstupního portu adresovaného registrem sY do registru sX	PORT_ID ← sY sX ← IN_PORT
JUMP aaa	nepodmíněný skok na adresu aaa	PC ← aaa
JUMP C, aaa	podmíněný skok na adresu aaa pokud je nastaven příznak CARRY	If CARRY=1, PC ← aaa
JUMP NC, aaa	podmíněný skok na adresu aaa pokud není nastaven příznak CARRY	If CARRY=0, PC ← aaa

JUMP NZ, aaa	podmíněný skok na adresu aaa pokud je nastaven příznak ZERO	If ZERO=0, PC ← aaa
JUMP Z, aaa	podmíněný skok na adresu aaa pokud není nastaven příznak ZERO	If ZERO=1, PC ← aaa
LOAD sX, kk	uloží do registru sX hodnotu kk	sX ← kk
LOAD sX, sY	uloží do registru sX hodnotu registru sY	sX ← sY
OR sX, kk	bitový OR registru sX a hodnoty kk	sX ← sX OR kk
OR sX, sY	bitový OR registrů sX a sY	sX ← sX OR sY
OUTPUT sX, (sY) (OUT sX, sY)	zapíše obsah registru sX na výstupní port odkazovaný registrem sY	PORT_ID ← sY OUT_PORT ← sX
OUTPUT sX, pp (OUT sX, pp)	zapíše obsah registru sX na výstupní port odkazovaný pp	PORT_ID ← pp OUT_PORT ← sX
RETURN (RET)	návrat z podprogramu	PC ← TOS+1
RETURN C (RET C)	návrat z podprogramu pokud je nastaven příznak CARRY	If CARRY=1, PC ← TOS+1
RETURN NC (RET NC)	návrat z podprogramu pokud není nastaven příznak CARRY	If CARRY=0, PC ← TOS+1
RETURN NZ (RET NZ)	návrat z podprogramu pokud není nastaven příznak ZERO	If ZERO=0, PC ← TOS+1
RETURN Z (RET Z)	návrat z podprogramu pokud je nastaven příznak ZERO	If ZERO=1, PC ← TOS+1
RETURNI DISABLE (RETI DISABLE)	návrat z podprogramu přerušení; přerušení zůstane zakázáno	
RETURNI ENABLE (RETI ENABLE)	návrat z podprogramu přerušení; opětovné povolení přerušení	
RL sX	rotace registru sX doleva	sX ← {sX[6:0],sX[7]} CARRY ← sX[7]
RR sX	rotace registru sX doprava	sX ← {sX[0],sX[7:1]} CARRY ← sX[0]
SL0 sX	posuv registru sX doleva, plnění 0	sX ← {sX[6:0],0} CARRY ← sX[7]
SL1 sX	posuv registru sX doleva, plnění 1	sX ← {sX[6:0],1} CARRY ← sX[7]
SLA sX	posuv sX doleva včetně CARRY	sX ← {sX[6:0],CARRY} CARRY ← sX[7]
SLX sX	posuv sX doleva, bit sX[0] je nezměněn	sX ← {sX[6:0],sX[0]} CARRY ← sX[7]
SR0 sX	posuv registru sX doprava, plnění 0	sX ← {0,sX[7:1]} CARRY ← sX[0]
SR1 sX	posuv registru sX doprava, plnění 1	sX ← {1,sX[7:1]} CARRY ← sX[0]
SRA sX	posuv sX doprava včetně CARRY	sX ← {CARRY,sX[7:1]} CARRY ← sX[0]
SRX sX	aritmetrický posuv sX doprava, bit sX[7] je nezměněn	sX ← {sX[7],sX[7:1]} CARRY ← sX[0]
STORE sX, (sY) (STORE sX, sY)	zapíše obsah registru sX do scratchpad RAM na pozici sY	RAM[(sY)] ← sX
STORE sX, ss	zapíše obsah registru sX do scratchpad RAM na pozici ss	RAM[ss] ← sX
SUB sX, kk	odečte hodnotu kk od registru sX	sX ← sX – kk
SUB sX, sY	odečte hodnotu registru sY od sX	sX ← sX – sY
SUBCY sX, kk	odečte hodnotu kk od registru sX včetně hodnoty CARRY	sX ← sX – kk - CARRY
SUBCY sX, sY	odečte hodnotu registru sY od sX včetně hodnoty CARRY	sX ← sX – sY - CARRY

TEST sX, kk	porovná bity registru sX a hodnoty kk a příslušně nastaví příznaky CARRY a ZERO	If (sX AND kk) = 0, ZERO ← 1 CARRY ← odd parity of (sX AND kk)
TEST sX, sY	porovná bity registrů sX a sY a příslušně nastaví příznaky CARRY a ZERO	If (sX AND sY) = 0, ZERO ← 1 CARRY ← odd parity of (sX AND kk)
XOR sX, kk	bitový XOR registru sX a hodnoty kk	sX ← sX XOR kk
XOR sX, sY	bitový XOR registrů sX a sY	sX ← sX XOR sY

Vysvětlivky k použitým zkratkám:

sX = jeden z 16 možných registrů s0 až sF
sY = jeden z 16 možných registrů s0 až sF
aaa = 10bitová adresa zapsaná v desítkové soustavě
nebo jako tří znaková hexadecimální v rozsahu 000 až 3FF
kk = 8bitová konstanta
PP = 8bitová adresa portu
ss = 6bitová adresa scratchpad RAM
RAM[n] = obsah scratchpad RAM na pozici n
TOS = návratová adresa

Příloha B
Rozdíly mezi dialekty KCPSM3 a pBlaze IDE

Následující tabulky byly převzaty z [1].

Instrukce

Instrukce KCPSM3	Instrukce pBlaze IDE
`RETURN`	`RET`
`RETURN C`	`RET C`
`RETURN NC`	`RET NC`
`RETURN Z`	`RET Z`
`RETURN NZ`	`RET NZ`
`RETURNI ENABLE`	`RETI ENABLE`
`RETURNI DISABLE`	`RETI DISABLE`
`ADDCY`	`ADDC`
`SUBCY`	`SUBC`
`ENABLE INTERRUPT`	`EINT`
`DISABLE INTERRUPT`	`DINT`
`INPUT sX, (sY)`	`IN sX, sY` (bez závorek)
`INPUT sX, kk`	`IN sX, kk`
`OUTPUT sX, (sY)`	`OUT sX, sY` (bez závorek)
`OUTPUT sX, kk`	`OUT sX, kk`
`COMPARE`	`COMP`
`STORE sX, (sY)`	`STORE sX, sY` (bez závorek)
`FETCH sX, (sY)`	`FETCH sX, sY` (bez závorek)

Vysvětlivky k použitým zkratkám:

sX = jeden z 16 možných registrů s0 až sF
sY = jeden z 16 možných registrů s0 až sF
kk = 8bitová konstanta

Direktivy

Funkce	Direktivy KCPSM3	Direktivy pBlaze IDE
Umístění kódu	**ADDRESS** `3FF`	**ORG** `$3FF`
Pojmenování registru	**NAMEREG** `s5, myregname`	`myregname` **EQU** `s5`
Deklarace konstanty	**CONSTANT** `myconstant,`	`80 myconstant` **EQU** `$80`

Příloha C
Parametry příkazového řádku

Následující výčet nezahrnuje všechny možné parametry kompilátoru SDCC, ale pouze ty, které jsou podstatné pro samotný PicoBlaze port.

Parametr	Funkce
`-S`	pouze překlad (nutný)
`-mpblaze`	výběr portu PicoBlaze
`--debug`	zapisuje debug informace do výstupního a pomocného souboru
`--callee-saves <func[,func,...]>`	uvedené funkce uloží registry na zásobník, než je začnou používat
`--all-callee-saves`	všechny funkce ukládají na zásobník registry před jejich prvním použitím v rámci funkce
`--no-c-code-in-asm`	bez komentářů v podobě zdrojového kódu ve výstupním souboru
`--dialect=kcpsm3 (--dialect=pblazeide)`	dialekt dle KCPSM3 nebo pBlazeIDE

Vzhledem k tomu, že asemblování se provádí pomocí externího nástroje KCPSM3, je parametr `-S` při zvoleném PicoBlaze portu nutný.