VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INFORMAČNÍCH SYSTÉMŮ
DEPARTMENT OF INFORMATION SYSTEMS

GENEROVÁNÍ OBJEKTOVÝCH SOUBORŮ PRO RISC-V

GENERATION OF OBJECT FILES FOR RISC-V

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE **Bc. FILIP BENNA**
AUTHOR

VEDOUCÍ PRÁCE **prof. Ing. TOMÁŠ HRUŠKA, CSc.**
SUPERVISOR

BRNO 2017

Zadání diplomové práce/20228/2016/xbenna01

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav informačních systémů Akademický rok 2016/2017

Zadání diplomové práce

Řešitel: **Benna Filip, Bc.**
Obor: Počítačové a vestavěné systémy
Téma: **Generování objektových souborů pro RISC-V**
Generation of Object Files for RISC-V
Kategorie: Překladače

Pokyny:

1. Seznamte se s formátem objektového souboru ELF a funkcí assemblerů a linkerů. Seznamte se s nástroji Codasip Studio pro návrh a optimalizaci procesorů.
2. Navrhněte postup, který umožní generovat objektové soubory kompatibilní s open-source nástroji pro architekturu RISC-V.
3. Implementujte rozšíření navržené v bodu zadání 2.
4. Po dohodě s vedoucím se zaměřte na vytváření objektových souborů pro operační systém Linux. Vyberte vhodná rozšíření a ta implementujte.
5. Vytvořená rozšíření otestujte na dodané testovací sadě. Popište výhody a nevýhody zvolených řešení.

Literatura:

- John R. Levine: Linkers and Loaders, Morgan Kaufmann, 1st edition, 1999.

Při obhajobě semestrální části projektu je požadováno:

- Body 1 a 2.

Podrobné závazné pokyny pro vypracování diplomové práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva diplomové práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap, které byly vyřešeny v rámci dřívějších projektů (30 až 40% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Hruška Tomáš, prof. Ing., CSc.**, UIFS FIT VUT
Konzultant: Husár Adam, Ing., Ph.D., VCIT FIT VUT
Datum zadání: 1. listopadu 2016
Datum odevzdání: 24. května 2017

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav informačních systémů
612 66 Brno, Božetěchova 2

doc. Dr. Ing. Dušan Kolář
vedoucí ústavu

Abstrakt

Tato diplomová práce se zabývá překladem zdrojových souborů programů pro procesorovou architekturu RISC-V. Smyslem rozšíření překladových nástrojů, které je v této práci popsáno, je kompatibilita vzniklých objektových souborů s open source nástroji sady GNU binutils dostupnými pro tuto architekturu. Problematika spočívá především v korektním rozpoznání a následném správném uložení různých typů relokací specifických pro architekturu RISC-V v rámci nástrojů Codasip Studio.

Abstract

This master's thesis deals with the topic of program source code compilation for RISC-V processor architecture. The generated object files need to be compatible with GNU binutils open source tools which are already available for the architecture. The focus is on relocations which must be correctly detected in Codasip Studio tools and transformed into RISC-V platform specific relocation types.

Klíčová slova

Objektový soubor, ELF, RISC-V, Codasip, relokace

Keywords

Object file, ELF, RISC-V, Codasip, relocation

Citace

BENNA, Filip. *Generování objektových souborů pro RISC-V*. Brno, 2017. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Hruška Tomáš.

Generování objektových souborů pro RISC-V

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana prof. Ing. Tomáše Hrušky, CSc. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Filip Benna
23. května 2017

Poděkování

Rád bych poděkoval všem, kteří mi s vytvořením této práce pomohli, především panu Ing. Adamu Husárovi, Ph.D. za odbornou pomoc a cenné rady.

Obsah

Kapitola 1

Úvod

Tato práce se v širším kontextu zabývá procesem překladu zdrojových souborů programu do podoby způsobilé k vykonání. Přesněji je práce soustředěna na část tohoto procesu, která transformuje program v jazyce symbolických instrukcí do binární podoby pro cílovou procesorovou architekturu.

Touto architekturou je v rámci této práce architektura RISC-V. Jedná se o poměrně novou a velmi perspektivní architekturu, která si klade za cíl stát se standardem ve světě vestavěných zařízení a internetu věcí.

Veškeré implementační změny, které jsou popsány v této práci se týkají nástrojů sady Codasip Studio. Codasip Studio umožňuje vygenerovat překladové nástroje pro téměř libovolnou procesorovou architekturu. Významná část této práce se zabývá úpravami Codasip Studio, které vedou k umožnění generování těchto překladových nástrojů právě i pro architekturu RISC-V. Hlavní pozornost je mířena na implementaci relokací, která za cílem podpory této architektury musela být značně vylepšena.

Řešení překladu programů nabízené nástroji Codasip Studio je do jisté míry proprietární. Až na některé výjimky totiž vyžaduje, aby pro celý proces překladu zdrojových souborů pro danou architekturu byly využity nástroje vygenerované pomocí Codasip Studio.

To může způsobit problémy v případě, kdy do překladu vstupují moduly již v binární podobě, které byly přeloženy nástroji od jiných vývojářů. Proto je zbytek práce orientován na změny implementované v nástrojích Codasip Studio, které zajišťují kompatibilitu překladu na úrovni objektového souboru a dovolují tedy využití zdarma dostupných alternativních nástrojů v procesu překladu programů.

Celá tato problematika je postavena na poměrně složitém teoretickém základu. Ten sestává jak ze znalostí principu překladu programů, tak ze znalosti návrhu procesorových architektur. Všechny informace potřebné pro pochopení textu této práce jsou v práci samotné obsaženy a čtenář by tedy měl být postupným pročítáním kapitol schopen pochopit kapitoly následující. Jedinými znalostmi, které jsou očekávány bez snahy je v této práci obsáhnout jsou základy programování a znalost převodu čísel mezi číselnými soustavami.

Kapitola 2 popisuje činnosti překladových nástrojů, které jsou relevantní pro obsah této práce. Kapitola také srovnává nástroje Codasip Studio s open source alternativami ze sady GNU binutils. Kromě překladových nástrojů se kapitola krátce zabývá objektovým formátem ELF.

Kapitola 3 popisuje základní principy návrhu procesorových architektur pomocí nástroje Codasip Studio. Na velmi základní úrovni jsou zde vysvětleny a na ukázkách předvedeny pojmy, které jsou zmiňovány v dalších kapitolách této práce.

Kapitola 4 se zabývá samotnou architekturou RISC-V, které je zásadní pro tuto práci. Jsou zde zjednodušeně popsány některé části specifikace této architektury tak, aby na nich bylo možné stavět v následujících kapitolách.

Kapitola 5 popisuje teoretický přístup k rozšíření nástrojů Codasip Studio za účelem podpory architektury RISC-V včetně zajištění kompatibility s nástroji sady GNU binutils.

Kapitola 6 se soustředí na způsob implementace rozšíření, která jsou popsána předešlou kapitolou.

Kapitola 7 vysvětluje smysl některých úprav, které jsou v rámci této práce popsány.

Kapitola 2

Překladové nástroje a formáty

V této kapitole jsou stručně popsány nástroje účastnící se procesu překladu zdrojových souborů v jazyce C do binární podoby spustitelné na cílové architektuře. Snahou není popsat detailně činnost těchto nástrojů, ale především zmínit jejich vstupy a výstupy a zaměřit se na funkce důležité pro správné pochopení zbytku této práce. To, jak na sebe jednotlivé nástroje navazují a náznaky podoby programu v jednotlivých fázích překladu je ilustrováno obrázkem 2.9.

Kromě samotných nástrojů je zde také popsán formát objektových souborů ELF. Získanou znalost jeho základní organizace čtenář využije v následujících kapitolách.

2.1 Assembler

Assembler je nástroj transformující program v jazyce symbolických instrukcí, který je jeho vstupem, na objektový soubor, který je výstupem assembleru. Během této transformace je textová podoba jednotlivých instrukcí nahrazena binární podobou. Ta je potom s dalšími informacemi, potřebnými pro správnou funkci ostatních nástrojů, uložena právě do objektového souboru. Více informací o instrukčních sadách je možné získat z [8].

Zjednodušeně řečeno assembler prochází vstupní soubor instrukci po instrukci a na základě znalosti binárního formátu každé instrukce transformuje jednotlivé části instrukce do binární podoby a umístí je na správné bitové pozice v nově vytvořeném instrukčním slově.

Jedním z nejjednodušších příkladů tohoto procesu je transformace instrukce *nop*. Tato instrukce nemá žádné operandy, a proto vytvořené instrukční slovo obsahuje pouze operační kód označující, že se jedná právě o instrukci *nop*. Předpokládejme, že vzniklé instrukční slovo je široké 32 bitů a prvních 8 bitů je vyhrazeno pro operační kód. Dále předpokládejme, že operační kód instrukce *nop* je 1. Potom je možné transformaci z textové do binární podoby znázornit obrázkem 2.1.

Obrázek 2.1: Transformace instrukce nop do binární podoby

Většina instrukcí však vyžaduje jeden nebo více operandů. Je zřejmé, že v případě transformace takové instrukce dojde k umístění binárního označení těchto operandů do

instrukčního slova. K demonstraci poslouží obrázek 2.2. Předpokládejme, že operační kód instrukce *jump* je 2 a registr *r10* je binárně značen konstantou 10.

Obrázek 2.2: Transformace instrukce jump do binární podoby

Operand instrukce *jump* se označuje jako nepřímý, protože v instrukčním slově není uložena přímo adresa, na kterou se v rámci programu skočí po vykonání této instrukce, ale je zde uložen registr, ve kterém bude při výkonu programu adresa uložena. Tento přístup však nemusí platit vždy.

Příkladem jiného přístupu může být instrukce *mov*, která slouží k přesunu konstanty do registru. Tentokrát budeme předpokládat, že operační kód takové instrukce je 3. Stejně jako v minulém případě je využit registr *r10*, do kterého je přesunuta konstanta s hexadecimální hodnotou *0xFFF*. Z obrázku 2.3 znázorňujícího tuto transformaci je patrné, že přímý operand v podobě konstanty *0xFFF* je uložen přímo do instrukčního slova.

Obrázek 2.3: Transformace instrukce mov do binární podoby

Ovšem přímý operand nemusí být vždy reprezentován číselnou konstantou. Je zcela přípustné, aby v roli přímého operandu figuroval symbol, tedy jméno označující určitou konstantu nebo adresu v programu. V takovém případě je nutné, aby v průběhu překladu programu byla hodnota tohoto symbolu dohledána a umístěna do binární podoby instrukčního slova stejně, jako tomu bylo v minulém případě.

Toto dohledání může v některých případech proběhnout v assembleru. Hodnota použitého symbolu však může být definována v jiném zdrojovém souboru, než je assemblerem aktuálně zpracovávaný soubor a potom není možné hodnotu symbolu určit. V tomto případě vzniká relokace.

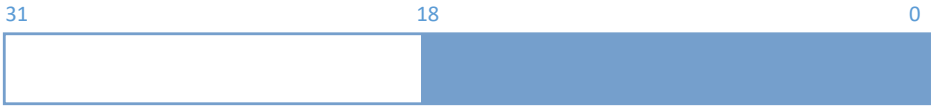
Relokace je reprezentována speciálním záznamem v objektovém souboru a informuje linker o tom, že v assemblerem přeloženém programu, který je také uložen ve stejném objektovém souboru, existuje místo, kam je potřeba doplnit hodnotu symbolu ve chvíli, kdy bude známa. Je dokonce možné, že je v jazyce symbolických instrukcí provedena s hodnotou symbolu nějaká matematické operace. Tato skutečnost musí být v záznamu o relokaci uložena také, aby ji provedl linker ve chvíli, kdy dohledá hodnotu použitého symbolu. Způsob, jakým jsou relokace do objektového souboru uloženy závisí na implementaci překladových nástrojů a jejich vzájemná nekompatibilita mezi nástroji různých tvůrců tvoří hlavní téma této práce.

Jako příklad poslouží znovu instrukce *mov*. Tentokrát však není do registru *r10* přesunuta číselná konstanta, ale hodnota symbolu, která je navíc vymaskovaná (pomocí binární operace *and*) hodnotou *0xFF*. Vzhledem k tomu, že assembler nezná hodnotu sybmolu *a*, uloží do instrukčního slova sekvenci nul. Celá situace je znovu ilustrována, tentokrát obrázkem 2.4.

Obrázek 2.4: Transformace instrukce mov obsahující relokaci do binární podoby

Všechny výše zmíněné instrukce, jejich operační kódy, značení operandů i binární formáty byly zvoleny v co nejjednodušší podobě pro účely vysvětlení této problematiky. Především binární kódování instrukce je v praxi podstatně složitější. Často neplatí, že pořadí operandů v textové podobě instrukce je stejné jako v její binární podobě. Binární kódování instrukce také může nabývat podoby, kdy jednotlivé operandy neobsazují souvislé pole bitů, ale jsou rozděleny na několik úseků v rámci instrukčního slova. Některé z těchto problémů však nebudou v rámci této práce podstatné.

Z hlediska binárního kódování instrukce je pro následující text důležitá především pozice přímého operandu v instrukčním slově. Proto zde zavedeme značení, které barevně označí bitové pozice, které jsou v rámci binární podoby instrukčního slova obsazeny přímým operandem. Na obrázku 2.5 je takto naznačen instrukční formát odpovídající instrukci *mov* z minulého příkladu, kde přímý operand obsazuje spodních 19 bitů instrukčního slova. Takto definované značení bude nadále využíváno v rámci této práce.



Obrázek 2.5: Vyznačení pozice přímého operandu v instrukčním slově

2.2 Linker

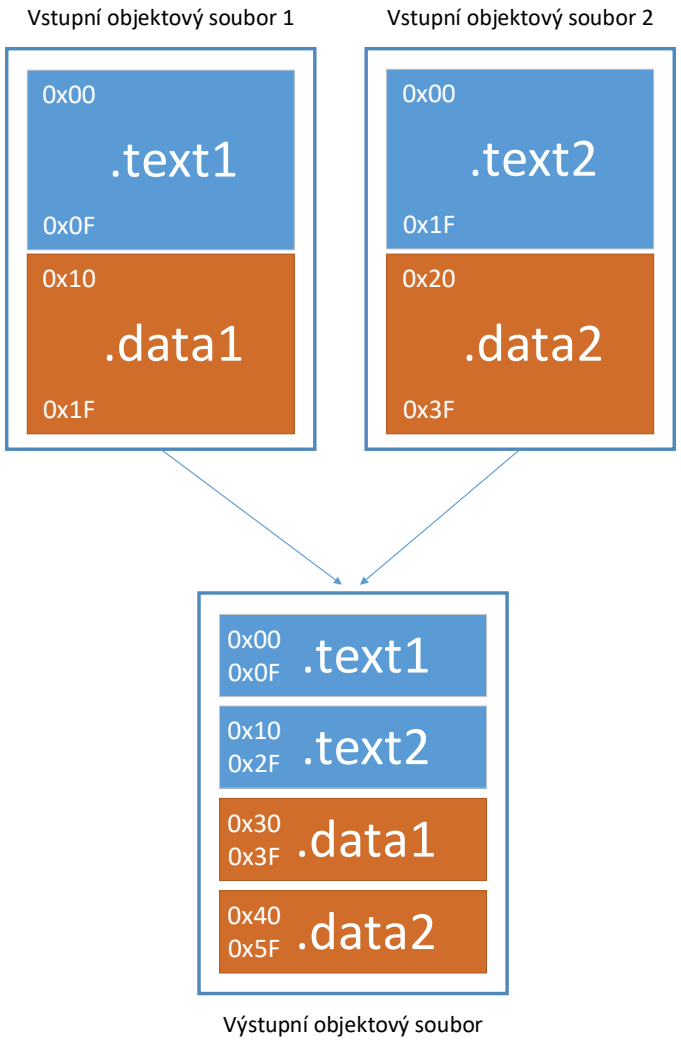
Velmi jednoduše můžeme funkci linkeru popsat jako přiřazení adres abstraktním jménům, symbolům, které se vyskytují v programu. Následně linker provede dosazení těchto hodnot na místa v programu, kde se symboly vyskytovaly a tím v důsledku zjednodušuje tvorbu programů. Pokud tedy chce programátor použít funkci *printf*, nemusí znát její umístění v paměti, stačí mu znát její název.

V začátcích programování byl za přiřazení správné adresy každému použitému symbolu zodpovědný právě programátor. V té době byly vytvářené programy podstatně jednodušší než dnes, a tak to bylo možné, ovšem už tehdy to způsobovalo jisté komplikace. V případě, že se programátor rozhodl program upravit způsobem, který změnil počet instrukcí, musel všechny adresy znovu přepočítat, jelikož se staly neplatnými.

Tento postup však přestal být dostačující umožněním modulárního programování. Díky němu může být jeden program rozdělen do několika zdrojových souborů, které mohou být v paměti umístěny libovolně za sebou. Navíc je možné, aby kód jednoho modulu odkazoval na symbol, který je definovaný v modulu jiném. Všechny tyto problémy řeší nástroj označovaný jako linker [9]. Základní úlohy linkeru jsou tedy dvě:

- Relokace - Překladač nebo assembler produkuje výsledné objektové soubory tak, jakoby každý z nich začínal na adrese 0. Pokud se však program skládá z několika modulů, což je dnes běžné, nemohou všechny začínat na stejné adrese.

Proto je úkolem linkeru upravit přeložený program tak, aby každý z modulů okupoval adresy, které se nepřekrývají. Navíc dochází k řazení sekcí stejného obsahu z různých vstupních objektových souborů za sebe v rámci souboru výstupního. To je ilustrováno obrázkem 2.6 kde jsou sekce obsahující kód programu pojmenovány prefixem *.text* a sekce obsahující data programu pojmenovány prefixem *.data*. Výsledný objektový soubor obsahuje na nižších adresách nejprve kód obou vstupních souborů a teprve potom data.

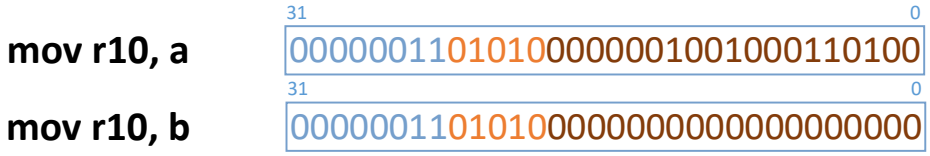


Obrázek 2.6: Změna adres v rámci sekcí způsobená slinkováním dvou modulů

- Rezoluce symbolů - Definice jednotlivých symbolů a jejich použití se mohou nacházet v různých modulech. Při překladu programu v jazyce symbolických instrukcí assemblerem tedy není možné adresu daného symbolu v místě použití určit. Správnou adresu do přeloženého kódu tedy doplní linker, který má povědomí o všech dodaných modulech.

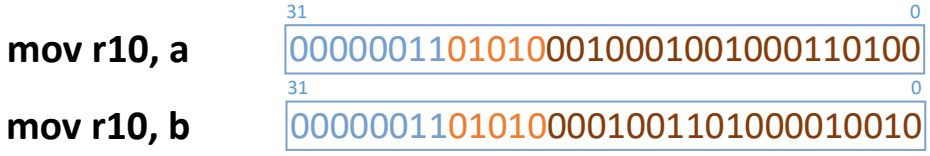
  Princip si ukažme na obrázku 2.7, který navazuje na příklady uvedené v sekci 2.1. Tento kód přesune do registru *r10* symbol *a* a následně symbol *b*. Jelikož je symbol

a definován v rámci stejného modulu jako uvedený kód je jeho adresa assembleru známa a proto je při převodu kódu symbolických instrukcí do binární reprezentace instrukčního slova doplněna. Adresa symbolu *a* je tedy *0x1234*, operační kód instrukce *mov* a označení registru *r10* zůstává stejné, jako tomu bylo v příkladu uvedeném v sekci 2.1. Adresa symbolu *b* není assembleru známa a proto na její místo doplní sekvenci nul.



Obrázek 2.7: Nevyřešená relokace

Linker již adresy všech symbolů samozřejmě zná. Je tedy schopen doplnit adresu symbolu *b*. Ovšem sekce obsahující tento kód byla při linkování posunuta o *10000B* oproti své původní pozici ve vstupním objektovém souboru. Nyní už tedy není správně adresa symbolu *a*. Proto linker upraví i tu. Takto vyřešené relokace jsou znázorněny obrázkem 2.8.



Obrázek 2.8: Vyřešená relokace

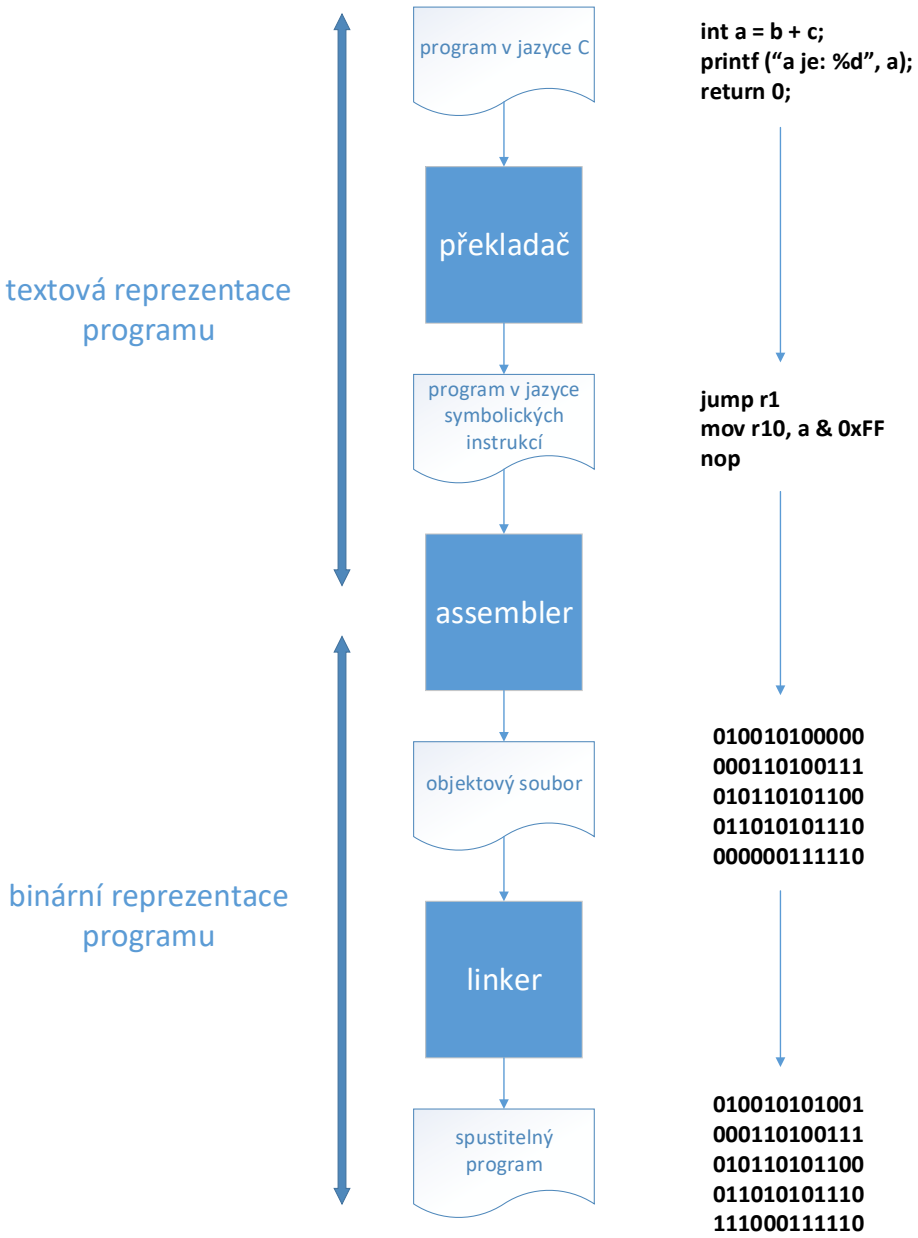
2.3 Objektový formát ELF

Výstupem nástroje assembler a vstupem nástroje linker je objektový soubor (viz obrázek 2.9). Tyto objektové soubory však mohou nabývat několika formátů. Tato práce se bude zabývat pouze formátem označovaným ELF [4]. Dále tedy pojmy objektový soubor a ELF značí to samé.

Obsah souborů formátu ELF je možné rozdělit do několika částí. Ty, které jsou zmiňovány dále v této práci jsou uvedeny zde:

- Hlavička - obsahuje základní informace o datech v souboru uložených, jako je jejich endianita nebo architektura pro kterou byl program přeložen.
- Programové a datové sekce - sekce obsahující kód a data programu.
- Relokace - záznamy jednotlivých relokací obsažených v tomto souboru. Po slinkování tohoto souboru, a tedy vyřešení relokací jsou záznamy ze souboru odstraněny.
- Tabulka symbolů - obsahuje názvy symbolů vyskytujících se v programu obsaženém v tomto objektovém souboru. Ke každému symbolu jsou uložena dodatková data jako je informace, zdali je symbol definován v tomto modulu a případně jeho adresa. Tuto tabulku linker používá při rezoluci symbolů (viz sekce 2.2).

Objektový formát ELF ukládá svůj obsah v binární podobě, kdežto některé alternativní formáty využívají podobu textovou. Není proto možné jeho obsah zkoumat prostým otevřením daného souboru v libovolném editoru. Ke čtení obsahu souborů ELF vzniklo tedy několik nástrojů. Často používaný je nástroj *object dump*, který je většinou označován zkratkou *objdump*.



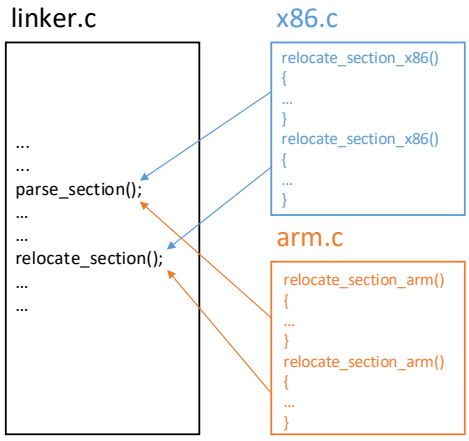
Obrázek 2.9: Překladové nástroje s jejich vstupy a výstupy

2.4 Porovnání dostupných nástrojů

Zbytek této práce se zabývá implementací rozšíření nástrojů Codasip Studio za účelem kompatibility s open source nástroji GNU binutils [5]. Ačkoliv obě ze zmíněných SDK obsahují stejné nástroje, filozofie jejich návrhu je poměrně odlišná. To způsobilo hlavní komplikace dále popsané implementace, a proto bych se rád krátce o těchto rozdílech zmínil.

Je důležité si uvědomit, že obě sady nástrojů mají sice stejný účel, kterým je umožnit překlad programů pro nejrůznější procesorové architektury, v určitých aspektech se liší. Nástroje GNU binutils umožňují přidávat podporu nových architektur na základě implementace sady funkcí, které jsou jasně definovány svým prototypem. Řekněme tedy, že implementace GNU binutils linkeru využívá funkci pro relokování sekce vstupního objektového souboru. Potom vývojář implementující podporu nové architektury musí tuto funkci vytvořit a respektovat jaké vstupní argumenty tato funkce dostane a jakou hodnotu by po svém provedení měla vrátit.

Sada těchto funkcí specifických pro jednu architekturu se v rámci kódu GNU binutils často označuje jako target. Určitá architektura je tedy vzniklými nástroji podporována proto, že byla při překladu zvolena správná sada funkcí z odpovídajícího targetu. Pro ilustraci principu, který je popsaný předchozím odstavcem je přiložen obrázek 2.10. Modul *linker.c* zde reprezentuje obecný kód, který je společný pro všechny architektury. Tento modul využívá funkce implementované specificky pro jednotlivé architektury, na obrázku reprezentovány soubory *x86.c* a *arm.c*.

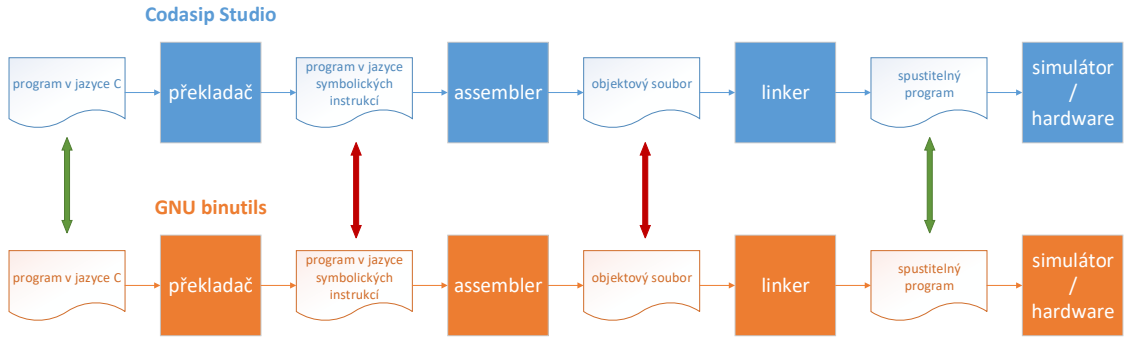


Obrázek 2.10: Ukázka návrhu nástrojů GNU binutils

Hlavní výhodou Codasip Studio je ovšem to, že vývojář implementující novou architekturu není povinen zkoumat implementaci jednotlivých nástrojů SDK a doplňovat do každého z nich potřebné části kódu. Pokud daný nástroj SDK vyžaduje některé části implementované specificky pro určitou architekturu je tento kód vygenerován přímo v Codasip Studio a následně použit při překladu samotného nástroje SDK. Ať už z hlediska rychlosti generování SDK nebo především z hlediska udržovatelnosti kódu Codasip Studio je pro vývojáře Codasip vhodné, aby co největší část kódu byla platformě nezávislá a pouze malé části byly generovány. Z toho vyplývá poté strategie jak v návrhu jednotlivých nástrojů tak ve volbě dostupných knihoven, který bude na konkrétním příkladě vysvětlen dále.

Kompatibilitou nástrojů, která je hlavním tématem této práce rozumíme možnost použít výstupy nástroje jednoho SDK a použít je jako vstup nástroje, který logicky navazuje v řetězci nástrojů spouštěných při překladu, ovšem je součástí jiného SDK. Z toho tedy vyplývá, že kompatibilita mezi dvěma SDK není absolutní, ale je možné určovat ji na několika úrovních.

Kompatibilita na jednotlivých úrovních mezi nástroji Codasip Studio a GNU binutils je ilustrována obrázkem 2.11. Zelené šipky značí úrovně, na kterých nástroje byly kompatibilní již před implementací úprav dokumentovaných touto prací. Kompatibilita na úrovni objektového souboru je předmětem této práce.



Obrázek 2.11: Kompatibilita nástrojů Codasip Studio a GNU bintuils

Nástroje Codasip Studio i GNU binutils budou zmiňovány i v dalších kapitolách. Pro jednoduchost budou jednotlivé nástroje označovány následujícím systémem: pro příklad nástroj assembler ze sady Codasip Studio bude označován jako Codasip assembler a nástroj assembler ze sady GNU binutils bude označován jako GNU assembler.

Kapitola 3

Codasip Studio

Codasip Studio [3] je sada nástrojů uživateli přístupná skrze vývojové prostředí zvané Codasip IDE. V tomto prostředí uživatelé mohou navrhovat procesorové architektury, přičemž Codasip Studio je schopno na základě těchto popisů generovat nástroje, které slouží pro překlad programů pro takto navrženou architekturu. Tyto nástroje jsou často označovány anglickými pojmy toolchain nebo SDK[1]. Oba tyto pojmy budou dále využívány v tomto textu.

Kromě nástrojů je uživateli vygenerován i RTL popis čipu, verifikační prostředí a jiná data potřebná při návrhu procesorů. Tyto další položky však v rámci této práce nejsou podstatné, a proto jsou zde jen stručně zmíněny.

Kompletní návrh procesoru prostřednictvím Codasip Studio vyžaduje dva popisy navrhované architektury. Nástroje pro překlad programů jsou vygenerovány na základě popisu na úrovni instrukcí [2]. Z tohoto důvodu bude dále zmiňován právě jen tento druh popisu. Popis je vytvořen v jazyce CodAL [1], což je HDL[3] jazyk [10] syntaxí připomínající jazyk C. Jelikož samotný popis slouží jako model pro vytvoření výše popsaných nástrojů a dalších výstupů, je popis procesoru v jazyce CodAL označován jako model dané architektury. Takto bude označován i v dalších částech tohoto textu.

Základním stavebním prvkem je *element*. Jeden *element* může popisovat jak samotnou instrukci, tak její součásti, kterými jsou operandy v jednotlivých instrukcích figurující.

Ukázka kódu 3.1: Element popisující instrukci lui v jazyce Codal

```
element i_lui
{
        use opc_lui;  // operacni kod instrukce
        use gpreg;    // cilovy registr
        use uimm16;   // zdrojovy operand

        assembler { opc_lui gpreg "," uimm16 }; // lui r1, 0x1
        binary { opc_lui gpreg uimm16 };

        semantics
        {
                // umisteni primeho operandu do horni poloviny registru gpreg
                rf_gpr[gpreg] = (uint32)uimm16 << 16;
        };
};
```

[1] *Software development kit*
[2] *instruction accurate*
[3] *Hardware description language* - jazyk pro popis hardware

Ukázka obsahuje definici instrukce *lui*, která slouží pro načtení 16bitové konstanty do horní poloviny 32bitového registru. Sémanticky se tedy jedná o poměrně jednoduchou instrukci a je proto vhodná pro vysvětlení popisu instrukcí v jazyce CodAL. Dále jsou stručně vysvětleny jednotlivé části tohoto popisu.

- Název - Název elementu *i_lui* je používán při odkazování se na tuto instrukci v rámci popisu procesoru. Nemusí mít tedy nic společného s označením dané instrukce v jazyce symbolických instrukcí.
- Direktiva *use* - Tato direktiva se používá pro deklaraci jiných elementů, které jsou použity v rámci definice aktuálního elementu. V případě elementu *i_lui* se tedy jedná o operační kód instrukce *lui*, element popisující registr obecného použití a element definující 16bitový neznaménkový přímý operand.
- Sekce *assembler* - Tato sekce popisuje textovou podobu instrukce v jazyce symbolických instrukcí s využitím dříve deklarovaných elementů. V tomto případě se jedná o operační kód instrukce následovaný cílovým registrem, který je čárkou oddělen od neznaménkové konstanty. Tato sekce je důležitá pro generování assembleru, především jeho části parsující vstupní soubor v jazyce symbolických instrukcí.
- Sekce *binary* - Tato sekce popisuje binární podobu instrukce, která je assemblerem vygenerována v případě, že je tato instrukce nalezena při parsování vstupního souboru v jazyce symbolických instrukcí (princip je vysvětlen v sekci 2.1). V tomto případě je horní část binární podoby instrukčního slova obsazena operačním kódem instrukce. Následuje identifikace cílového registru a nejméně významové bity instrukce reprezentují přímý operand.
- Sekce *semantics* - Sekce popisující samotný význam této instrukce. Je tedy využita především překladačem, který na jejím základě usuzuje, k čemu je možné instrukci využít a samozřejmě také simulátorem, který při výskytu této instrukce v programu vykoná akce v této sekci popsané. V případě elementu *i_lui* se tedy jedná o posunutí přímého operandu o 16 bitů doleva a jeho následné zapsání do cílového registru.

3.1 Kódování operandů

Předcházející úsek vysvětloval použití konstrukce *element* k popisu instrukce v jazyce CodAL. Pro další kapitoly této práce je však důležitější pochopení definice přímého operandu s použitím stejné konstrukce. Následuje ukázka popisu 16bitového neznaménkového operandu, který figuroval i v ukázce použité v předchozí kapitole.

Ukázka kódu 3.2: Element popisující 16bitový neznaménkový operand

```
element uimm16
{
        // neznamenkovy operand o bitove sirce 16
        unsigned attribute bit[16] val

        assembler { val };
        binary { val };
        return { val };
};
```

- Název - Název elementu je použit pro odkazování se na daný typ operandu v rámci popisu procesoru. Typicky je využit v rámci elementů popisujících instrukce, které daný operand využívají.
- Znaménkovost a bitová šířka - Element *uimm16* definuje 16bitový znaménkový operand. Nejjednodušší typy operandů jsou definovány prakticky jen na základě bitové šířky a znaménkovosti.
- Sekce *assembler* - Stejně jako v případě definice instrukce popisuje textovou podobu operandu v jazyce symbolických instrukcí.
- Sekce *binary* - Stejně jako v případě definice instrukce popisuje binární podobu operandu.
- Sekce *return* - Označuje hodnotu, která je vrácena v případě, že je tento operand odkazován v rámci sekce semantics obsažené v definici nějaké instrukce.

Poslední tři sekce nejsou pro pochopení následujícího textu důležité a jsou tedy zmíněny jen pro úplnost. V případě zájmu o podrobnější vysvětlení definic jak instrukcí, tak operandů je možné nahlédnout do [2].

Uvedený operand *uimm16* je absolutní. Opakem jsou operandy relativní, které se oproti těm absolutním liší tím, že je jejich hodnota relativní vzhledem k programovému čítači. V kontextu elementu popisujícího takový operand to znamená přidání dvou dalších sekcí popisu.

Ukázka kódu 3.3: Element popisující 16bitový relativní znaménkový operand

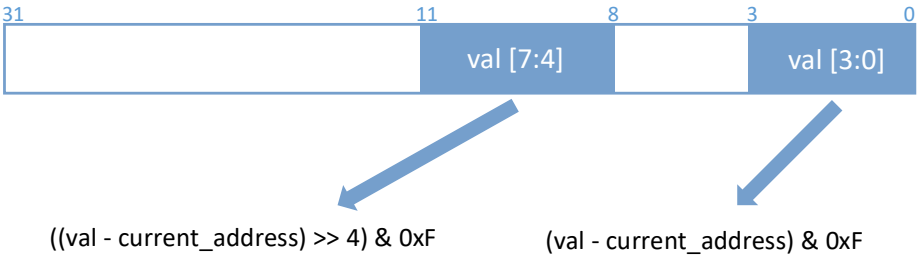
```
element rel_imm16
{
        // znamenkovy operand o bitove sirce 16
        signed attribute bit[16] val
        {
                // od hodnoty operandu val je odectena hodnota programoveho citace
                encoding = val - current_address;
                decoding = val + current_address;
        };

        assembler { val };
        binary { val };
        return { val };
};
```

- Sekce *encoding* - Tato sekce popisuje kódování operandu. Uživatelem definované kódování využívá assembler a to k tomu, aby určil, jakým způsobem má být hodnota operandu uložena v objektovém souboru. Může se jednat například o posunutí hodnoty doprava v případě, kdy je známo, že je operand zarovnaný na určitý počet bitů nebo o uložení hodnoty operandu relativně k aktuálnímu programovému čítači.

  Element *rel_imm16* popisuje právě operand, který je relativní vzhledem k programovému čítači, čehož je docíleno odečtením hodnoty tohoto čítače od hodnoty operandu. Při vytvoření relokace pro tento operand (viz sekce 2.1) assembler do záznamu v objektovém souboru ukládá i informace obsažené v popisu kódování operandu.
- Sekce *decoding* - Sekce popisující dekódování operandu. V rámci této práce je možné dekódování považovat za operaci inverzní k operaci kódování.

Problematika vytváření relokací pro přímé operandy se však může zkomplikovat ještě více. Důležitým vlivem je také formát instrukce obsahující relokovaný operand, přesněji umístění operandu v rámci instrukčního slova.

V jednoduchém případě přímý operand okupuje souvislé pole bitů, ve kterém je uložen od nejvýznamnějšího bitu po ten nejméně významný (případně v opačném pořadí). Toto pravidlo však neplatí obecně. Z důvodu jednoduššího návrhu hardware dekodéru[4] instrukcí často dochází k rozdělení přímého operandu v rámci instrukčního slova do několika úseků. V takovém případě se pro každý takto vzniklý úsek přímého operandu generuje v assembleru nová relokace. Tato relokace v sobě však musí mít zakódován i posun v rámci samotného operandu, jelikož linker již nemá žádnou informaci o tom, že určitá skupina relokací spolu souvisí (ilustrováno obrázkem 3.1).



Obrázek 3.1: Vliv umístění operandu v instrukčním slově na vytvořené relokace

3.2 Formát relokací

Každá relokace vytvořená při zpracování zdrojového souboru assemblerem je v objektovém souboru reprezentována jedním záznamem. Ačkoliv samotný objektový soubor je ve formátu ELF (tento formát je popsán v sekci 2.3), který je specifikován tak, aby byla zajištěna alespoň částečná kompatibilita nástrojů od různých tvůrců s tímto formátem pracujících, samotný formát uložená relokace nabízí jistou volnost.

Proto většinou platí, že je nutné při překladu programu využít jak assembler, tak linker od stejného tvůrce, jelikož implementují kompatibilní zápis a čtení záznamů o relokacích. Tento záznam v případě nástrojů generovaných s využitím Codasip Studio obsahuje několik položek, které jsou důležité pro pochopení následujících kapitol této práce, a proto budou níže vysvětleny.

Ačkoliv to zřejmě není zcela intuitivní, neplatí, že by pro každou instrukci byla vytvořena maximálně jedna relokace. V případě, že je operand v rámci binární podoby instrukčního slova rozdělen do několika částí, je pak assemblerem generován stejný počet relokací.

Jako příklad může posloužit instrukce s umístěním přímého operandu na základě obrázku 3.1 (pro pochopení viz sekci 2.1). Pokud bude instrukce s tímto binárním formátem využívat operand *rel_imm16* popsaný v sekci 3.1, budou pro takovýto operand vytvořeny dvě relokace, přičemž každá z nich musí linker informovat o tom, se kterou částí operandu pracuje. Toho se dá docílit pomocí operací posunutí a vymaskování správné části operandu.

Tyto operace je navíc nutné zkombinovat s operacemi již uvedenými v popisu kódování operandu v jazyce CodAL, které jsou součástí modelu procesoru. Výsledné výrazy definující

[4] jednotka procesoru starající se o rozdělení právě načtené instrukce do jednotlivých významových částí

jednotlivé části relokovaného operandu jsou taktéž zobrazeny v obrázku 3.1. Výrazy jsou poté společně s názvem symbolu uloženy assemblerem do jednotlivých položek relokačního záznamu. Popis těchto záznamů následuje dále.

Relokace používané v nástrojích Codasip se dělí podle dvou kritérií. Prvním z nich je dělení na znaménkové a neznaménkové. Který typ má být použit vyplývá z definice operandu v modelu procesoru (viz sekci 3.1). V případě znaménkového operandu rozděleného v instrukčním slově do několika částí je znaménková relokace generována pouze pro tu část, která obsahuje znaménkový bit, ostatní části jsou neznaménkové.

Druhé kritérium dělí relokace na absolutní a relativní. Absolutní relokace ukládají hodnotu symbolu do výsledné binární podoby instrukčního slova přímo, kdežto relativní relokace symbol ukládají relativně vzhledem k aktuální hodnotě programového čítače. Tento rozdíl byl popsán již ve sekci 3.1.

Obě tyto vlastnosti jsou do záznamu relokace uloženy v podobě položky, kterou označujeme typem relokace. Je tedy patrné, že pro obsažení všech možností musí být typy celkem čtyři. Na následující ukázce výpisu nástroje objdump (funkce tohoto nástroje byla popsána v sekci 2.3) jsou vidět jak relativní relokace symbolizované typem *R_CODASIP_REL*, tak relokace absolutní typu *R_CODASIP_ABS*. Případná koncovka *_S* [5] v názvu typu označuje, že se jedná o znaménkovou relokaci.

Ve výpisu je patrné, že relokace symbolu *a* je rozdělena celkově do čtyř relokací. Důvod proč se toto stalo je popsán v sekci 3.1. Všechny čtyři relokace jsou relativní, přičemž jen ta obsahující nejvyšší významový bit je znaménková.

Oproti tomu relokace symbolu *b* je reprezentována jen jedním záznamem a je absolutní.

```
0050 R_CODASIP_REL_S   0x000280f83e3
    (bc: 4, msb:  31, lsb:  31, add:   0, shr:  20, msk:0x0000) a
0050 R_CODASIP_REL     0x04018060263
    (bc: 4, msb:  19, lsb:  12, add:   0, shr:  12, msk:0x0ff) a
0050 R_CODASIP_REL     0x008160a0283
    (bc: 4, msb:  20, lsb:  20, add:   0, shr:  11, msk:0x0001) a
0050 R_CODASIP_REL     0x050020a83c3
    (bc: 4, msb:  30, lsb:  21, add:   0, shr:   1, msk:0x03ff) a
0010 R_CODASIP_ABS     0x000000003e3
    (bc: 4, msb:  31, lsb:   0, add:   0, shr:   0, msk:0x0000) b
```

Dále je v každém záznamu relokace vyhrazeno 64 bitů pro dodatečné informace, které specifikují operace, které je nutné se získanou hodnotou symbolu v linkeru provést tak, aby byla hodnota symbolu do instrukčního slova správně zakódována a také v rámci instrukčního slova umístěna na správné bitové pozice. Tyto informace jsou uvedeny níže, v závorkách je bitová šířka vymezená pro danou položku a odpovídající označení ve výpisu nástroje objdump.

- Počet bajtů (5, *bc*) - Určuje přes kolik bajtů se daná relokace rozkládá. Až na některé speciální případy, které budou v dalších částech textu vysvětleny, odpovídá velikosti instrukčního slova.
- Nejvyšší významový bit (10, *msb*) - nejvyšší bit, označován MSB, relokace v rámci instrukčního slova.
- Nejnižší významový bit (10, *lsb*) - nejnižší bit, označován LSB, relokace v rámci instrukčního slova.

[5] z anglického *signed*

- Addend (19, *add*) - Definuje konstantu přičtenou k hodnotě symbolu před jejím uložením do instrukčního slova.
- Bitové posunutí[6] doprava (10, *shr*) - Definuje velikost bitového posunutí hodnoty symbolu před jejím uložením do instrukčního slova.
- Maska (10, *msk*) - Definuje masku aplikovanou na hodnotu symbolu před jejím uložením do instrukčního slova.

Poslední položkou záznamu relokace je samotný relokovaný symbol.

[6] *shift*

Kapitola 4

RISC-V

RISC-V [7] je procesorová architektura specifikovaná instrukční sadou - ISA[1]. To znamená, že je pro tuto architekturu definována standardizovaná instrukční sada, kterou se všechny implementace zavazují podporovat. Ovšem samotná hardwarová implementace již standardizovaná není, a proto si ji může každý potenciální výrobce vytvořit podle svého uvážení.

V současné době je běžné, že každý výrobce některé z používaných procesorových architektur navrhuje pro tuto architekturu novou instrukční sadu. Pokud se však stane, že tento výrobce přestane čipy dané architektury vyrábět, stanou se programy pro tuto instrukční sadu přeložené bezcennými, protože je nebude možné spustit na čipech jiných výrobců, které s největší pravděpodobností využívají instrukční sadou jinou. Záměrem asociace RISC-V Foundation, stojící za touto ISA, je poskytnout specifikaci instrukční sady zdarma, tím motivovat různé výrobce hardwaru, aby jejich nové čipy implementovaly právě instrukční sadu RISC-V a tím zajistili větší kompatibilitu programů mezi čipy různých výrobců.

Základní instrukční sada, kterou se zabývá i tato práce, je tedy tzv. *zmražená* a v budoucnu nesmí probíhat její změny, které by mohly pokazit zpětnou kompatibilitu. Samozřejmě se však očekává, že potřeby instrukčních sad se mohou v budoucnu vyvíjet, a proto je možné, aby RISC-V procesory kromě základních instrukcí podporovaly také instrukční rozšíření.

Tato rozšíření kromě možnosti vylepšovat instrukční sadu v budoucnosti nabízí také možnost škálování hardwarové implementace čipu. Nejjednodušší implementace čipu RISC-V kupříkladu nepracují s čísly s plovoucí desetinnou čárkou [2], proto také nepodporují instrukce pro práci s tímto formátem dat, které jsou součástí jednoho z balíčků instrukčních rozšíření. Ovšem i takto jednoduchá hardwarová implementace musí podporovat základní instrukční sadu RISC-V.

Z tohoto důvodu byla základní instrukční sada navrhnuta tak, aby dovolovala právě co nejjednodušší hardwarovou realizaci, což vyústilo v některé komplikace, které musí řešit nástroje starající se o překlad programů pro tuto ISA. Některé z těchto komplikací jsou zmíněny v dalších částech tohoto textu.

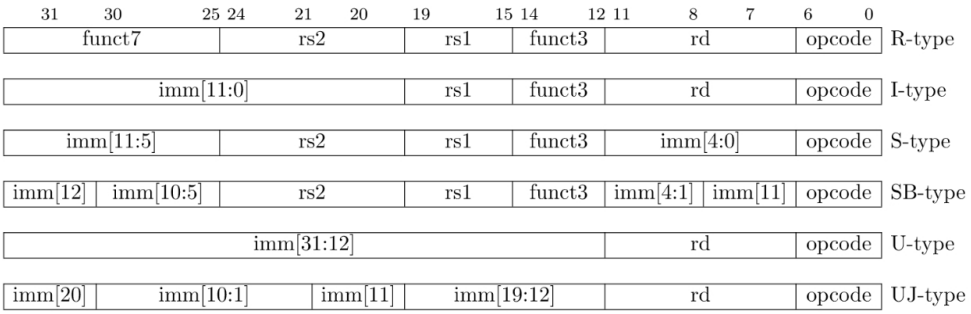
4.1 Základní instrukční sada

Základní instrukční sada procesoru RISC-V obsahuje čtyři formáty instrukcí [11]. Všechny formáty mají pevnou délku 32 bitů a formát specifikuje význam jednotlivých bitů každého

[1] *Instruction Set Architecture*
[2] *floating-point*

instrukčního slova. Typicky je část instrukce vyhrazena pro operační kód, zbytek je pak rozdělen mezi operandy instrukce, kterými jsou nejčastěji registry nebo přímé operandy[3].

Formáty instrukcí se označují písmeny *R*, *I*, *S* a *U*, ovšem první z nich není v tuto chvíli zajímavý, protože nepopisuje instrukce pracující s přímými operandy. Navíc varianty *S* a *U* obsahují ještě dvě alternativy *SB* respektive *UJ*. Tyto alternativy zachovávají umístění přímého operandu v rámci instrukčního slova, liší se však významem jednotlivých bitů přímého operandu. Obrázek 4.1 graficky znázorňuje tyto formáty.



Obrázek 4.1: Instrukční formáty základní instrukční sady RISC-V [11]

Přímý operand ve formátu instrukcí *SB* obsahuje cíle skoků, jejichž adresy jsou zarovnané, a proto není nutné do instrukce ukládat nejméně významový bit (je vždy nulový). Operand však není v rámci instrukčního slova zakódován po prostém posunutí o jeden bit doprava, jelikož při návrhu instrukčních formátu byl kladen vliv na to, aby se významy jednotlivých bitů v různých formátech v co největším počtu shodovaly a tím bylo docíleno zjednodušení dekodéru.

Tato snaha je patrná především na příkladu nejvyššího významového, a tedy znaménkového, bitu přímého operandu. Ve všech instrukčních formátech je umístěn na stejném bitu instrukčního slova. Obdobná snaha je patrná i u velké části dalších bitů přímého operandu. To má však za následek, že se operand v rámci instrukce rozpadl ve formátu *SB* na čtyři části z původních dvou ve formátu *S*.

Formáty *J* a *UJ* se od sebe liší tím. že formát *J* ukládá operand posunutý o 12 bitů doprava, kdežto formát *UJ* operand posunutý doprava o pouhý jeden bit. Stejně jako v případě formátů *S* a *SB* byla při návrhu snaha o největší překrytí významu jednotlivých bitů v obou instrukčních formátech. Tentokrát to má však za následek rozpad přímého operandu z původně souvislého pole bitů ve formátu *J* na čtyři části ve formátu *UJ*.

4.2 Komprimovaná instrukční sada

Kromě základní instrukční sady, která byla popsána v předešlé kapitole definuje specifikace architektury RISC-V i standardní instrukční rozšíření. Jedná se o sady instrukcí, které nejsou nutné pro dostatečnou obecnost procesoru, ale mohou značně urychlit výpočet některých programů nebo zmenšit velikost kódu programu. Většina z těchto rozšíření nejsou pro následující text podstatná, protože nepřidávají nové instrukční formáty.

Instrukční sada označovaná písmenem C seskupuje sadu komprimovaných instrukcí. Tyto instrukce se oproti těm standardním liší především tím, že instrukční slovo je velké pouze 16 bitů oproti 32 bitům instrukčního slova standardní instrukční sady.

[3] *immediate*

Sémantika každé komprimované instrukce odpovídá sémantice některé instrukce základní instrukční sady. Rozdíl dvou takto sémanticky stejných instrukcí, kdy jedna je součástí standardní instrukční sady a druhá té komprimované, je v tom, kolik bitů instrukčního slova je vyhrazeno pro operandy. Pokud se jedná o přímý operand, znamená to, že komprimovaná instrukce dovoluje menší rozsah hodnot přímého operandu. V případě registrového operandu to znamená to samé a projeví se to tak, že operandem komprimované instrukce mohou být jen některé registry.

Za cenu těchto omezení je však ušetřeno 16 bitů z velikosti programu, což je rozdíl velikostí těchto dvou instrukcí. Pokud je tedy přímý operand instrukce dostatečně malý, případně je možné využít vhodný registrový operand, je použití komprimované instrukce výhodné. Formáty instrukcí komprimované instrukční sady jsou uvedeny na obrázku 4.2

Format	Meaning	15 14 13	12	11 10	9 8 7	6 5	4 3 2	1 0
CR	Register	funct4		rd/rs1		rs2		op
CI	Immediate	funct3	imm	rd/rs1		imm		op
CSS	Stack-relative Store	funct3	imm			rs2		op
CIW	Wide Immediate	funct3	imm				rd′	op
CL	Load	funct3	imm		rs1′	imm	rd′	op
CS	Store	funct3	imm		rs1′	imm	rs2′	op
CB	Branch	funct3	offset		rs1′	offset		op
CJ	Jump	funct3	jump target					op

Obrázek 4.2: Instrukční formáty komprimované instrukční sady RISC-V [11]

4.3 Formát relokací

Za standard RISC-V relokací je považována jejich implementace v open source nástrojích sady GNU binutils [5]. Tato implementace však používá jiný přístup než je tomu u nástrojů Codasip Studio (viz sekci 3.2). Nesnaží se snížit počet typů relokací a informace ke správnému zakódování přímého operandu přidat do záznamu relokace přímo v objektovém souboru. Naopak, typ relokce je kromě názvu relokovaného symbolu jedinou informací popisující danou relokaci.

V záznamu relokace v objektovém souboru tedy nenajdeme informace o bitových pozicích operandu, velikosti relokované paměti ani o případném posunutí operandu nebo jiné operaci s ním provedené. To má za následek větší množství typů relokací, jelikož každý způsob relokování musí mít svůj vlastní typ. Navíc je nutné, aby operace prováděné při relokování operandu podle určitého typu relokace byly přímo naprogramovány ve zdrojových kódech linkeru.

To, jakým typem relokace je v kontextu nástrojů GNU binutils relokována daná instrukce, respektive její přímý operand, je dáno zjednodušeně dvěma vlastnostmi.

- S každým formátem instrukce (který obsahuje přímý operand) se váže jeden a více typů relokace. To znamená, že na základě typu relokace je možné přesně určit na kterých bitech instrukčního slova se přímý operand nachází. Toto je vlastnost GNU relokací, které využívá GNU linker k tomu, aby poznal se kterými bity instrukčního slova má provádět případné transformace.

- Ne vždy je instrukční formát svázaný s jedním typem relokace. Pokud tomu tak není, je typ odlišen ještě tím, v jaké podobě je uložen v instrukčním slově přímý operand. Přímý operand může být do instrukčního slova uložen ve své absolutní podobě nebo relativně. Nejčastěji je relativní podoba určena vzhledem k programovému čítači, ale může to být i jiný registr, jako například registr označovaný *global pointer*.

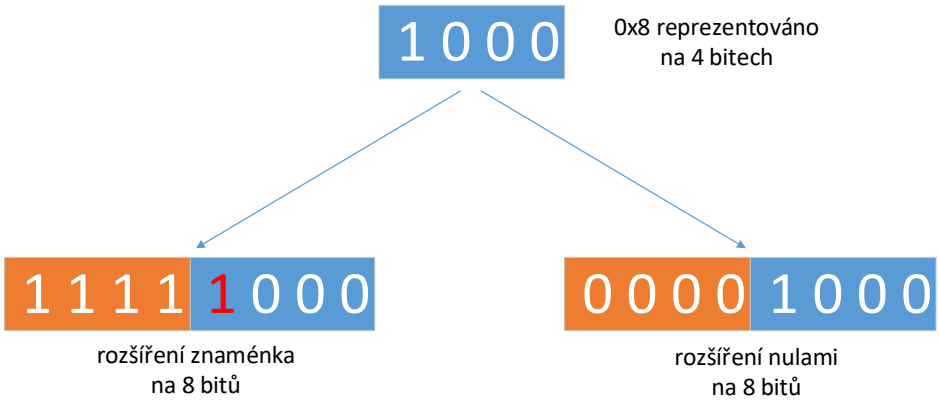
Jako příklad mohou posloužit dva typy relokací označované *R_RISCV_LO12_I* a *R_RISCV_PCREL_LO12_I*. Oba uvedené typy se váží s instrukcemi formátu *I* (formáty instrukcí byly popsány v sekci 4.1). Odlišuje je však to, že první jmenovaný popisuje operand instrukce uložený ve své absolutní podobě, kdežto druhý popisuje operand uložený relativně vůči pozici programového čítače.

Některé typy relokací se navíc sdružují do dvojic. To se děje z toho důvodu, že obstarávají relokaci toho stejného symbolu, každá z nich však obsahuje jeho jinou část, relokovanou v rámci jiné instrukce. Tato situace nastává z toho důvodu, že v případě architektury se šířkou slova 32 bitů není možné do jednoho instrukčního slova uložit celý 32bitový operand. Řešením je použití dvou různých instrukcí, kdy se jedna stará o horní část operandu a druhá o tu spodní. Ovšem díky návrhu instrukční sady RISC-V je tato situace v kontextu relokací o něco složitější.

Pro pochopení zbytku kapitoly je nutné vysvětlit rozdíl mezi neznaménkovým a znaménkovým rozšířením. Proces rozšíření obecně spočívá v navýšení bitové šířky, na které je reprezentována určitá hodnota (naznačeno obrázkem 4.3).

Neznaménkové rozšíření je možné označit také jako rozšíření nulami[4] a nově získané bitové pozice tedy naplní nulami nehledě na rozšiřovanou hodnotu. V případě znaménkového rozšíření[5] je do horních bitů rozšířené hodnoty kopírován znaménkový, tedy nejvyšší významový, bit původní hodnoty.

Z toho vyplývá, že v případě, kdy je znaménkový bit nulový, není mezi těmito dvěma způsoby rozšíření žádný rozdíl s respektem na výsledek rozšíření. Ovšem znaménkové rozšíření umožňuje naplnění nově vzniklých bitových pozic jedničkami v případě, že je znaménkový bit ještě nerozšířené hodnoty taktéž jedna. Rozdíl výsledku při použití obou rozšíření je také patrný z obrázku 4.3.



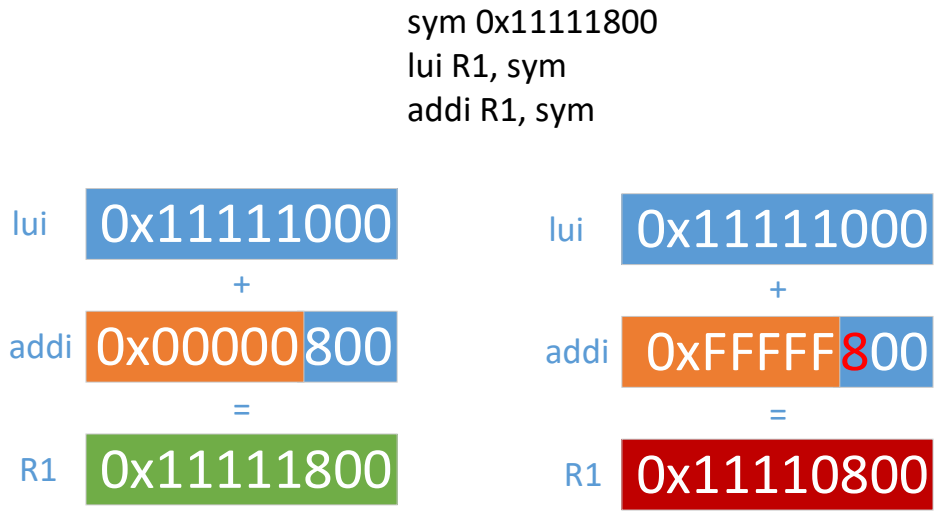
Obrázek 4.3: Rozšíření znaménka a rozšíření nulami

[4] zero extend
[5] sign extend

Jako příklad, kdy potřeba rozšíření hodnoty operandu způsobuje komplikace, použijme načtení 32bitové konstanty do registru. Tuto situaci je bezpochyby nutné obstarat dvěma instrukcemi. Instrukce *lui* se postará o naplnění horních 20 bitů registru. Následně je pomocí instrukce *addi* k registru přičtena hodnota vyjadřující spodních 12 bitů. Zde ovšem nastane problém. Během sčítání je nutné oněch 12 bitů rozšířit na 32 tak, aby odpovídala bitová šířka sčítaného registru a konstanty. Pokud dojde k rozšíření nulami, bude výsledná hodnota v registru správná.

Tato situace tedy směřuje k tomu, že procesor musí při dekódování instrukce rozhodnout o způsobu rozšíření přímých operandů této instrukce. Nelze totiž počítat s tím, že by každý operand byl rozšířen nulami. Znaménkové rozšíření je důležité pro správnou funkci jiných operací a je tedy nutné, aby jej procesor implementoval. Rozhodnutí, které rozšíření použít však komplikuje hardware dekodéru instrukcí. Proto se tvůrci instrukční sady rozhodli, že ji navrhnou tak, aby bylo možné rozšiřovat všechny operandy stejně, a to rozšířením znaménka. Situace, kdy je toto rozšíření nevhodné a způsobilo by nekorektnost výsledku musí být tedy ošetřeny jinak.

Vliv různých způsobů rozšíření přímého operandu na výslednou hodnotu operandu načtenou v registru je možné pochopit z obrázku 4.4. První případ znázorňuje použití rozšíření nulami. V tomto případě je hodnota načtená do registru *R1* správná. Ovšem jak bylo uvedeno dříve, v případě RISC-V dojde k použití rozšíření znaménka. Kritická je tedy hodnota nejvyššího významového bitu spodní části hodnoty symbolu *sym*. Jelikož je to v tomto případě jedna, dochází nad operandem instrukce *addi* k rozšíření právě této jedničky do horních dvaceti bitů. Tímto se z nově vzniklé 32bitové hodnoty stává záporné číslo[6] a výsledek načtený do registru *R1* není správný.



Obrázek 4.4: Vliv rozšíření operandu při načítání 32bitové konstanty do registru

Je patrné, že musí existovat způsob jak tento problém řešit, jinak by instrukční sada RISC-V nebyla dostatečně obecná. Pokud je onen kritický bit nastaven do hodnoty jedna, je možné přičíst k horní části operandu, která figuruje v instrukci *lui*, konstantu, která vyváží chybu vzniklou rozšířením spodní části operandu v instrukci *addi*. Tuto úpravu lze

[6] nejvyšší a tedy znaménkový bit je jedna

samozřejmě provést až ve chvíli, kdy je hodnota symbolu známa, což se stane při linkování. Z toho důvodu je pro tento případ definován vlastní typ relokace, který linkeru sděluje, aby zmiňovaný problém ošetřil.

Obrázek 4.5: Správné ošetření nevhodného znaménkového rozšíření

Kapitola 5

Návrh

Následující kapitola se zabývá návrhem změn, které je nutné provést v nástrojích Codasip Studio za účelem umožnění správného překladu zdrojových souborů pro architekturu RISC-V. Celkový pohled na tyto změny je rozdělen do dvou částí. Obě tyto části se však zabývají nejdůležitější změnou a tou je implementace relokací.

První se zabývá obecnou implementací, která umožňuje definovat velmi široké spektrum výrazů popisujících relokovaný operand. Toto řešení umožní překlad programů pro RISC-V za předpokladu, že všechny nástroje použitého toolchainu jsou ze sady Codasip Studio.

Druhé řešení se soustředí na dodržení formátu relokací použitého v nástrojích GNU binutils dostupných pro architekturu RISC-V. Dodržením toho formátu je možné kombinovat nástroje Codasip s nástroji získanými z jiných zdrojů.

5.1 Kódování operandů

Výrazy popisující operandy instrukcí architektury RISC-V používají složitější operace, než jaké jsou běžně používané v relokacích nástrojů Codasip Studio. Ty jsou limitovány jednak množinou operací, ale také bitovou šířkou jejich operandů, které je možné uložit do záznamu relokace (viz sekci 3.2). Navíc RISC-V relokace využívají při výpočtu hodnoty relokovaného symbolu i jiné registry než programový čítač používaný i v Codasip relokacích.

Z tohoto důvodu je nutné povolit složitější výrazy popisující kódování operandů v modelu procesoru. Díky této úpravě je možné popsat mimo jiné i operand instrukce *lui*, jehož hodnota se modifikuje na základě hodnoty jednoho konkrétního bitu tohoto operandu. Tato modifikace byla vysvětlena v sekci 4.3. Jedna z možností jak tento operand popsat je znázorněna na následující ukázce s využitím ternárního operátoru.

Ukázka kódu 5.1: Kódování operandu využívající ternární operátor

```
unsigned attribute bit[20] val
{
    encoding =
        (val & 0x800) ? ((val + 0x1000) >> 12 & 0xFFFFF) : (val >> 12 & 0xFFFFF);
    decoding = val;
};
```

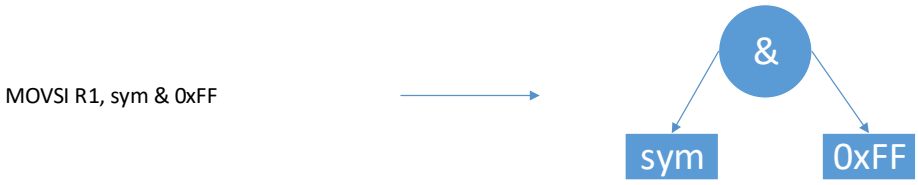
5.2 Univerzální relokace

Umožnění složitějšího popisu operandů v modelu procesoru vede k nutnosti lepší reprezentace výrazů popisujících daný operand, a to jak pro účely tohoto textu tak následně samotné implementace v nástrojích Codasip Studio.

Vhodnou reprezentací těchto výrazů je strom, který je vytvořen assemblerem v několika krocích. První část stromu je vytvořena na základě definice kódování operandu na úrovni modelu procesoru (pro ilustraci naznačeno na posunutí obrázkem 5.1).

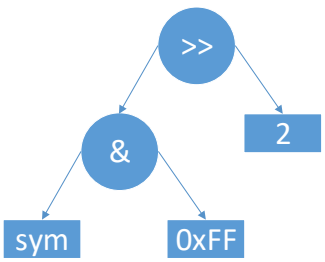
Obrázek 5.1: Stromová reprezentace kódování operandu

Ovšem kódování není jediné místo, které ovlivní výslednou binární podobu hodnoty operandu v rámci instrukčního slova. Určité operace, mezi které patří především maskování, může na operand aplikovat i překladač (ilustrováno obrázkem 5.2). Stejně tak pořád dochází ke štěpení relokací vlivem rozdělení operandu v instrukčním slově (viz sekci 3.1).



Obrázek 5.2: Stromová reprezentace maskování operandu

Všechny tyto operace musí assembler poskládat do výsledného stromu tak, aby bylo stále patrné, v jakém pořadí mají být operace na operand aplikovány. Výsledný strom kromě klasických binárních operací a případné ternární podmínky obsahuje také bitové šířky výsledků jednotlivých podvýrazů (výsledná podoba ilustrována obrázkem 5.3, pro přehlednost bez aplikace operátorů změny bitové šířky).



Obrázek 5.3: Výsledná stromová reprezentace

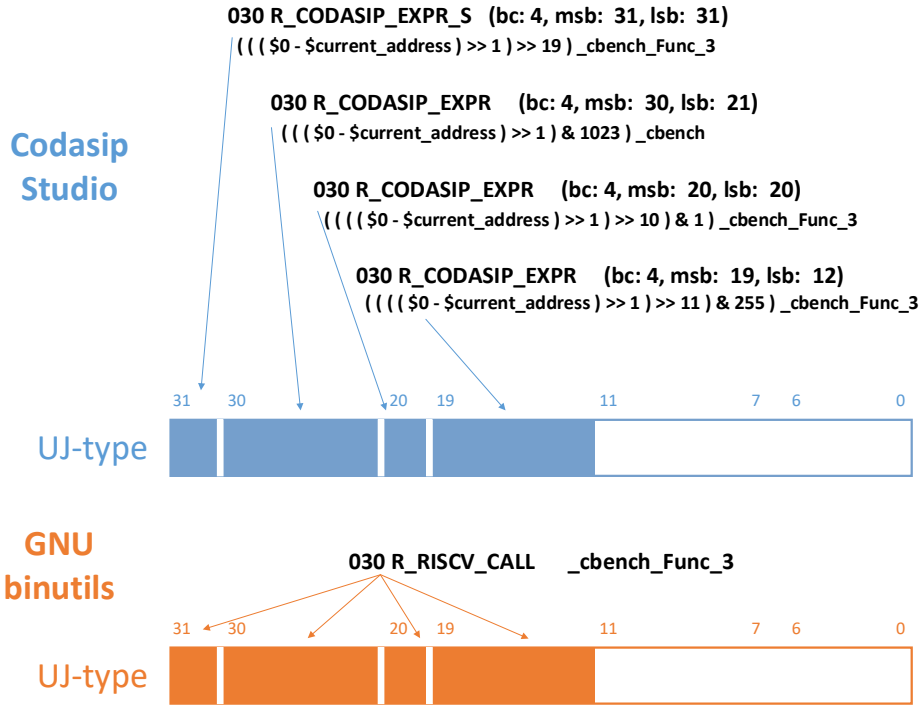
5.3 Kompatibilní relokace

Předpokládejme, že Codasip assembler je již upraven do podoby, která odpovídá změnám popsaným v sekci 5.2. To znamená, že pro každý symbol nacházející se ve vstupním zdrojovém kódu buduje výrazový strom.

Před vytvořením cílového objektového souboru nástroj kontroluje, které z těchto výrazů je možné vyřešit již v tuto chvíli. Výrazy obsahující nedefinované symboly jsou transformovány v relokace, které jsou uloženy do objektového souboru. Zde je tedy v případě generování GNU kompatibilního objektového souboru nutné určit správný typ relokace.

Pro každý typ GNU RISC-V relokace je možné definovat výraz, který popisuje operace, které s relokovaným symbolem daný typ provádí. Pokud se tedy výraz aktuálně ukládaný assemblerem do objektového souboru shoduje s některým z výrazů popisujících typy GNU relokací je relokace do objektového souboru uložena právě v tomto formátu.

Samozřejmě pro možnost linkování takto vzniklého objektového souboru GNU linkerem je nutné, aby se tímto způsobem podařilo přiřadit všechny v souboru nalezené relokace. Pokud zůstanou některé relokace nepřiřazeny GNU typům relokací a nejedná se o chybu assembleru, mělo by být rozhodnutí o výsledku překladu založeno na uživatelově rozhodnutí. Pokud i následné linkování proběhne pomocí Codasip linkeru, není tento stav problém a relokace obsažené v tomto souboru mohou být uloženy v Codasip formátu. Pokud je kompatibilita objektového souboru s cizími nástroji požadována, jedná se o chybu překladu.



Obrázek 5.4: Rozbití relokace na části

Dalším problémem, který v současné situaci způsobuje nekompatibilitu relokací produkovaných Codasip assemblerem s GNU linkerem je samotný počet vzniklých relokací v rámci jednoho vstupního zdrojového souboru. Vlivem formátu instrukce může být v Codasip assembleru vytvořeno několik relokací pro jeden relokovaný symbol (viz sekci 3.1). Naopak

v GNU assembleru vzniká vždy maximálně jedna relokace pro každou instrukci. V případě, že je výstupem assembleru GNU RISC-V kompatibilní objektový soubor je tedy nutné, aby přiřazení typů nalezeným relokacím proběhlo ještě předtím, než jsou relokace rozbity nebo byl assembler upraven tak, aby byl schopen související relokace znovu seskupit.

Obrázek 5.4 ukazuje výstup nástroje objdump pro jednu relokaci vytvořenou při překladu stejného zdrojového souboru oběma assemblery. Ukázka obsahuje relokace týkající se instrukce ve formátu *UJ*, což je možné poznat při porovnání umístění operandu v daném formátu (viz sekci 4.1) s bitovými pozicemi relokace v rámci instrukčního slova určenými položkami *msb* a *lsb*.

Vzhledem k univerzální implementaci relokací v nástrojích Codasip Studio (která je popsána dále v sekci 6.1) došlo ke správnému vytvoření čtyř relokací, kdy každá popisuje jednu souvislou část přímého operandu instrukce. Oproti tomu relokace generovaná GNU assemblerem je pouze jedna a její typ *R_RISCV_CALL* je dostačující k dodání všech dalších vlastností linkeru (viz sekci 4.3).

Kapitola 6

Řešení

Smyslem této kapitoly je popis úprav implementovaných do nástrojů Codasip Studio za účelem docílení kompatibility s nástroji GNU binutils. Návrh těchto úprav byl popsán již kapitolou 5.

Velká část této kapitoly se zbývá detekcí typů relokací v assembleru, což je z teoretického hlediska nejdůležitější bod zadání této práce. Úvodní část kapitoly vysvětluje implementaci univerzálních relokací, která byla naznačena v sekci 5.2, což je naopak implementačně nejnáročnější součást diplomové práce.

Pro lepší pochopení budou jsou ve zbytku práce používány ilustrace útržky kódu v jazyce symbolických instrukcí a konstrukce jazyka C++ přibližující implementaci assembleru. V obou případech se jedná o zjednodušenou podobu, která přesně neodpovídá jak instrukční sadě RISC-V, tak implementaci nástrojů Codasip Studio. Toto zjednodušení jsem použil zcela záměrně, aby bylo pro čtenáře jednodušší tyto části pochopit. V opačném případě by bylo nutné vysvětlit další teoretické základy, které by zbytečně prodlužovaly celý text.

6.1 Univerzální relokace

Prvním velkým krokem byla implementace univerzálních relokací. Jedná se o rozsáhlý zásah do nástrojů Codasip Studio v tom směru, že bylo ovlivněno nástrojů hned několik. Kromě binárních utilit, tedy assembleru a linkeru, na jejichž údržbě se podílím, a proto jsem pracoval i na implementaci tohoto rozšíření, bylo nutné upravit i nástroje starající se o práci s jazykem CodAL a převodu reprezentace procesorové architektury z tohoto jazyka do podoby, kterou využívají jednotlivé nástroje.

6.1.1 Úpravy assembleru

Důsledkem povolení definice složitějších výrazů popisujících kódování operandu, které jsou vysvětleny v sekci 5.1 musel být implementován nový způsob reprezentace relokací v objektovém souboru.

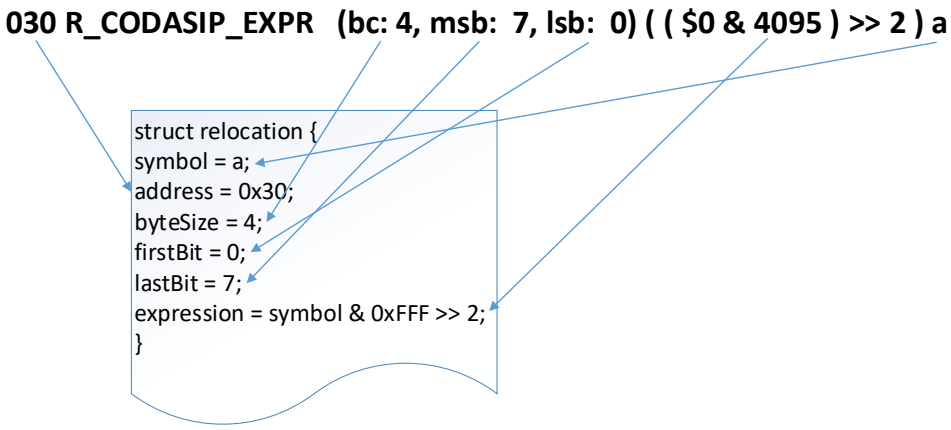
Tento způsob nesměl být omezen jen na určitou malou množinu operací, jako tomu bylo u původním implementace, která je zmíněna v sekci 3.2. Ze záznamu relokace v objektovém souboru byla tedy odstraněna pole specifikující posunutí operandu a konstantu k operandu přičtenou. Místo nich záznam obsahuje odkaz do nově přidané sekce objektového souboru, která obsahuje právě výrazy popisující jednotlivé relokace. Tyto výrazy jsou získány na základě stromu, který buduje assembler pro každou relokaci (viz sekci 5.2). Samozřejmě v případě, že je pro několik relokací použit stejný výraz, je do nové sekce objektového

souboru tento výraz uložen jen jednou a všechny relokační záznamy, které tento výraz využívají se na něj odkazují.

Výrazy popisující jednotlivé relokace jsou v objektovém souboru uloženy v textové podobě. To značně ulehčilo modifikaci ostatních nástrojů, které zobrazují obsah ELF souborů, jako je objdump. Uživatel má možnost získat výpis všech relokací v objektovém souboru společně s výrazem, který se ke každé relokaci pojí. Oproti GNU implementaci relokací tak není odkázán jen na typ relokace, jehož význam si je nucen v případě potřeby dohledat, ale princip dané relokace usoudí z přiloženého výrazu.

Znaménkovost celé relokace je stále dána typem relokačního záznamu, který je buď *R_CODASIP_EXPR* v případě neznaménkové relokace nebo *R_CODASIP_EXPR_S* v případně relokace jejíž nejvyšší bit je znaménkový.

Obrázek 6.1 ukazuje výstup nástroje objdump pro relokaci symbolu *a*, ve které je patrné vypuštění informací oproti původní implementaci a také přidání relokačního výrazu. Dále je v tomto obrázku představena zjednodušená grafická podoba relokace připomínající kód v jazyce C. Tato podoba bude využívána i v dalších částech textu.



Obrázek 6.1: Univerzální relokace ve výpisu object dump

Implementace univerzálních relokací se zatím může jevit jako nedůležitá v kontextu zadání této práce, ale opak je pravdou. Význam této úpravy svou obecností přesahuje zadání této práce. Proto došlo k nahrazení původní implementace relokací v nástrojích Codasip Studio právě implementací popsanou v této sekci. Bez popsaných úprav by navíc nebylo možné uvažovat o generování objektových souborů kompatibilních s nástroji GNU binutils, což bude vysvětleno dále.

6.1.2 Úpravy linkeru

Při načtení vstupního objektového souboru Codasip linkerem, dojde k načtení některých součástí souboru do vnitřních struktur nástroje. Stejným způsobem je nyní zpracována sekce obsahující relokační výrazy.

Sekce je rozdělena na jednotlivé výrazy, které jsou v pozdějších fázích linkování dohledatalné na základě svého offsetu v rámci této sekce a také jména vstupního objektového souboru, ze kterého byl tento výraz získán. Každý výraz je dále ze stromové reprezentace převeden do postfixové notace. Tato notace je velmi dobrá z hlediska rychlosti vyhodnocování. Úvodní časová ztráta, způsobená převodem reprezentací, je tedy obětována za účelem

následného rychlého vyhodnocení výrazu, což je výhodné především při vícenásobném použití jednoho relokačního výrazu.

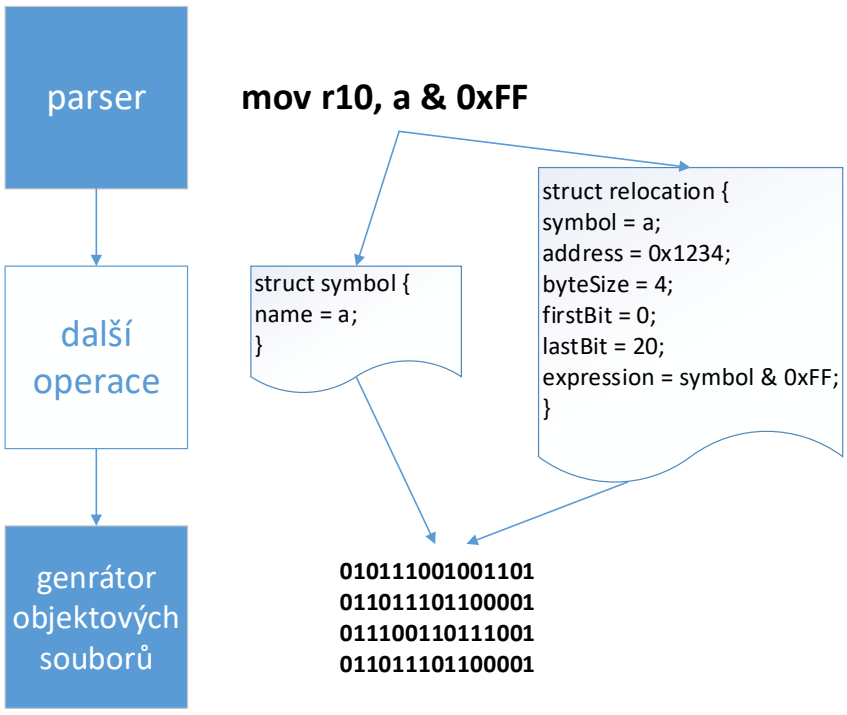
Ve fázi vyhodnocování relokací dochází k postupnému procházení záznamů relokací, ze kterých je získán jednak název symbolu, který je aktuálně relokován, ale také offset výrazu, podle kterého má být výsledek relokace uložen. Tento offset je použit k dohledání příslušného výrazu ve vnitřních strukturách linkeru. Zde je vhodné říci, že ačkoliv některé další nástroje Codasip Studio zatím nedovolují zastoupení dvou symbolů v rámci jedné relokace, na úrovni linkeru toto problém není a implementace univerzálních relokací v tomto nástroji počítá s využitím více symbolů v budoucnu.

Dále samozřejmě linker stále kontroluje, zdali je hodnota získaná jako výsledek vybraného výrazu dostatečně malá pro uložení do instrukčního slova na bitové pozice vyhrazené pro přímý operand. Tato kontrola je provedena s respektem ke znaménkovosti relokace, která je určena na základě typu záznamu relokace v objektovém souboru.

6.2 Části assembleru

Následující sekce rámcově vysvětluje návrh implementace assembleru. Úpravy tohoto nástroje jsou předmětem sekce 6.3 a informace zde uvedené by měly pomoci čtenáři v pochopení implementovaných úprav.

Princip činnosti assembleru můžeme chápat jako sadu transformací a kontrol provedených nad vstupním zdrojovým kódem. Vzhledem k následujícímu textu není nutné se zabývat všemi částmi assembleru. Podstatná je jen úvodní fáze parsování vstupního zdrojového kódu a závěrečná fáze ukládání výsledku překladu do objektového souboru. Ve velmi zjednodušené podobě jsou části assembleru znázorněny obrázkem 6.2.



Obrázek 6.2: Ilustrace důležitých částí assembleru

Transformace v assembleru vedou v prvních fázích překladu k vytvoření řady struktur reprezentujících vstupní program v jazyce symbolických instrukcí uvnitř assembleru. Dále bude tato sada struktur označována jako vnitřní reprezentace.

Na obrázku 6.2 je toto ilustrováno vytvořením struktury *symbol* při nalezení nového symbolu ve vstupním zdrojovém kódu a vytvořením struktury *relocation*, která značí, že hodnota symbolu *a* je assembleru neznámá a bude tedy nutné se s touto situací vypořádat.

Uvedené struktury neodpovídají přesně implementaci assembleru, aby zbytečně nekomplikovaly pochopení tohoto popisu. Jednotlivé položky těchto struktur však samozřejmě nejsou nesmyslné. Ve struktuře *relocation* je uložena informace o symbolu, kterého se daná relokace týká. Dále je zde uvedena adresa ve vstupním programu, kde byla relokace nalezena. Položka *byteSize* značí, že se jedná o relokování oblasti, která je velká 4 byty, což odpovídá velikosti instrukce *mov* a bitové pozice upřesňují polohu relokovaného symbolu v rámci této instrukce. Toto vše odpovídá formátu relokací Codasip assembleru jak byl popsán v sekci 3.2. Poslední položka expression obsahuje výraz, který definuje operace provedené se symbolem a před jeho uložením do binární podoby programu. Tato položka je tedy výsledkem nové rozšířené podoby relokací Codasip assembleru jak bylo detailně popsáno sekcemi 5.2 a 6.1. Takto znázorněná podoba vnitřní reprezentace relokací je dostačující pro následující popis implementace.

Na základě přenesení všech důležitých dat ze vstupního programu do vnitřní reprezentace je možné vstupní program přečíst jen jednou. S vnitřní reprezentací jsou následně prováděny další operace, které nebudou předmětem této práce. Jakmile jsou všechny potřebné fáze překladu dokončeny obsahuje vnitřní reprezentace data, která je potřeba uložit do výstupního objektového souboru, přičemž je nutné respektovat specifikaci jeho formátu.

Následuje popis důležitých částí assembleru, a to především ze hlediska rozdílů v implementaci dostupných nástrojů.

6.2.1 Parser

Jedná se o úvodní část assembleru, kontrolující syntaktickou správnost vstupního kódu v jazyce symbolických instrukcí a tvořící vnitřní reprezentaci vstupního kódu, se kterou následně pracují všechny zbývající části assembleru.

Open source implementace assembleru pro RISC-V ze sady GNU binutils implementuje toto parsování pomocí konečného automatu. Ve chvíli, kdy je zpracována celá instrukce, je znám jak její operační kód a operandy, což jsou informace potřebné, pro určení typu relokace. Dává tedy smysl, aby byl typ relokace v tuto přiřazen k vnitřní representaci instrukce v assembleru.

Parser v assembleru z SDK generovaného pomocí Codasip Studio je implementován pomocí nástroje BISON [6], který umožňuje jeho automatické generování na základě dodané gramatiky vstupního jazyka. Toto řešení je velmi vhodné, protože gramatiku je možné získat na základě modelu procesoru a zbytek je vyřešen automaticky díky BISON.

Pokud by nebylo využito open source řešení, jako je právě BISON, bylo by nutné naimplementovat vlastní generátor parseru. Návrháři dané procesorové architektury totiž nic nebrání v přidávaní a odebírání jednotlivých instrukcí, což značně ovlivňuje právě parser assembleru a jeho ruční úpravy jsou z principu filozofie nástrojů Codasip Studio nepřijatelné.

Vzhledem k tomu, že součástí parseru generovaného pomocí BISON je kontrola syntaxe jazyka, přestává být nutné udržovat pro další části assembleru všechny informace uvedené ve vstupním zdrojovém kódu. Mezi informacemi, které již dále nejsou známy jsou i typy

instrukcí, které se ve vstupu vyskytovaly. Pro následující odstavce je především nutné zmínit, že v případě, kdy parser narazí na přímý operand v podobě symbolu a zhodnotí, že ze syntaktického hlediska je toto správně, uloží informaci o pozici relokace pro další zpracování. Ovšem k takto vytvořené relokaci nepřiřadí informaci o typu instrukce, ve které se vyskytuje, a to jednoduše proto, že v současné implementaci assembleru to není nadále potřeba (toto je naznačeno i na obrázku 6.2).

Jak již bylo zmíněno v sekci 3.2, implementace relokací v nástrojích Codasip Studio žádným způsobem nereflektuje to, jaké instrukce se daná relokace týká. Stejně tak nedošlo k přidání této informace do nové implementace relokací (viz sekci 6.1.1). Je důležité zmínit, že tato skutečnost nijak nebrání v tom, aby Codasip Studio generovalo SDK použitelné s RISC-V architekturou. Jen značně komplikuje snahu o kompatibilitu nástrojů s nástroji GNU binutils.

6.2.2 Generátor objektových souborů

Tato část se zabývá transformací výsledku překladu provedeného assemblerem z vnitřní reprezentace do objektového souboru. Vzhledem k tomu, že se jedná o část, která je velmi závislá na implementaci předešlých částí nástroje, je velmi nepravděpodobné, že by se pro ni tvořily obecné generátory, jako tomu bylo u parseru popsaného v předešlé sekci. V rámci návrhu assembleru je většinou počítáno s tím, že tato část může být vybrána z několika implementací, které se liší tím, jaký formát objektového souboru generují.

6.3 Kompatibilní relokace

Tato sekce popisuje implementované změny nutné pro umožnění generování objektových souborů kompatibilních s GNU binutils nástroji. Oproti sekci 6.1, kde byly popsány změny jak v assembleru tak v linkeru, zde došlo k modifikaci pouze prvního jmenovaného nástroje. Celá tato úprava je vykonána ve snaze umožnit použití GNU linkeru pro linkování získaných objektových souborů, proto úpravy Codasip linkeru nejsou potřebné.

6.3.1 Umístění úprav

Jádrem celé implementace je návrh algoritmu, který přiřazuje jednotlivým instrukcím, respektive jejím přímým operandům, správný typ relokace ze sady podporované nástroji GNU binutils. Neméně důležitou součástí řešení je však také určení správného místa v současné implementaci Codasip assembleru, kam je možné tuto klasifikaci zakomponovat.

- Parser - Na základě vysvětlení v sekci 6.2.1 se tato alternativa jeví jako vhodná hned ze dvou důvodů. Prvním je, že je to jediná část assembleru, která má informaci o typu instrukcí. Mapování jednotlivých relokací tedy může probíhat přímo na základě operačního kódu aktuálně parsované instrukce. Druhým důvodem je fakt, že ve stejné části assembleru je toto přiřazování implementováno i v GNU bintutils assembleru.

  Ovšem faktory působící proti zvolení této alternativy jsou také významné. Úpravy týkající se částí pracujících s BISON nejsou triviální a vyžadují znalosti tohoto generátoru. V případě, že se v budoucnu rozšíří specifikace RISC-V způsobem, který zavede nové typy relokací, může být pro případného dalšího vývojáře problematické je reflektovat.

Co je však podstatnější, je fakt, že tento způsob implementace by vyžadoval do vnitřní reprezentace relokací, která byla zmíněna v sekci 6.2, umístit určitá metadata specifikující typ GNU binutils kompatibilní relokace. Pouze takto by bylo možné přiřazené typy relokací přenést až do závěrečné fáze, kdy budou uloženy do výstupního objektového souboru. Tyto změny by však zůstaly součástí assembleru i v případě, že by byl assembler generován pro jinou architekturu, než je právě RISC-V. Aby tento vedlejší efekt nenastal, bylo by nutné generovat prakticky všechny zdrojové kódy assembleru. To není žádoucí, ať už z hlediska rychlosti generování nebo nutnosti takové soubory uvolnit uživateli Codasip Studia.

- Generátor objektového souboru - Celá tato část Codasip assembleru byla od začátku jeho vývoje zamýšlena jako zaměnitelná. V současnosti se používá implementace generující objektový soubor ve formátu ELF (viz sekci 2.3). Formátů objektových souborů je však více a je proto možné vytvořit implementaci novou, která generuje jiný formát. Proto není v zásadě problém nahradit jen část, která se stará o generování záznamu relokací na základě jejich reprezentace v assembleru. Umístění kódu zajišťujícího kompatibilitu s GNU binutils do této části assembleru jednoduše ovlivní nejmenší část z jeho implementace, jelikož se jedná o část poslední.

  Jasnou nevýhodou je, že vnitřní reprezentace relokací v této části assembleru již neuvádí instrukci, ve které se vyskytovala ve vstupním kódu v jazyce symbolických instrukcí. Toto mohla být nepřekonatelná překážka, která by zabránila přijetí této alternativy. Díky specifickému formátu instrukcí architektury RISC-V ji však je možné překonat, jakým způsobem bude vysvětleno později.

- CodAL - Existuje i třetí možnost a tou je definice operandu v modelu procesoru, která je vysvětlena v sekci 3.1. Součástí této konstrukce v jazyce CodAL by mohla být položka označující typ relokace, která se váže s tímto druhem operandu. Za předpokladu, že typ relokace vyplývá z typu operandu by se jednalo o velmi přímočaré řešení. Pokud by byl typ relokace svázán jak s typem operandu, tak s typem instrukce, ve které je daný operand použit, vyžadovalo by toto řešení duplikaci některých definic v modelu procesoru, ale i tak by umožňovalo jednoznačné určení typu relokace, což je podstatné.

  Zásadní nevýhodou je však skutečnost, že návrhář procesorového jádra by musel znát principy relokování kódu a musel by být schopen určit jakým způsobem se mají jednotlivé vytvořené operandy relokovat. Navíc stanovení potřebného počtu různých typů relokací a jejich významu je spíše implementační záležitost nástrojů SDK a dává tedy větší smysl, aby se právě ty tvořily na základě popisu procesorové architektury a ne naopak.

Především z důvodu provedení co nejméně rozsáhlých úprav byla pro implementaci zvolena alternativa umístění algoritmu přiřazení typů relokací do generátoru objektového souboru.

6.3.2 Přiřazení typu relokace

Implementované rozšíření assembleru ukládá do objektového souboru správné relokace na základě několika kroků. Všechny tyto kroky probíhají již ve fázi generování objektového souboru nad vnitřní reprezentací relokací.

- Nejprve je nutné seskupit relokace do skupin tak, že jedna skupina obsahuje relokace týkající se jednoho operandu v rámci stejné instrukce. Za jednu takovou skupinu je možné považovat relokace uvedené na obrázku 5.4. Na základě vysvětlení uvedeného v sekci 3.2 není toto zaručeno automaticky. Právě naopak, vnitřní reprezentace relokací je generována v parseru a postup procházení vstupního programu je dán implementací BISONu.

  Nelze tedy předpokládat žádné konkrétní pořadí nalezených relokací. Všechny relokace je potřeba projít, rozdělit do menších skupin relokujících stejný operand. Relokace v těchto menších skupinách jsou seřazeny tak, že relokace týkající se úseku přímého operandu, který se vyskytuje na nižších bitech instrukčního slova bude předcházet ty, které popisují relokaci úseku na bitech vyšších.

- Zkontrolovat, že každá skupina obsahuje právě tolik relokací, aby popisovala přesně jednu validní instrukci. To znamená, že instrukce s operandem rozděleným do tří úseků musí obsahovat ve vnitřní reprezentaci přesně tři relokace. To se jeví jako jasný požadavek, ale jeho opomenutí by mohlo znamenat značné problémy, a to především v budoucnosti, kdy bude model procesoru rozšiřován o nové instrukce vyžadující nové typy relokací.

  Pokud by v takovém případě nedošlo ke správnému reflektování těchto změn v nástrojích, jmenovitě tedy v assembleru, měl by skončit překlad programů pro tuto novou architekturu chybou. Bez dostatečných kontrol by však mohlo dojít na mapování nových instrukcí a jejich operandů na již implementované typy relokací. Ladění takovýchto chyb je poté často velmi náročné a nepříjemné.

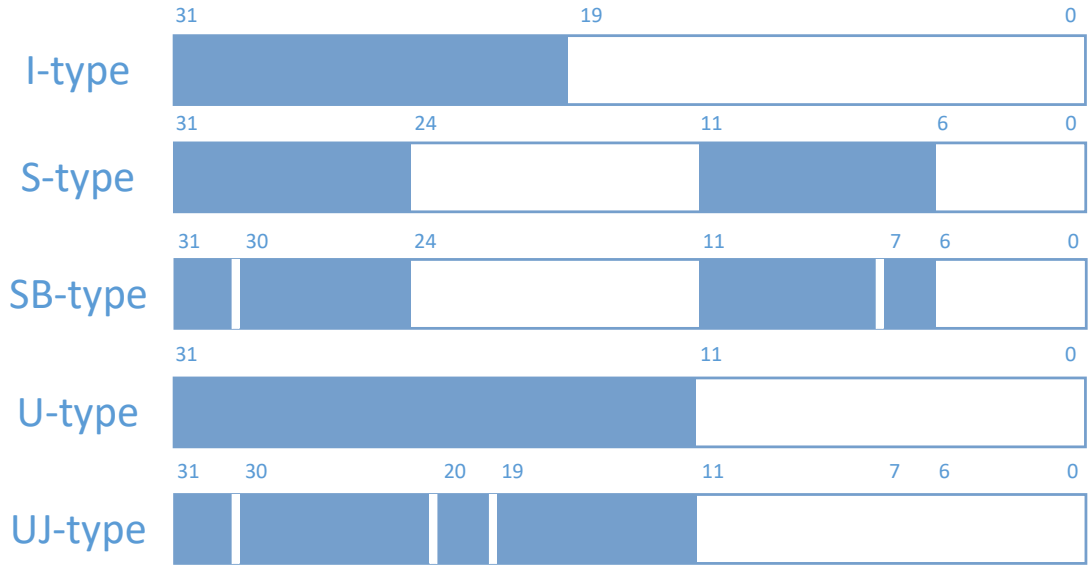
- Určení typu relokace, která popisuje skupinu relokací vnitřní reprezentace assembleru. Pokud není nalezen žádný typ, který by aktuálně zpracovávanou skupinu relokací popisoval, musí assembler uživateli ohlásit srozumitelnou chybu. Tato situace může nastat na základě chyby v implementaci modelu procesoru, případně na základě implementace rozšíření modelu, které nebylo reflektováno v implementaci assembleru.

Předchozí odstavce již jasně popsaly důvody vedoucí k tomu, že vnitřní reprezentace relokací neobsahuje informaci o typu instrukce ani typu operandu, ke kterému se váže. Může se zdát, že by to bylo výhodou při implementaci tohoto rozšíření assembleru.

Ovšem pokud by implementace byla založena na prostém mapování názvu instrukce, potažmo typu jejího operandu na typ relokace, znamenalo by to, že s každou přidanou instrukcí bude nutné upravit assembler. To by v zásadě porušovalo filozofii Codasip Studio a zatěžovalo jeho vývojáře prací kdykoliv se někdo z jeho uživatelů pro takovou změnu rozhodne. Úpravy assembleru budou v každém případě nevyhnutelné, pokud dojde k rozšíření instrukční sady architektury RISC-V o instrukce vyžadující nové typy relokací. Vzhledem k principům, na kterých je tato architektura založena (viz kapitola 4), toto není v blízké době očekáváno.

Přiřazování typů relokací se tedy řídí jinou vlastností instrukce a tou jsou bitové pozice jejího přímého operandu. Tato informace musí zůstat zachována i ve vnitřní reprezentaci relokací po naparsování vstupního kódu, a to především proto, že je nutné, ji nakonec zapsat do objektového souboru. Každý záznam relokace obsahuje index nejvyššího a nejnižšího bitu instrukčního slova, které budou v linkeru relokovány, tak jak bylo ilustrováno obrázkem 6.2 a v příslušné sekci i vysvětleno.

Obrázek 6.3 obsahuje pozice přímých operandů všech důležitých instrukčních formátů základní instrukční sady RISC-V v podobě, která byla zavedena v sekci 2.1. Kompletní popis významu jednotlivých bitů instrukčních formátů je uveden na obrázku 4.1.



Obrázek 6.3: Pozice přímého operandu v instrukčních formátech RISC-V

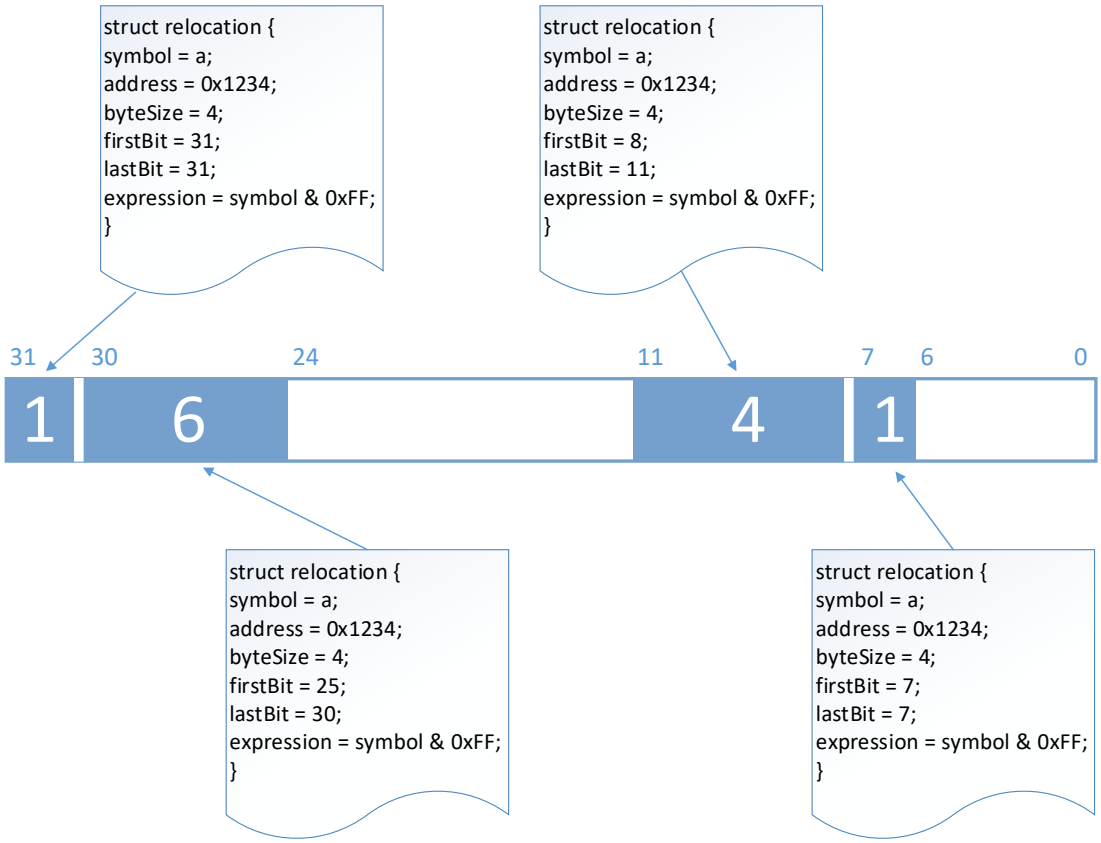
Pokud tedy dojde k seskupení a seřazení relokací do skupin, jak to bylo popsáno výše, je možné určit, které bity daného instrukčního slova jsou obsazeny přímým operandem. Toho jsem využil k získání formátu dané instrukce, což je první vlastnost instrukce definující typ na ní aplikované relokace.

Obrázek 6.4 slouží jako příklad určení formátu instrukce. Pokud si představíme, že instrukce *mov* z obrázku 6.2 má formát právě *SB*, nebude vnitřní reprezentace obsahovat jen jeden záznam relokace jejího operandu, ale právě záznamy čtyři. Čísla uvnitř bloků označujících pozici přímého operandu v instrukčním slově značí bitovou šířku daného bloku. Jestliže tedy popíšeme každý z formátů pomocí posloupnosti těchto bitových šířek, zjistíme, že je tento popis jednoznačný. Navíc je možné tuto informaci získat i z vnitřní reprezentace relokací, jak je také patrné z obrázku.

Podíváme-li se na všechny formáty instrukcí architektury RISC-V (viz obrázek 6.3) je možné vypozorovat, že žádné dva formáty se neshodují v posloupnosti bitových šířek bloků obsahujících přímý operand. Situace je pro účely této práce ještě lepší v tom, že neexistuje ani případ, kdy by tato posloupnost bitových šířek byla v případě jednoho z formátů podmnožinou formátu jiného.

Přiřazení typů relokací je implementováno jako stavový automat. Pokud je celá skupina relokací zpracována, mohou nastat dvě možnosti. Automat může být ve stavu popisujícím některý z platných typů relokací nebo je automat ve stavu indikujícím chybový stav.

Přechody v rámci výpočtu tohoto automatu jsou řízeny primárně bitovými šířkami relokací ve zpracovávané skupině. Podívejme se však třeba na typ relokací *R_RISCV_LO12_I* a *R_RISCV_PCREL_LO12_I*, které byly zmíněny již v sekci 4.3. Obě tyto relokace jsou svázány se stejným instrukčním formátem. Liší se však tím, zdali je hodnota operandu relativní vůči programovému čítači nebo není. Tuto vlastnost nelze samozřejmě určit na základě bitových šířek operandu.



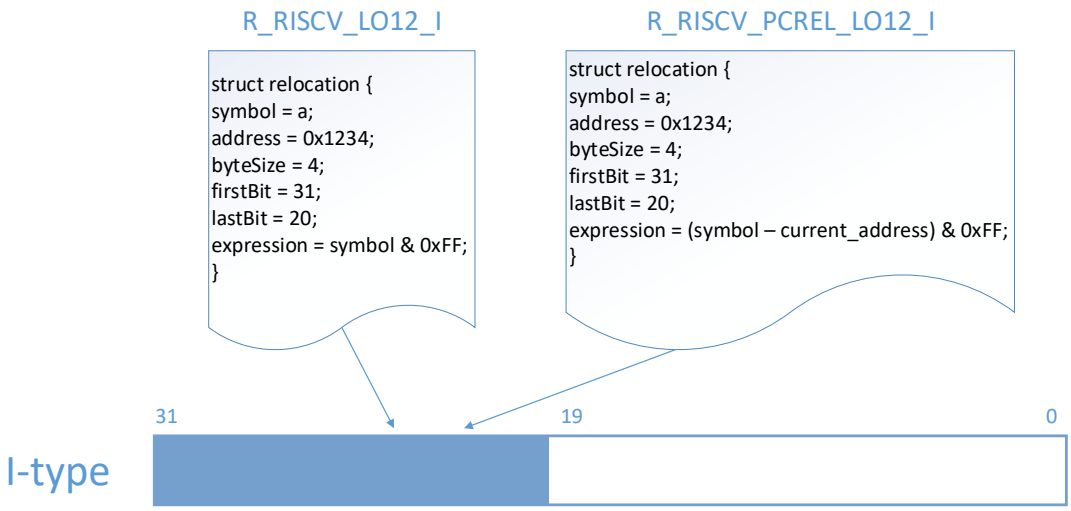
Obrázek 6.4: Pozice přímého operandu v instrukčním formátu SB a související relokace

Nyní tedy přichází vhod upravená implementace relokací v nástrojích Codasip Studio, která byla popsána v sekci 5.2. Pokud se jedná o relokaci relativní k programovému čítači, musí být tato informace zanesena do popisu operandu (viz sekce 3.1) což se následně projeví i ve výrazu popisujícím relokaci ve vnitřní reprezentaci v assembleru. Tuto informaci je následně možné z výrazu vyčíst a rozhodnout tak mezi těmito dvěma typy relokací.

Pro ilustraci slouží obrázek 6.5. Pro jeho pochopení si představme, že instrukce *mov* z obrázku 6.2 je ve formátu *I*. Pokud by její operand *sym* měl být uložen do binární podoby programu v absolutní podobě, byl by popsán relokací *R_RISCV_LO12_I*. Pokud by však měl být uložen relativně k programovému čítači, byla by využita relokace *R_RISCV_PCREL_LO12_I*.

Zde je vhodné zmínit, že informaci o relativitě operandu vzhledem k programovému čítači umožňovala i stará implementace relokací Codasip Studio. Ovšem RISC-V relokace umožňují uvažovat operand relativní i k jiným registrům této architektury, a proto byla tato změna nástrojů nezbytná i pro následující úpravy vedoucí ke kompatibilitě s nástroji GNU binutils.

Na základě určení formátu instrukce a vyčtení případných potřebných informací z výrazu popisujícího kódování operandu je možné určit typ relokace všech instrukcí instrukční sady architektury RISC-V. Při implementaci zde popsaného rozšíření assembleru jsem však narazil na menší komplikace, které budou popsány dále.



Obrázek 6.5: Rozdíl relativní a absolutní relokace stejného instrukčního formátu

6.3.3 Komprimovaná instrukční sada

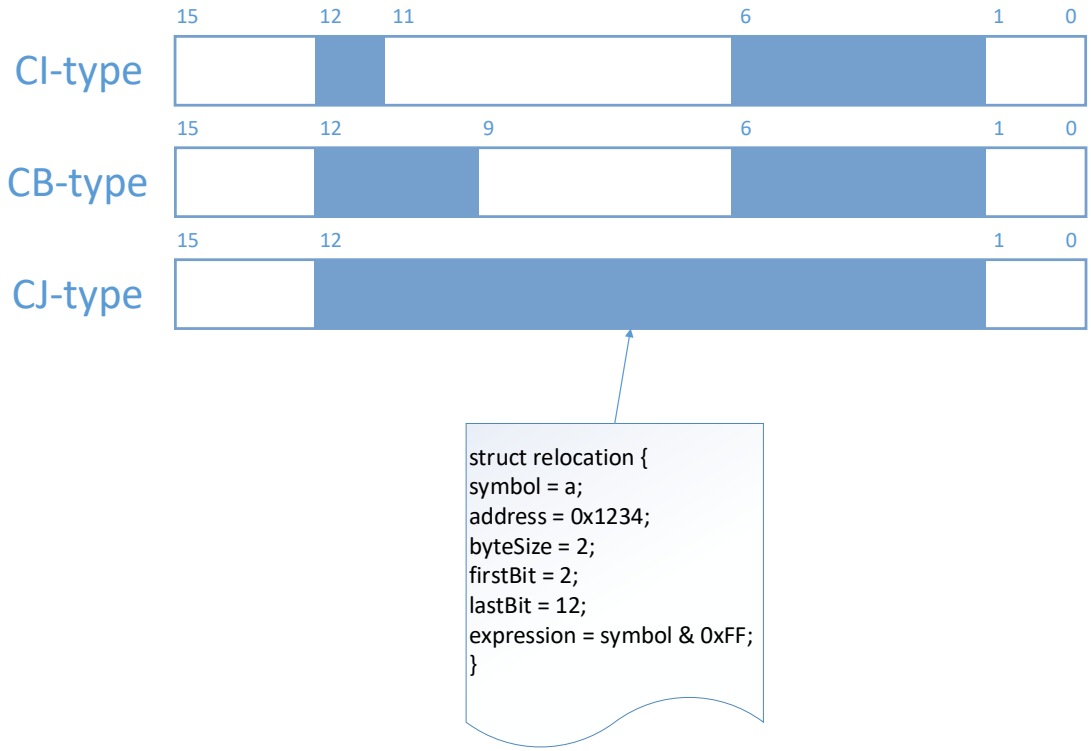
Při určení formátu instrukce v assembleru je nutné kromě instrukcí základní instrukční sady architektury RISC-V brát v úvahu také instrukce z komprimované instrukční sady, jelikož právě ty obsahují rozdílné instrukční formáty oproti základní instrukční sadě. Komprimované instrukce tvoří standardní instrukční rozšíření architektury RISC-V, a proto je nutné počítat s tím, že překladač může tyto instrukce využívat.

V podstatně se nejedná o žádný zásadní problém, kvůli kterého by bylo nutné změnit přístup k přiřazování typu relokací. Jen je nutné vzít v úvahu nové formáty instrukcí, které komprimované sada přináší. Navíc každý relokační záznam ve vnitřní reprezentaci obsahuje informaci o počtu bytů, kterých se tato relokace týká. Tento počet bytů je možné uvažovat jako šířku samotného instrukčního slova. Díky tomu je poměrně jednoduché určit, zdali aktuálně zpracovávaná relokace souvisí s instrukcí základní instrukční sady nebo s instrukcí komprimovanou. Více o komprimované instrukční sadě je uvedeno v kapitole 4.2.

Obrázek 6.6 ukazuje pozice přímých operandů instrukčních formátů komprimované sady, které mohou pracovat s relokovatelným operandem. Kupříkladu formát *CL* může obsahovat také přímý operand, ale vždy se musí jednat o číselnou konstantu, a tedy nikdy nedojde k jeho relokaci. V obrázku je také naznačena vnitřní reprezentace relokace formátu *CJ*. Důležitá je především položka *byteSize* informující o délce instrukce, která ulehčuje určení správného typu relokace, jak bylo uvedeno o několik řádků výše.

6.3.4 Assembler alias

Assembler alias je prostředek pro zjednodušení tvorby programů v jazyce symbolických instrukcí. Některé standardní operace, pracující s velkým přímým operandem, jako je například přesunutí konstanty o velikosti celého slova do registru, nelze provést pomocí jedné instrukce. Proto se používá sekvence dvou po sobě jdoucích instrukcí, které jsou však pro danou operaci vždy stejné. Z tohoto důvodu syntaxe některých jazyků umožňuje použití instrukcí vykonávajících tyto operace, které ve skutečnosti daná architektura nepodporuje,



Obrázek 6.6: Pozice přímého operandu v komprimovaných instrukčních formátech RISC-V

ale během zpracování programu assemblerem jsou nahrazeny za posloupnost instrukcí, které již daná architektura dokáže vykonat.

Obrázek 6.7 naznačuje expanzi assembler aliasu *la*. Tento alias souží právě k načtení velké adresy[1] do registru. Ve skutečnosti je tato operace vykonána pomocí instrukce *auipc*, která do registru umístí horní část adresy, a následně je využita instrukce *addi*, která naplní spodní část registru.

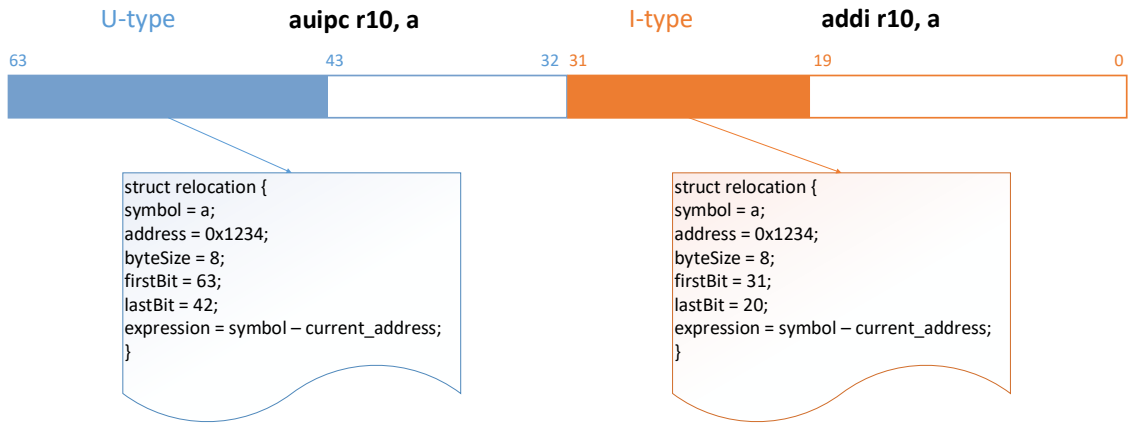
Obrázek 6.7: Expanze assembler aliasu

V případě nástrojů Codasip Studio jsou aliasy uvedeny v modelu procesoru, jakou součást instrukční sady. Překladač sám tyto instrukce nikdy nevyužívá a místo nich volí přímo posloupnost instrukcí vedoucí k ekvivalentnímu výsledku. Ovšem možnost využití assembler aliasu zjednodušuje tvorbu programů napsaných ručně v jazyce symbolických instrukcí. Aliasy jsou většinou tvořeny pro často vykonávané operace, a tudíž umožní tvorbu kratšího

[1] *la* je zkratka anglického *load address*

zdrojového kódu a navíc nezatěžují programátora nutností znát posloupnosti operací nutné pro provedení stejného efektu, jaký vykoná právě daný assembler alias.

V kontextu Codasip assembleru a jeho rozšíření popsaného v této práci způsobují assembler aliasy jistý problém, který je naznačen obrázkem 6.8. Vnitřní reprezentace relokací, které mohou v rámci takového aliasu vzniknout, nereflektují fakt, že se jedná o dvě po sobě jdoucí instrukce. Znamená to tedy, že záznam se tváří jako kdyby se jednalo o instrukci délky odpovídající součtu délek instrukcí assembler aliasu a jednotlivé bitové pozice operandu jsou tedy vypočítány adekvátně k této délce. Na obrázku jsou znázorněny pozice přímého operandu v assembler aliasu *la*. Z vyobrazené vnitřní reprezentace je viditelné, že relokace pokrývá 8 bytů což odpovídá délce obou instrukcí dohromady. Navíc toto reflektují i bitové pozice přímého operandu instrukce *auipc*.



Obrázek 6.8: Relokování assembler aliasu

Může se zdát rozumné, aby byl assembler upraven tak, že bude společně s nahrazením aliasu za skutečné instrukce upravovat i odpovídající relokace, tedy rozbíjet je na více relokací, připadajících vždy jen k jedné z instrukcí assembler aliasu. Pokud by navíc toto rozbití proběhlo dříve, než dojde na provedení rozšíření popsaného v této kapitole, mohly by být následně správně přiděleny typy GNU relokací bez dalších úprav této nové části assembleru. Bohužel toto není jednoduchý úkol, jelikož má tento způsob implementace v Codasip assembleru své opodstatnění v případě použití Codasip nástrojů s jinými architekturami, než je RISC-V.

Dalším důvodem proti této úpravě je fakt, že RISC-V nevyužívá assembler aliasů jen jako prostředku ke zjednodušení programů. Konkrétně zde zmiňovaný alias *la* je toho vhodným příkladem.

Operandy instrukcí *auipc* i *addi*, které tvoří tento alias, jsou uloženy v podobě relativní k programovému čítači, což bylo naznačeno i na obrázku 6.8. Aby byla výsledná adresa načtená do cílového registru správná, je nutné, aby k výpočtu relativní hodnoty operandů byla použita v obou případech stejná hodnota programového čítače.

Jelikož se však jedná o dvě instrukce, bude mít při běhu programu čítač určitý obsah během vykonání instrukce *auipc* a jiný obsah během vykonání instrukce *addi*. Rozdíl bude odpovídat nejméně velikosti instrukce *auipc* a to v případě, že budou obě instrukce následovat bezprostředně za sebou, ale může být dokonce větší, a to pokud budou mezi tyto dvě instrukce vlivem přeskládání kódu vloženy další.

GNU assembler proto pro tyto dvě instrukce nevytvoří dvě nezávislé relokace pracující se stejným symbolem. Relokovaný symbol je uveden pouze v relokaci instrukce *auipc*. Relokace týkající se instrukce *addi* obsahuje v položce symbol hodnotu *.L0*, což je v programu neexistující lokální návěští. V tabulce symbolů vzniklého objektového souboru je jako hodnota symbolu *.L0* uvedena adresa příslušné instrukce *auipc*. Díky tomu může GNU linker obě relokace znovu spárovat, což je nutné ke správnému relokování objektového souboru.

6.3.5 Výsledný algoritmus

Na základě popisu detekce typu relokace pro jednoduchou instrukci a komplikacím, které vznikají zavedením komprimované instrukční sady a assembler aliasů dochází k určení typu relokace v několika krocích. Jakmile je v některém z těchto kroků možné typ relokace bezpečně určit, je tento typ následně uložen do objektového souboru a další kroky jsou vynechány. Pokud naopak není možné dané instrukci přiřadit validní typ v ani jednom z dále uvedených kroků, je toto považováno za chybu a činnost assembleru je adekvátně ukončena.

Celý proces přiřazení správného typu relokace je znázorněn obrázkem 6.9. Ilustrace začíná roztříděním vnitřní reprezentace relokací do skupin na základě shody symbolu a adresy, tak jak bylo popsáno v sekci 6.3.2. Následně je pro každou skupinu určen typ relokace ve třech krocích. Popis těchto kroků následuje.

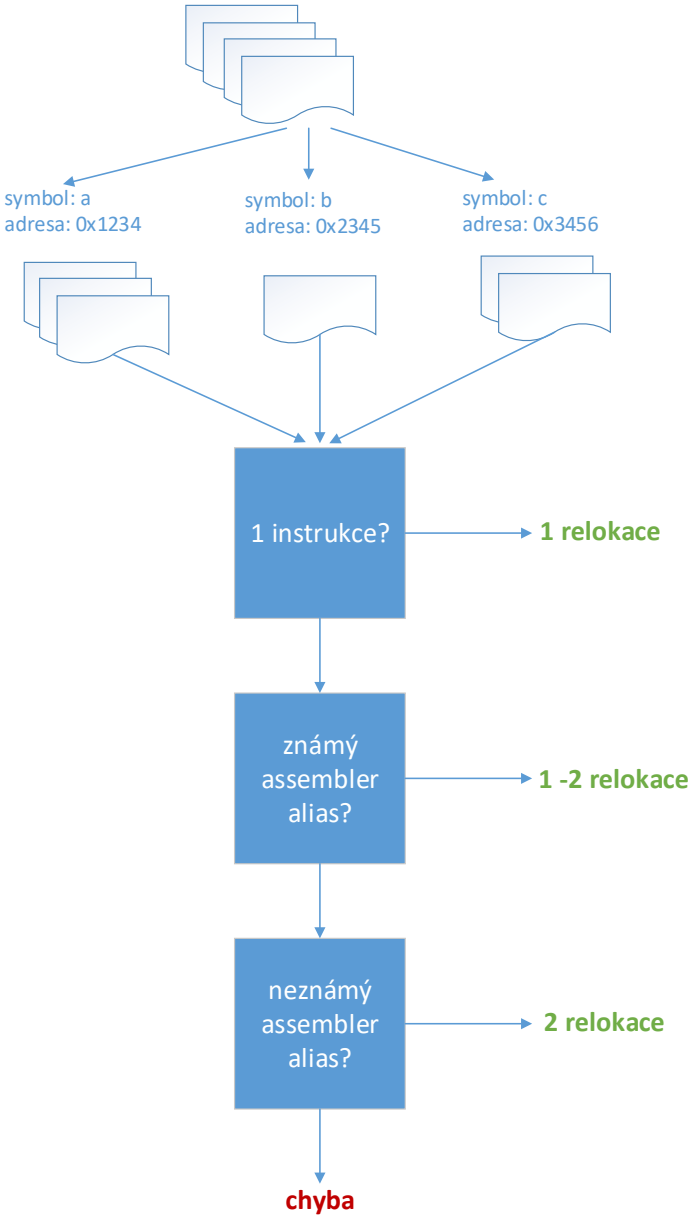
- Celá skupina relokací je považována za relokaci jedné instrukce. Na základě formátu instrukce, který by odpovídal této skupině a dalších informací získaných z výrazu popisujícího kódování operandu je určeno, zdali je možné pro takovou instrukci určit validní typ relokace. Pokud ano, je pro tuto skupinu vygenerována do objektového souboru právě jedna relokace. Tato část byla popsána sekcí 6.3.2.

- Celá skupina relokací je považována za relokaci některého assembler aliasu. Pro každý známý assembler alias musí být v assembleru implementována metoda, která jej správně zpracuje a tím je myšleno jak samotné určení, zdali aktuální skupina relokací aliasu odpovídá tak další operace jako jsou ty týkající se aliasu *la*, které jsou popsány v sekci 6.3.4.

  Pokud dojde k úspěšné detekci některého ze známých aliasů, jsou do objektového souboru vygenerovány nejčastěji dvě relokace. V jednom speciálním případě je celý alias tvořený dvěma instrukcemi popsán pouze jednou relokací.

- Skupina relokací neodpovídá žádné známé instrukci ani assembler aliasu. Je možné, že se jedná o alias, který byl do modelu přidán a nebyla pro něj implementována příslušná metoda v assembleru. Skupina je tedy rozdělena napůl tak, aby nové skupiny obsahovaly relokace pro části přímého operandu odpovídající spodní, respektive horní polovině aliasu. Každá z těchto skupin je následně samostatně zpracována.

  Tímto je umožněno přidávat do modelu procesoru nové aliasy, nevyžadující žádné specifické zacházení, bez nutnosti úpravy assembleru. Pokud se každé z nově vzniklých skupin podaří přiřadit typ relokace budou následně do objektového souboru uloženy dvě relokace. Pokud se to alespoň v jednom případu nepodaří, je assembler ukončen a uživateli je předložena chybová hláška popisující tuto situaci.



Obrázek 6.9: Proces přiřazení typů relokací

Čtenáři může připadat, že to, zdali daná skupina relokací relokuje jednu instrukci či více instrukcí ve formě aliasu je možné určit již na základě informace o velikosti relokované paměti, která je součástí vnitřní reprezentace relokací. Na základě sekce 6.3.4 je však nutné si uvědomit, že instrukce základní sady mají délku 32 bitů, což odpovídá případnému aliasu složenému ze dvou komprimovaných 16bitových instrukcí. Proto je opravdu nutné vyzkoušet všechny tři možnosti uvedené výše.

Kapitola 7

Význam implementovaného rozšíření a výsledky

Předcházející kapitola obsahuje popis implementace rozšíření nástrojů Codasip Studio. Do této části práce bylo toto rozšíření opodstatněno zajištěním kompatibility s nástroji GNU bintutils. Výhody plynoucí z této kompatibility budou vysvětleny nyní, kdy, již byly v textu zmíněny všechny náležitosti potřebné k pochopení těchto výhod.

7.1 Kompatibilita objektových souborů

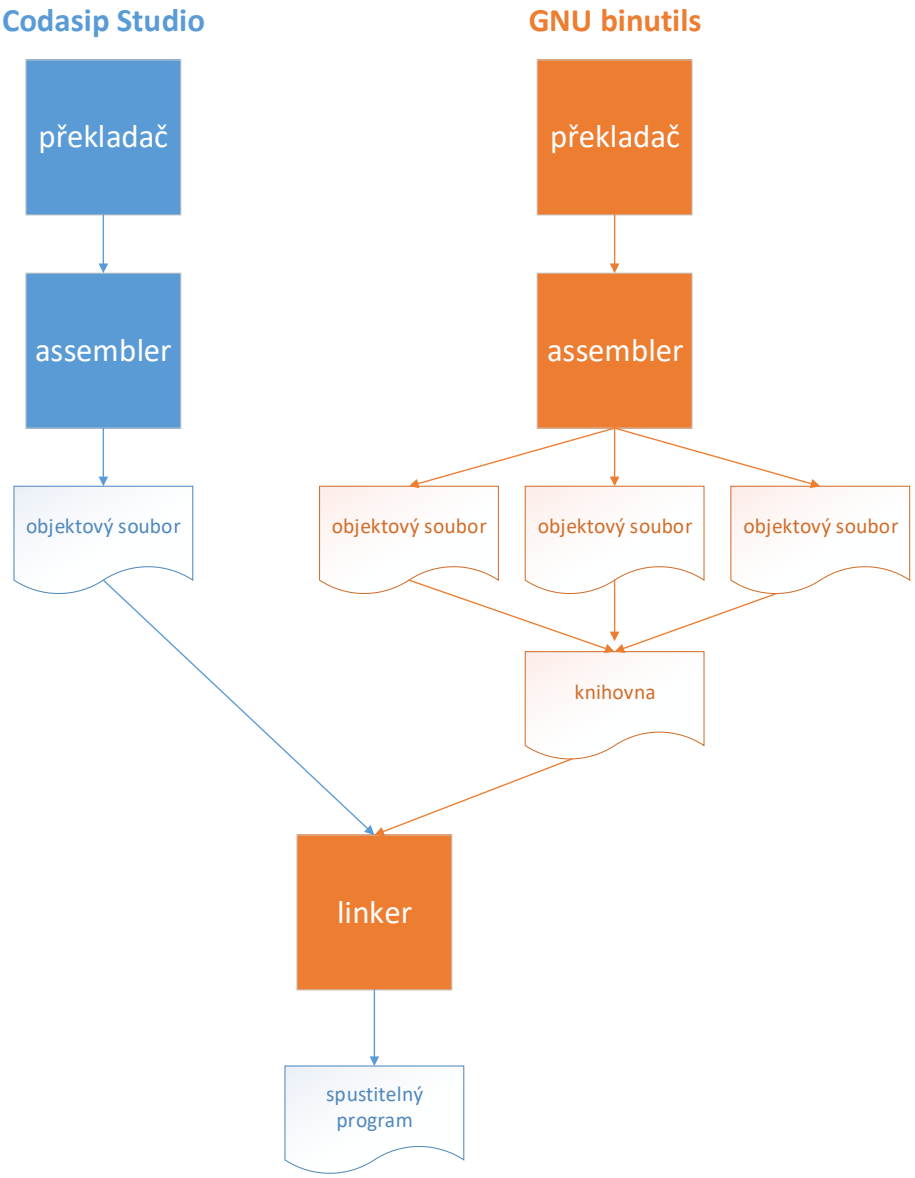
Objektový soubor byl v kapitole 2 pro účely této práce zaveden jako výstup assembleru. Vzhledem k tomu, že většina uživatelů k překladu programů používá nástroje předinstalované v jejich systému, které samy spustí jednotlivé fáze překladu, je možné, že se spousta programátorů nedostane do styku s objektovými soubory v takovéto podobě.

Běžně však programátoři při tvorbě svých programů využívají již hotové funkce a jiné části kódu obsažené v knihovnách. Pro ilustraci je možné uvést standardní knihovnu jazyka C, kterou během své práce využijí snad všichni programátoři vyvíjející v tomto jazyku. Knihovnu je možné chápat jako množinu objektových souborů, které byly pro zjednodušení manipulace s nimi sbaleny do jednoho objektu, kterému se říká archiv. Důvodů pro vytváření archivů je více, ale nejsou nijak relevantní v rámci této práce. Důležité je, že archivy (nebo knihovny, tato označení je možné v rámci tohoto textu zaměnit) mohou figurovat společně s objektovými soubory jako vstupy linkeru.

Tvůrce knihovny danou knihovnu často vytváří proto, aby ji následně poskytl jiným programátorům, tedy uživatelům dané knihovny. Ti budou chtít získanou knihovnu využít tak, že ji slinkují se svým programem. Tvůrce knihovny může disponovat komerční verzí překladových nástrojů, které však nesmí dále distribuovat. Musí tedy spoléhat na to, že objektové soubory a následně tedy i vytvořené knihovny je možné linkovat pomocí linkeru, ke kterému má přístup i uživatel knihovny.

Stejný předpoklad platí i pro uživatele knihovny. Ten může také využívat jakékoliv překladové nástroje. Ovšem pokud se chystá využít pří linkování knihovnu, ke které nevlastní zdrojové kódy a nemůže si ji tedy přeložit svými nástroji, musí spoléhat na to, že objektové soubory v knihovně a ty, které vznikly překladem jeho vlastního programu, je možné slinkovat nějakým dostupným linkerem. Jako takový linker může teoreticky posloužit ten, který je součástí komerční verze nástrojů, které uživatel vlastní. Ovšem to by neřešilo situaci z pohledu tvůrce knihovny, který nemá žádné ponětí o tom, jaké nástroje vlastní uživatelé

jeho knihoven. Proto je vhodné, aby se za tento referenční linker považovala open source implementace, ke které mají přístup všichni a je zdarma. Tou je v této práci GNU binutils linker.



Obrázek 7.1: Kompatibilita knihoven

Uživatelé nástrojů Codasip Studio se nejspíše dostanou do situace na obrázku 7.1. Své programy překládají nástroji Codasip Studio. Ovšem v některých případech budou chtít využít knihovny, které získali buď zdarma na internetu nebo si je zakoupili. Tyto knihovny byly nejspíše vytvořeny nástroji GNU binutils a proto je není možné slinkovat s použitím Codasip linkeru.

Vzhledem k tomu, že uživatel pravděpodobně vlastní zdrojové kódy svého programu, mohl by jej také přeložit nástroji GNU binutils a následně jej také pomocí GNU linkeru slinkovat se získanou knihovnou.

Využití GNU překladače nemusí být ale možné vždy. Uživatel mohl pomocí Codasip IDE vytvořit model procesoru, který nějakým nestandardním způsobem rozšiřuje implementaci RISC-V, nejčastěji novými instrukcemi, které by rád ve svém programu využil. Nástroje GNU jsou vytvořeny s podporou pouze standardních instrukčních rozšíření, a proto při překladu programu nebudou tyto uživatelem vytvořené instrukce používat. Proto je nutné, aby svůj program přeložil Codasip překladačem. Díky rozšíření popsanému v této práci může vzniklé objektové soubory slinkovat pomocí GNU linkeru a získá kompletní spustitelný program.

7.2 Linux

V kontextu vestavěných systémů existují dvě velké skupiny zařízení. Jednu tvoří zařízení, na kterých je vykonáván pouze určitý program. Druhou tvoří zařízení, která mají nainstalován některý operační systém a v rámci tohoto operačního systému jsou následně spouštěny programy.

Nástroje Codasip Studio jsou zaměřeny především na první z těchto kategorií. Program, která tato zařízení vykonávají je po překladu umístěn do zařízení na základě překopírování dat objektového souboru na adresy v paměti zařízení, které při linkování vypočítal linker a následně zapsal také do objektového souboru. Vzhledem k tomu, že se možnosti přístupu do paměti zařízení mohou lišit je i proces nahrání programu do paměti často vytvořen na míru danému zařízení. Proto není problém, pokud je formát objektového souboru, respektive dat v něm uložených, něčím nestandardní.

V případě operačního systému je situace zcela jiná. Operační systém obsahuje program zvaný loader, který nakopíruje uživatelský program například z pevného disku do operační paměti, kde je následně vykonáván. V kontextu operačního systému však dochází k velkému množství abstrakcí, které mají za následek možnost mít v operační paměti takto umístěných programů několik.

Je tedy jasné, že loader je standardní součástí každého operačního systému a není prakticky možné jej upravit podobně, jako je možné upravit proces umístění programu do paměti u vestavěných zařízení bez operačního systému. Proto je nutné v případě potřeby spouštění uživatelských programů pod operačním systémem dodržet standardní formát objektových souborů tak, aby je loader daného operačního systému dokázal správně přečíst a umístit do paměti. Proto jsou úpravy popsané v této práci také nezbytné pro spouštění programů přeložených nástroji Codasip Studio pod operačním systémem Linux.

7.3 Optimalizace v GNU binutils linkeru

Při překladu programů se nejčastěji setkáváme s optimalizacemi prováděnými překladačem, respektive jeho částí označovanou jako optimalizátor. Tyto optimalizace se dějí na úrovni programovacího jazyka, případně na úrovni vnitřní reprezentace programu v překladači. Na základě principu překladu programů, který byl popsán v kapitole 2, je jasné, že překladač nemá k dispozici veškeré informace o celém programu a to z toho důvodu, že vždy překládá právě jeden modul programu a celý program může být složen z většího počtu modulů než je tento jeden.

Jediný nástroj překladového toolchainu, který má k dispozici kompletní program je linker. Optimalizace proveditelné v linkeru se velmi málo překrývají s těmi, které provádí překladač. Optimalizace prováděné překladačem často spočívají v přeskupení kódu, což je

v linkeru, kde je program reprezentován binárně, problematické. Optimalizace v linkeru se soustředí především na změny kódu, které je možné provést na základě znalosti hodnot symbolů a adres, které jsou známy až po provedení relokací. Snad i proto, že je tento přístup k optimalizacím tak odlišný oproti překladači, jsou tyto optimalizace nazývány vlastním pojmem - relaxace.

Codasip linker žádné takovéto relaxace neimplementuje. Relaxace jsou velmi závislé na cílové architektuře a ty implementované pro určitou architekturu nelze většinou využít na jiné. Codasip linker není nijak generován na základě popisu modelu, což by mohlo částečně umožnit generickou implementaci těchto relokací, která by byla při generování nástroje uzpůsobena na míru cílové architektury. Proto by bylo nutné všechny vyžadované relaxace implementovat ručně. Navíc by linker musel znát cílovou architekturu linkování, aby se správně rozhodl, které relokace použít.

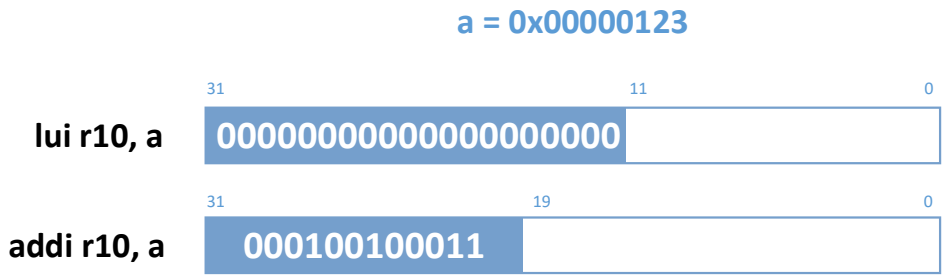
GNU linker oproti tomu obsahuje zásadní části zdrojového kódu implementované pro každou architekturu zvlášť, tak jak bylo vysvětleno v sekci 2.4. Navíc se jedná o open source nástroj umožňující potenciálně velkému množství vývojářů se podílet na jeho implementaci, což by mělo mít za následek její větší komplexnost.

Relaxace implementované v GNU linkeru se zaměřují především na snížení velikosti kódu programu. Program je možné zmenšit tím, že jsou z něj odstraněny některé nepotřebné instrukce. Tato problematika se pojí s assembler aliasy, které byly popsány v sekci 6.3.4.

Tyto aliasy často popisují kombinaci instrukcí, která umožní použít v jisté operaci přímý operand o velikosti celého instrukčního slova. Takový operand se do jedné instrukce nevejde, protože instrukční slovo musí kromě operandu obsahovat ještě alespoň operační kód. V určitých případech překladač nemůže určit jaké velikosti bude přímý operand a tak jednoduše počítá s nejhorší variantou, která odpovídá právě velikosti instrukčního slova na dané architektuře. Z toho důvodu jsou do kódu v jazyce symbolických instrukcí překladačem umístěny kombinace instrukcí, které zajistí, že program bude fungovat správě i pokud operand nabude takto velkých rozměrů.

Ve chvíli, kdy je kód relokován linkerem je už hodnota přímého operandu známa a tedy je možné určit, zdali bylo opodstatněné potřebné operace provádět pomocí kombinace instrukcí nebo bylo možné program přeložit do podoby, využívající méně instrukcí, ale pracující pouze s omezenou velikostí přímého operandu. Proto je vhodné, aby případnou zbytečnou opatrnost překladače napravil linker, který přebytečné instrukce odstraní.

Jelikož mohou být tyto optimalizace složitější na pochopení následuje příklad. Pomocí kombinace instrukcí *lui* a *addi*, která byla uvedena již dříve je možné do 32bitového registru načíst 32bitovou hodnotu. Instrukce *lui* naplní horních 20 bitů registru a instrukce *addi* poté spodních 12 bitů.



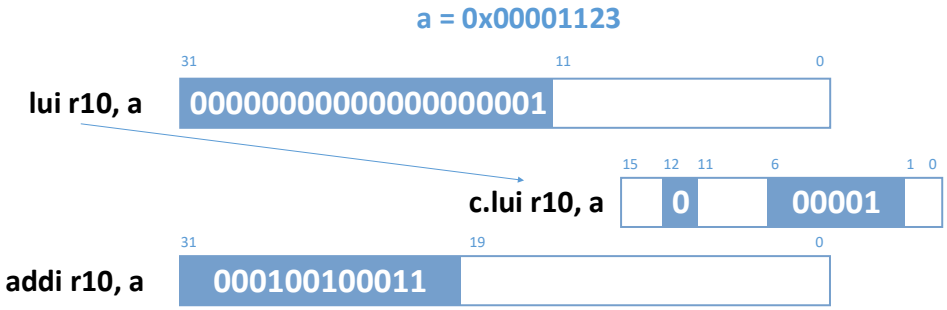
Obrázek 7.2: Možnost optimalizace načtení 32bitové konstanty do registru

Na obrázku 7.2 je přesně tato situace naznačena. Do registru *r10* je potřeba načíst hodnotu symbolu *a*. Řekněme, že při relokování kódu v linkeru je určeno, že symbol *a* nabývá hodnoty *0x123*. Tuto hodnotu je však možné vyjádřit pomocí 9 bitů. Instrukce *addi* může obsahovat přímý operand o velikosti až 12 bitů, což je dáno jejím instrukčním formátem, které jsou detailněji vysvětleny v sekci 4.1.

Instrukce *lui* tedy bude obsahovat přímý operand s hodnotou 0 a nebude mít tedy na výsledný obsah registru *r10* žádný efekt. Tato instrukce je tedy zbytečná a není v programu potřeba. Relaxace implementované v GNU linkeru dokáží takovou instrukci z programu odstranit, aniž by jinak program poškodily.

Optimalizace implementovaná v GNU linkeru je však ještě komplexnější. Řekněme, že symbol *a* nabývá tentokrát hodnoty *0x1123*, tak jak je zobrazeno na obrázku 7.3. Tato hodnota již nejde vyjádřit na 12 bitů, které by bylo možné uložit do instrukce *addi* a je tedy nutné využít také instrukci *lui*. Z přímého operandu instrukce *lui* takto bude využit pouhý jeden bit, protože hodnotu *0x1123* lze vyjádřit pomocí 13 bitů.

V komprimované instrukční sadě (viz sekce 4.2) existuje ekvivalent instrukce *lui*, označovaný *c.lui*. Tato instrukce je ze sémantického hlediska shodná s instrukcí *lui*. V instrukčním slově má však pro přímý operand vyhrazeno pouze 6 bitů oproti 20 bitům instrukce *lui* ze základní instrukční sady. Ovšem instrukce *c.lui* je velká pouze 16 bitů oproti 32 bitům instrukce *lui* a tak její použití zmenší celkovou velikost programu právě o 16 bitů.



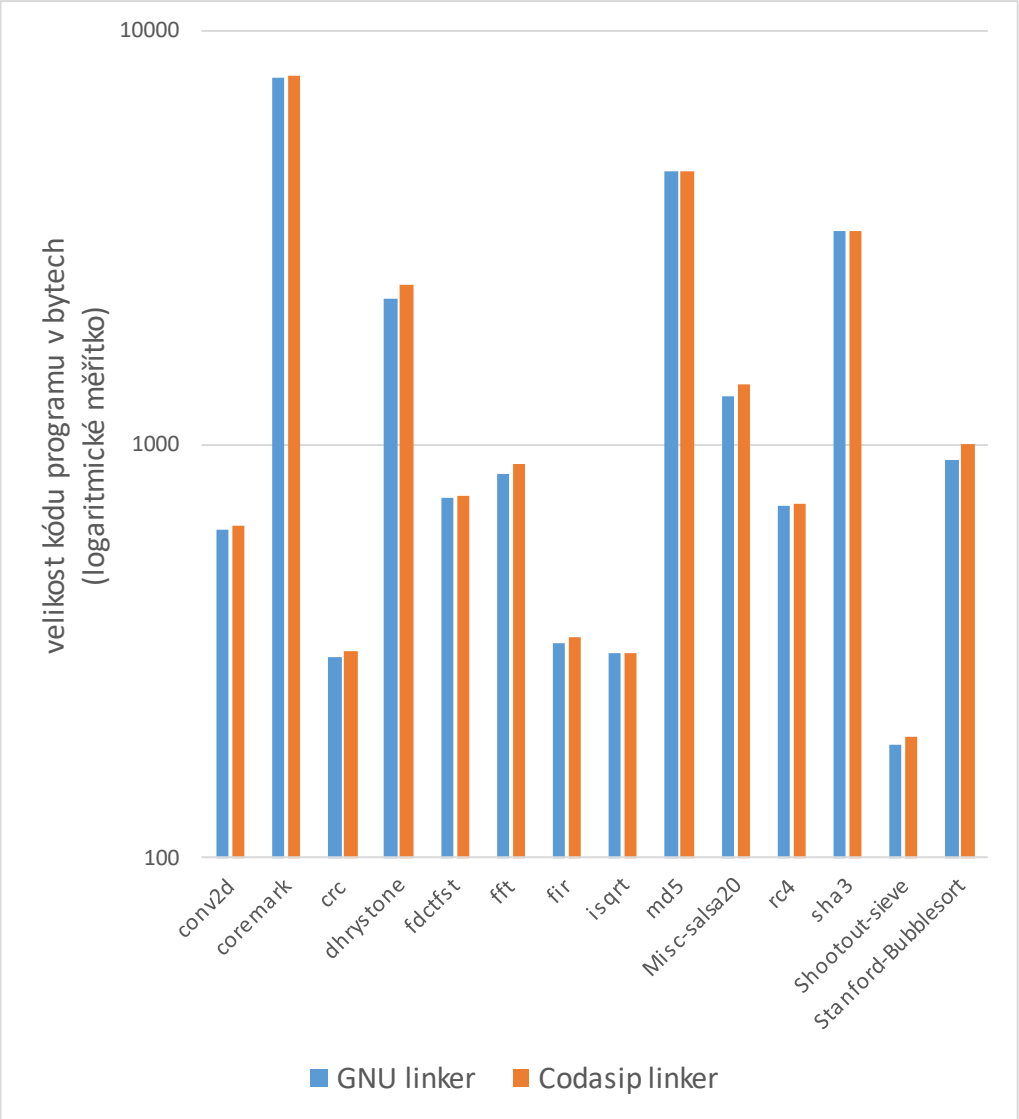
Obrázek 7.3: Možnost optimalizace načtení 32bitové konstanty do registru s použitím komprimované instrukce

Optimalizací vykonávaných GNU linkerem je více. Vedou především ke zmenšení výsledné binární podoby aplikace. Ve světě stolních počítačů a podobných zařízení toto již dávno není parametr, který by byl přísně sledován. Ovšem ve světě vestavěných systémů, kde možnost použití menší paměti znamená jak úsporu nákladů na vyrobený kus zařízení, tak menší velikost zařízení, je velikost programu stále podstatná.

Nahrazení instrukce za menší alternativu z komprimované instrukční sady vede pouze ke zmenšení programu. Kompletní odstranění nepotřebné instrukce však vede navíc ke zrychlení výpočtu programu. To, že instrukce nevykonává žádnou smysluplnou operaci totiž neznamená, že nezaměstnává procesor. Proto je odstranění nepotřebných instrukcí velmi vítáno.

Pro ověření výhodnosti použití GNU linkeru oproti Codasip linkeru byla s pomocí obou nástrojů přeložena sada benchmarků. U vzniklých spustitelných binárních souborů byla potom zjištěna velikost kódu samotného benchmarku bez kódu případných potřebných knihoven, které by mohly výsledky zkreslovat.

Graf na obrázku 7.4 ukazuje rozdíly v získaných velikostech kódu. Tyto rozdíly vznikly právě díky použití optimalizací, které byly popsány v předešlých odstavcích. V průměru je při použití GNU linkeru dosaženo o 3% menšího kódu. Ačkoliv se toto číslo může zdát malé, jedná se ve srovnání s účinností jednotlivých optimalizací implementovaných překladačem o velmi rozumný výsledek.



Obrázek 7.4: Porovnání velikosti kódu benchmarků při použití obou linkerů

I přes použití logaritmického měřítka mohou být z grafu rozdíly velikostí některých benchmarků špatně patrné. Proto jsou číselné hodnoty zaneseny do tabulky 7.1.

Sloupce *GNU linker* a *Codasip linker* obsahují velikosti kódu jednotlivých benchmarků v bytech. Sloupec *GNU / Codasip* poté obsahuje porovnání těchto velikostí. V případě, že je výsledek tohoto porovnání menší než 1, znamená to, že použití GNU linkeru produkuje

menší kód výsledné aplikace. Tučně je poté pod posledním sloupcem hodnota označující průměr těchto poměrů.

Tabulka 7.1: Velikosti kódu benchmarků při použití obou linkerů

Název benchmarku	GNU linker	Codasip linker	GNU / Codasip
conv2d	622	638	0,97
coremark	7738	7780	0,99
crc	304	316	0,96
dhrystone	2248	2444	0,92
fdctfst	744	752	0,99
fft	850	896	0,95
fir	330	340	0,97
isqrt	312	312	1
md5	4598	4602	1
Misc-salsa20	1312	1400	0,94
rc4	708	718	0,99
sha3	3268	3280	1
Shootout-sieve	188	196	0,96
Stanford-Bubblesort	914	1002	0,91
průměr			**0,97**

Kapitola 8

Závěr

Obsah této práce je v zásadě možné rozdělit na dvě části. První se zabývá zajištěním podpory architektury RISC-V v rámci nástrojů Codasip Studio. Jedná se tedy o implementaci nového systému relokací, který je v rámci této práce označován jako univerzální relokace. Toto označení je velmi přesné v tom smyslu, že takto implementované relokace jsou použitelné obecně a rozšíření tedy není limitováno jen na architekturu RISC-V. Ačkoliv se jedná o změnu, která byla motivována podporou této architektury v nástrojích Codasip Studio, je zřejmé, že by byla časem potřebná i pro jiné architektury modelované pomocí Codasip Studio.

Na základě úspěšné implementace tohoto rozšíření tak, jak je popsáno v této diplomové práci, byla tímto systémem relokací nahrazena původní implementace použitá v nástrojích Codasip Studio a nyní již zákazníci, kteří tyto nástroje využívají, pracují s implementací popsanou zde. V tomto smyslu je možné říci, že tato část přesahuje zadání této diplomové práce.

Významná část tohoto textu se zabývá další úpravou nástrojů Codasip Studio, která se tentokrát soustředí pouze na assembler a cílí na zajištění kompatibility produkovaných objektových souborů s nástroji GNU binutils. Proto je v kontextu této práce označována jako kompatibilní relokace. Logicky tato část navazuje na tu předešlou, protože ke své funkci využívá některé její části.

Implementace kompatibilních relokací umožnila uživatelům překladových nástrojů vygenerovaných pomocí Codasip Studio používat v rámci svých programů knihovny, které byly přeloženy nástroji GNU bintuils. Dále toto implementované rozšíření otevírá uživatelům Codasip Studio možnost uvažovat o provozování operačního systému Linux na procesorových jádrech, která navrhují pomocí Codasip Studio.

Ačkoliv bylo toto rozšíření implementována za účely popsanými v předešlém odstavci. přináší ještě další výhodu. Pro architekturu RISC-V je nyní možné při překladu programů používat místo linkeru vygenerovaného pomocí Codasip Studio linker GNU binutils. Vzhledem k tomu, že je druhý jmenovaný nástroj vyvíjen širší komunitou vývojářů nabízí propracovaný systém optimalizací, který není v Codasip alternativě dostupný. Díky použití GNU binutils linkeru tedy dochází ke zmenšení výsledné binární podoby programu a tím snížení nároků na paměť procesorového jádra. Tato úspora paměti není vůbec zanedbatelná, protože přináší v průměru zmenšení kódu o 3% a v případě některých měřených benchmarků se blíží dokonce hranici 10%.

Literatura

[1] *CodAL Language Reference Manual*. [Online; navštíveno 14.12.2016].

[2] *Codasip Cycle Accurate Model Tutorial*. [Online; navštíveno 14.12.2016].

[3] *Codasip Studio Quick Start Tutorial*. [Online; navštíveno 14.12.2016].

[4] *Executable and Linkable Format*. [Online; navštíveno 09.01.2017].
URL http://www.skyfree.org/linux/references/ELF_Format.pdf

[5] *GNU Binutils*. [Online; navštíveno 21.5.2017].
URL https://www.gnu.org/software/binutils/

[6] *GNU Bison*. [Online; navštíveno 21.5.2017].
URL https://www.gnu.org/software/bison/

[7] Asanović, K.: *RISC-V Workshop Introductions and RISC-V Update*. [Online; navštíveno 25.12.2016].
URL http://riscv.org/wp-content/uploads/2015/06/riscv-intro-workshop-june2015.pdf

[8] Hennessy, J. L.; Patterson, D. A.: *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 2007, ISBN 0123704901.

[9] Levine, J. R.: *Linkers and Loaders*. Morgan Kaufmann, 1999, ISBN 1558604960.

[10] Mishra, P.; Dutt, N.: *Processor Description Languages*. Morgan Kaufmann, 2008, ISBN 9780123742872.

[11] Waterman, A.; Lee, Y.; Patterson, D.; aj.: *The RISC-V Instruction Set Manual Volume I: User-Level ISA Version 2.1*. [Online; navštíveno 25.12.2016].
URL https://riscv.org/specifications/

Přílohy

Příloha A

Obsah DVD

Přiložené DVD obsahuje následující adresáře:

- */doc* - Obsahuje technickou zprávu ve formátu PDF.
- */tex* - Obsahuje zdrojové kódy technické zprávy a obrázky v ní použité.
- */src* - Obsahuje zdrojové kódy vztahující se k implementaci popsané v této práci.
- */demo* - Obsahuje ukázku funkčnosti rozšíření popsaných v této práci. Použití této ukázky je vysvětleno v souboru README.