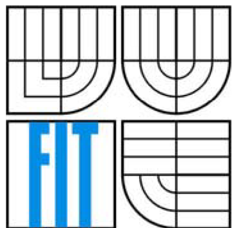
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



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

IMPLEMENTACE OBECNÉHO ASSEMBLERU

UNIVERSAL ASSEMBLER IMPLEMENTATION

DIPLOMOVÁ PRÁCE
MASTER‘S THESIS

AUTOR PRÁCE ADAM HUSÁR
AUTHOR

VEDOUCÍ PRÁCE PROF. ING. TOMÁŠ HRUŠKA, CSC.
SUPERVISOR

BRNO 2007

Zadání diplomové práce

Vedoucí:
Hruška Tomáš, prof. Ing., CSc., UIFS FIT VUT

Oponent:
Masařík Karel, Ing., UIFS FIT VUT

Přihlášen:
Husár Adam

Zadání:

1. Seznamte s se jazyky pro popis mikroprocesorů (nML, případně novější z rodiny LISA) definující vstupní a výstupní tvar assembleru.
2. Seznamte se jazykem pro popis mikroprocesoru v projektu Lissom a jeho vnitřním modelem.
3. Implementujte model obecného assembleru, přičemž využijte navržené struktury z ročníkového projektu.
4. Diskutujte případné možné další rozšíření implementovaného modelu obecného assembleru.

Část požadovaná pro obhajobu SP:
Splnění bodů 1 a 2.

Kategorie:
Překladače

Literatura:
Hruška T. (2004). Instruction Set Architecture C. FIT VUT Brno.

LICENČNÍ SMLOUVA
POSKYTOVANÁ K VÝKONU PRÁVA UŽÍT ŠKOLNÍ DÍLO

uzavřená mezi smluvními stranami:

1. Pan/paní

Jméno a příjmení: Adam Husár

Bytem: Klimentov 92, 354 71 Velká Hleďsebe

Narozen/a (datum a místo): 22. ledna 1983, Mariánské Lázně

(dále jen „autor“)

a

2. Vysoké učení technické v Brně

Fakulta informačních technologií

se sídlem Božetěchova 2, 612 66

jejímž jménem jedná na základě písemného pověření děkanem fakulty:

....

(dále jen „nabyvatel“)

Čl. 1
Specifikace školního díla

1. Předmětem této smlouvy je vysokoškolská kvalifikační práce (VŠKP):
   - ☐ disertační práce
   - ☑ diplomová práce
   - ☐ bakalářská práce
   - ☐ jiná práce, jejíž druh je specifikován jako ....

   (dále jen VŠKP nebo dílo)

Název VŠKP:	Implementace obecného assembleru
Vedoucí/ školitel VŠKP:	prof. Ing. Tomáš Hruška, CSc.
Ústav:	Ústav informačních systémů
Datum obhajoby VŠKP:	18. května 2007

VŠKP odevzdal autor nabyvateli v*:

☐ tištěné formě – počet exemplářů 2

☐ elektronické formě – počet exemplářů 2

---

* hodící se zaškrtněte

2. Autor prohlašuje, že vytvořil samostatnou vlastní tvůrčí činností dílo shora popsané a specifikované. Autor dále prohlašuje, že při zpracovávání díla se sám nedostal do rozporu s autorským zákonem a předpisy souvisejícími a že je dílo dílem původním.

3. Dílo je chráněno jako dílo dle autorského zákona v platném znění.
4. Autor potvrzuje, že listinná a elektronická verze díla je identická.

Článek 2
Udělení licenčního oprávnění

1. Autor touto smlouvou poskytuje nabyvateli oprávnění (licenci) k výkonu práva uvedené dílo nevýdělečně užít, archivovat a zpřístupnit ke studijním, výukovým a výzkumným účelům včetně pořizovaní výpisů, opisů a rozmnoženin.
2. Licence je poskytována celosvětově, pro celou dobu trvání autorských a majetkových práv k dílu.
3. Autor souhlasí se zveřejněním díla v databázi přístupné v mezinárodní síti
   □ ihned po uzavření této smlouvy
   □ 1 rok po uzavření této smlouvy
   □ 3 roky po uzavření této smlouvy
   □ 5 let po uzavření této smlouvy
   □ 10 let po uzavření této smlouvy
   (z důvodu utajení v něm obsažených informací)
4. Nevýdělečné zveřejňování díla nabyvatelem v souladu s ustanovením § 47b zákona č. 111/ 1998 Sb., v platném znění, nevyžaduje licenci a nabyvatel je k němu povinen a oprávněn ze zákona.

Článek 3
Závěrečná ustanovení

1. Smlouva je sepsána ve třech vyhotoveních s platností originálu, přičemž po jednom vyhotovení obdrží autor a nabyvatel, další vyhotovení je vloženo do VŠKP.
2. Vztahy mezi smluvními stranami vzniklé a neupravené touto smlouvou se řídí autorským zákonem, občanským zákoníkem, vysokoškolským zákonem, zákonem o archivnictví, v platném
   znění a popř. dalšími právními předpisy.
3. Licenční smlouva byla uzavřena na základě svobodné a pravé vůle smluvních stran, s plným porozuměním jejímu textu i důsledkům, nikoliv v tísni a za nápadně nevýhodných podmínek.
4. Licenční smlouva nabývá platnosti a účinnosti dnem jejího podpisu oběma smluvními stranami.

V Brně dne: 16. května 2007

……………………………………….. …………………………………………
Nabyvatel Autor

Abstrakt

Tato diplomová práce se zabývá návrhem obecného assembleru, který je součástí projektu Lissom. Naleznete zde popis architektur assemblerů, jejich obvyklých úkolů a zvláštní pozornost je pak věnována assembleru GNU as. Navržený assembler se skládá z pevné a generované části. Generovaná část je automaticky vytvářena na základě popisu instrukční sady, která je definována pomocí jazyka pro popis architektury a instrukční sady ISAC. Využitím tohoto přístupu je umožněno automaticky změnit cílovou architekturu, pro kterou assembler překládá. Další část práce pak popisuje implementaci knihovny Parserlib2, která je využívána generátorem assembleru a i dalšími součástmi projektu Lissom a poskytuje informace o cílové instrukční sadě.

Klíčová slova

Assembler, obecný assembler, univerzální assembler, retargetabilní assembler, cross assembler, jednoprůchodový assembler, dvouprůchodový assembler, architektura assembleru, návrh assembleru, Lissom, ISAC, LISA, instrukční sada, jazyky pro popis architektury a instrukční sady, ADL, procesor s aplikačně specifickou instrukční sadou, ASIP, nástroj pro návrh procesorů, dvojcestné párové automaty, relokace, relaxace, bitová oprava, zpracování výrazů assemblerem, direktiva, pseudooperace, zpracování direktiv, Parserlib2, vnitřní model jazyka ISAC.

Abstract

This thesis describes the design of the universal assembler that represents a part of the Lissom project. You will be provided with the description of the assembler architectures and their usual tasks. Special attention is paid to GNU assembler. Designed assembler consists of the fixed and the generated part. The generated part is created automatically from the description of instruction set, that is defined using architecture and instructions set description language ISAC. Using this approach, it is possible to change assembler target architecture automatically. The second part of thesis describes the Parserlib2 library implementation that is a part of the Lissom project and provides the information about the target instruction set for an assembler generator.

Keywords

Assembler, universal assembler, retargetable assembler, cross-assembler, one-pass assembler, two-pass assembler, assembler architecture, assembler design, Lissom, ISAC, LISA, instruction set, architecture and instruction set description language, ADL, application-specific instruction set processor, ASIP, processor design tools, two-way coupled finite automata, relocation, relaxation, fix-up, assembler expression processing, directive, pseudo-operation, directive handling, Parserlib2, internal ISAC language model.

Citace

Husár Adam: Implementace obecného assembler. Brno, 2007, diplomová práce, FIT VUT v Brně.

Implementace obecného assembleru

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením prof. Ing. Tomáše Hrušky, CSc.
Další informace mi poskytli Ing. Karel Masařík, Ing. Roman Lukáš, Ph.D., Doc. Dr. Ing. Dušan Kolář a Bc. Zdeněk Přikryl.
Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……………………
Adam Husár
22. května 2007

Poděkování

Na tomto místě bych rád poděkoval prof. Ing. Tomáši Hruškovi, CSc., vedoucímu mé diplomové práce, za jeho cenné rady. Také bych chtěl poděkovat Ing. Romanu Lukášovi, Ph.D. a Doc. Dr. Ing. Dušanu Kolářovi za další odbornou pomoc při konzultacích a Bc. Zdeňku Přikrylovi a také dalším členům teamu Lissom, s nimiž jsem na projektu spolupracoval.

© Adam Husár, 2007.
Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Obsah

Obsah 1
Úvod 3

1 Projekt Lissom a jazyky pro popis architektury 4
1.1 Cíl projektu Lissom 4
1.2 Projekty LISA a Chess/Checkers 5
1.3 Jazyky pro popis architektury, členění 6
1.4 Instrukce a operace v jazycích pro popis architektury 7
1.5 Jazyk ISAC a projekt Lissom 8
1.5.1 Jazyk ISAC, verze 0.0 8
1.5.2 Jazyk ISAC, verze 0.1 9

2 Assemblery a instrukční sady 11
2.1 Účel assembleru 11
2.2 Obecné assemblery 11
2.3 Instrukční sady 11
2.3.1 Různá přirovnání k instrukčním sadám 11
2.3.2 Návrh architektury instrukční sady 12

3 Assemblery 13
3.1 Literatura o obecných assemblerech 13
3.2 Jedno a dvouprůchodové assemblery 13
3.2.1 Jednoprůchodový assembler 14
3.2.2 Dvouprůchodový assembler 14
3.3 Pojmy sekce, location counter a relokace 14
3.3.1 Sekce 14
3.3.2 Ukazatel aktuální pozice – location counter (LC) 15
3.3.3 Symboly 15
3.3.4 Relokace 15

4 Příklady obecných assemblerů a GNU assembler 20
4.1 Některé obecné assemblery 20
4.2 GNU as 20
4.3 Kroky překladu 21
4.4 Subsekce 21
4.5 Struktury GNU as 22
4.6 Změna cílové platformy 23
4.7 Relaxace 24

5 Shrnutí předchozího vývoje, dvojcestné párové automaty 25
5.1 Verze 0.1 25
5.2 Verze 0.2 26
5.3 Propojení generovaného jazyka instrukcí a jazyka assembleru ve verzích 0.1 a 0.2 27
5.3.1 Jazyk instrukční sady (JI) 27
5.3.2 Jazyk assembleru (JA) 27
5.3.3 Propojení jazyků JA a JI 27
5.3.4 Možné řešení přidání podpory výrazů pro assembler verze 0.1 a 0.2 28
5.4 Překlad instrukcí pomocí dvojcestných párových automatů 29
5.4.1 Co jsou to dvojcestné párové automaty (DPA) 29
5.4.2 Překlad pomocí DPA 30
5.4.3 Výhody překladu pomocí dvojcestných párových automatů 32
5.4.4 Jaké jsou hlavní rozdíly při změně algoritmu pro překlad instrukcí 32

6 Návrh assembleru 33
6.1 Cíle assembleru projektu Lissom 33
6.2 Postup při překladu 33
6.3 Assembler a jeho součásti 34
6.4 Lexikální a syntaktický analyzátor a kodér instrukcí 36
6.4.1 Příklad generovaných částí lexikálního a syntaktického analyzátoru 38
6.4.2 Rozhraní mezi syntaktickým analyzátorem a kodérem instrukcí, jednotkou pro zpracování direktiv a jednotkou pro zpracování symbolů výrazů 41
6.5 Jednotka pro zpracování direktiv 42
6.6 Jednotka pro zpracování symbolů a výrazů 42
6.6.1 Symboly 42
6.6.2 Symboly, výrazy a jejich atributy 44
6.6.3 Operátory a tok atributů 45
6.6.4 Návrh reprezentace symbolů a výrazů 48
6.6.5 Funkce a rozhraní jednotky pro zpracování výrazů 49
6.7 Jednotka pro ukládání dat do tabulky sekcí a bitových oprav 49
6.7.1 Tabulka sekcí 50
6.7.2 Tabulka bitových oprav 51
6.8 Jednotka pro vyřešení bitových oprav a pro transformaci přeložených informací do objektového souboru 51
6.9 Zhodnocení návrhu assembleru 51

7 Závěr 53

Literatura 54

A Knihovna Parserlib2, třídy pro uložení informací o operacích a skupinách 58
A.1 Nová verze Parserlib2 58
A.1.1 Požadavky kladené na třídy Parserlib2 58
A.1.2 Důvody pro novou verzi 58
A.2 Vnitřní model jazyka ISAC 59
A.2.1 Účel vnitřního modelu 59
A.2.2 Současný stav formátu vnitřního modelu, popis jeho diagramu 60
A.2.3 Zhodnocení současného formátu vnitřního modelu, návrhy pro vylepšení 62
A.3 Architektura tříd pro uložení informací o operacích a skupinách 63
A.3.1 Principy 63
A.3.2 Architektura tříd 63
A.3.3 Rozhraní tříd operací 66
A.4 Architektura parseru vnitřního modelu pro operace a skupiny 71

B Direktivy jazyka assembleru, jejich význam a syntaxe 75
C Fáze vývoje assembleru 80
D Diagram komponent assembleru 83
E Struktury GNU as 84
F Formát vnitřního modelu procesoru 85
F.1 Vnitřní model – část operací 85
F.2 Vnitřní model – část zdrojů 86
G Diagramy knihovny Parserlib2 87
G.1 Diagram tříd operací Parserlib2 88
G.2 Diagram pomocných tříd představujících členské prvky tříd operací Parseerlib2 89
G.3 Příklad uložení informací o jedné operaci do tříd operací Parserlib2 90
G.4 Diagram volání funkcí XML parseru knihovny Parserlib2 pro část operací 91
G.5 Diagram tříd XML parseru knihovny Parserlib2 pro část operací 92

Úvod

V současnosti si již svět bez spotřební elektroniky asi těžko dokážeme představit. Ve všech takovýchto zařízeních, pouze kromě těch nejjednodušších, je použit alespoň jeden mikroprocesor. Často to bývají procesory specializované na určitou činnost, které se vyvíjejí pro tu jednu konkrétní aplikaci. Návrh procesorů od dob, kdy se ručně tvořil například dnes již legendární procesor Z80, samozřejmě pokročily, avšak stále, i s využitím jazyků pro popis hardwaru a syntetizátorů, je návrh nových procesorů náročným úkolem, který vyžaduje desítek specialistů a několik let vývoje.

S rostoucí složitostí přestávají jazyky jako je VHDL, kvůli svému nízkému stupni abstrakce, stačit. Dalším logickým krokem, či novým stupněm evoluce, je tedy vynalézt nový jazyk, který by procesory popisoval abstraktněji, avšak stále s dostatečnou přesností.

Na Fakultě informačních technologií Vysokého učení technického v Brně byl projekt s tímto cílem založen. Jmenuje se Lissom a jeho účelem je, kromě návrhu nového jazyka ISAC pro popis architektury procesoru, vytvořit kompletní vývojové prostředí. To pak umožní efektivněji navrhovat aplikačně specifické procesory, automaticky vytvářet softwarové nástroje pro danou architekturu a několikanásobně zkrátit dobu vývoje oproti klasickým postupům. Je to nelehký úkol, avšak úspěchy podobně zaměřených projektů ukazují, že je stanovený cíl je reálný a výsledek projektu jistě nalezne využití.

Tato diplomová práce se zabývá jednou ze součástí projektu Lissom a to konkrétně assemblerem. Jeho část je generována na základě popisu instrukční sady procesoru pomocí jazyka ISAC a umožňuje především překládat mnemotechnický zápis instrukcí do jejich binární podoby. Dále se skládá z pevné části, ta se stará o zpracování direktiv, výrazů a generování objektového souboru pro linker.

V první kapitole naleznete shrnutí cílů projektů Lissom, pojednání o různých jazycích pro popis architektury a stručný popis jazyka ISAC. V navazující druhé kapitole je krátký úvod o obecných assemblerech a instrukčních sadách. Ve třetí kapitole se pak podíváme na assemblery, jaké operace obvykle provádějí a z čeho se skládají. Čtvrtá kapitola je pak věnována nejpoužívanějšímu obecnému či retargetabilnímu assembleru GNU as.

Kapitola pátá obsahuje shrnutí předchozího vývoje assembleru a v šesté, asi nejdůležitější kapitole, je popsán návrh assembleru, jeho jednotlivé součásti a jejich rozhraní.

Dále, příloha A se věnuje implementaci jedné části projektu, která assemblerem bude využívána, a to knihovně Parserlib2. V dalších přílohách pak, kromě seznamu direktiv, které assemblerem budou podporovány, naleznete různé diagramy, které doplňují popis návrhu assembleru a implementace knihovny Parserlib2.

1 Projekt Lissom a jazyky pro popis architektury

1.1 Cíl projektu Lissom

Cílem projektu je vytvořit kompletní vývojové prostředí pro návrh procesorů. Pomocí speciálního jazyka ISAC pro popis architektury si návrhář popíše požadovaný procesor. Pak, na základě popisu procesoru, je možné automaticky vygenerovat nástroje potřebné pro vývoj programového vybavení, ladění a také k ověřování platnosti požadavků kladených na procesor. Jsou to především tyto nástroje: assembler, disassembler, debugger a simulátor.

Odstraněno: rychle

Když už se někdo rozhodne, že potřebuje vytvořit procesor s aplikačně specifickou instrukční sadou (ASIP, Application Specific Instruction Set Processor), tak má v plánu jich vyrobit veliký počet (např. milióny). U takovéhoto počtu je velice důležitým hlediskem cena výsledného procesoru. Navíc, pro cílovou aplikaci si výrobce přeje, aby výsledný procesor měl co nejmenší velikost a pouze minimální nutné množství paměti. V případě použití v přenosných zařízeních je pak dalším hlediskem spotřeba. Ta přímo souvisí s efektivitou. Procesor se specializovanou instrukční sadou nepotřebuje tolik taktů k provedení určité práce na specifickém problému, může tedy běžet na nižší frekvenci a má nižší spotřebu.

Odstraněno: n

Odstraněno: , p

Zde jsou shrnuty hlavní důvody, proč se nové ASIP procesory vytvářejí:

- cena výroby jednoho procesoru,
- co nejmenší potřebná paměť pro aplikaci, s tím souvisí specializované kódování instrukční sady,
- a poměr výkon/spotřeba u specifických aplikací.

Dalším důvodem by mohl být celkový výkon, ale ve většině případů je, dle mého názoru, jednodušší využít některý z již existujících procesorů, protože vysoce výkonné procesory jsou více složité a technologie pro jejich výrobu mnohem dražší. Trendy ve vývoji výkonných procesorů v poslední době směřují ke zvyšování paralelizmu. To jak na instrukční úrovni (ILP, Instruction-Level Paralelism), tak na úrovni vláken (TLP, Thread-Level Paralelism). Na instrukční úrovni jde především o superskalární procesory s mnoha úrovněmi zřetězení (deep pipeline) a dále VLIW procesory. Paralelizmus na úrovni vláken se pak týká více-jádrových procesorů. Jazyk ISAC umožňuje popsat architekturu procesorů využívajících ILP. Také je možné pomocí něj navrhnout jedno jádro vícejádrového procesoru.

Odstraněno: -

Pro ASIP procesor se stanoví určitá kritéria vycházející z důvodů uvedených výše (cena, výkon, spotřeba). Vývoj pak probíhá ve více iteracích, kdy se vývojáři snaží dosáhnout požadovaných vlastností a vyhovět stanoveným omezením.

Projekt Lissom si klade za cíl tento proces návrhu a vývoje nového ASIP procesoru co nejvíce urychlit tím, že poskytne nástroje pro analýzu vlastností procesoru. Ty pak umožní zkrátit dobu

vývojových iterací. Ve finální fázi pak urychlí vytvoření hardwarového popisu procesoru automatickým generováním některých z jeho součástí.

Souběžně s vývojem procesoru je také možné psát výslednou aplikaci a testovací programy pro tento procesor s využitím automaticky generovaného assembleru, disassembleru a debuggeru. V budoucnu se také bude generovat překladač vyššího programovacího jazyka C.

1.2 Projekty LISA a Chess/Checkers

Jazyk ISAC (Instruction Set Architecture C) vychází z jazyka LISA (Language for Instruction Set Architectures). Jazyk LISA byl původně vyvinut na Institutu pro integrované systémy pro zpracování signálů Technické university v Aachenu (IIS RWTH Aachen). Původní motivací (v roce 1996) bylo odstranění pracného ručního vytváření simulátorů instrukčních sad. Návrh jazyka byl tedy silně ovlivněn potřebami pro automatické generování simulátorů. V dalších letech byl jazyk obohacen tak, aby bylo s jeho pomocí možné vytvářet automaticky nástroje pro generování kódu jako je assembler a linker a nástroje pro analýzu (profiling tools). Další rozšíření se týkaly podpory generování implementace procesoru – hardwarového popisu. Jedním z hlavních cílů při návrhu jazyka LISA a odpovídajících nástrojů se pak stala schopnost popsat širokou škálu různých architektur – od DSP procesorů, přes mikrokontrolery až po účelově specializované architektury. Andreas Hoffmann, jeden z hlavních členů týmu, odhaduje, že od započetí, do přelomu roku 2002/2003 si projekt vyžádal 40 člověkolet práce studentů doktorského studia. To vše bez započítání práce studentů, kteří na projektu pracovali v rámci svých semestrálních a diplomových prací. (podle knihy Architecture Exploration for Embedded Processors with LISA [1])

V roce 2001 byla založena společnost LisaTEK, která se zabývala komerčním využitím výsledků projektu LISA. Dále pak v lednu 2003 byla společnost LisaTEK koupena mezinárodní společností CoWare (www.coware.com), která začlenila výsledky práce do svých produktů. CoWare se zabývá, jak vývojem nástrojů pro HW/SW co-design a návrh procesorů, tak samotným vytvářením konkrétních aplikací a spolupracuje s největšími výrobci procesorů a spotřební elektrotechniky jako je ARM, ST Microelectronics, Sony, Toshiba a další.

Dalším příkladem podobně zaměřeného projektu může být Chess/Checkers. Ten je založen na ADL jazyce nMl (viz The NML Machine Description Formalism [2], Describing Instruction Set Processors Using nML [3]). Přibližně od roku 1990 začal vývoj tohoto jazyka na Technické univerzitě v Berlíně. Později, ve spolupráci s belgickým vývojovým centrem IMEC (Interuniversities Microelectronics Center), se začal vyvíjet systém Chess/Checkers. Pro komerční využití projektu pak byla v roce 1996 v Belgii založena společnost Target (www.retarget.com). V roce 1998 měli hotovou první ostrou verzi aplikace a od té doby se projekt stále úspěšně rozvíjí. V současnosti společnost Target spolupracuje s firmami jako je Atmel či Philips a také na konci roku 2006 zřídila své pobočky a vývojová centra v Severní Americe a Izraeli.

Na příkladu dvou uvedených projektů je vidět, že cíl stanovený projektem Lissom je velice reálný. Jak ukazuje zájem firem ARM či Atmel, je stále potřeba vyvíjet nové procesory a nástroje, které urychlí vývoj, programování a verifikaci aplikačně specifických procesorů a jejich aplikací jsou žádané. Otázkou je, jak uspět v takovémto prostředí, kde mají naši konkurenti mnohaletý náskok.

Možným směrem by mohl být návrh a simulace vícejádrových procesorů (i s heterogenními jádry) a podpora ladění paralelních aplikací, to si však vyžádá ještě dlouhý vývoj.

Odstraněno: -

1.3 Jazyky pro popis architektury, členění

Tradičně se jazyky pro popis architektury (dále jen ADL) dají členit do tří kategorií. Tato klasifikace je založena na povaze informace poskytované popisem v jistém jazyce či úrovni abstrakce. Jsou to tyto kategorie:

- strukturální,
- behaviorální/zaměřené na instrukční sadu,
- a smíšené jazyky/jazyky pro popis instrukční sady a architektury.

V této podkapitole čerpám především ze dvou zdrojů: prvním je zajímavý článek Architecture Description Languages for Retargetable Compilation [4], kde můžete nalézt popisy různých ADL jazyků, jejich analýzu a výčet nesnází, se kterými se návrháři těchto jazyků potýkají. Druhým zdrojem je prezentace Jazyky pro popis architektury počítačových systémů [5], kde jsou, mimo jiné, vyčerpávajícím způsobem shrnuty existující ADL jazyky, jejich výhody a nevýhody.

Díky veliké rozdílnosti architektur procesorů je složité nalézt rozumný kompromis při návrhu ADL jazyka. Stojí zde proti sobě především úroveň abstrakce a obecnost jazyka. Obvyklým způsobem, jak docílit větší obecnosti, je snížit úroveň abstrakce. Nižší úroveň abstrakce umožňuje detailněji a přesněji popsat jednotlivé součásti, avšak za cenu vyšší složitosti popisu. Dalším důležitým aspektem ADL jazyka je jeho schopnost popsat veškeré potřebné informace jak pro generátory kódu, simulátory a syntetizátory.

Strukturální jazyky jsou nízkoúrovňovými jazyky a příkladem může být jazyk MIMOLA. Hardwarové struktury jsou popsány v jazyce podobném VHDL a pomocí speciálních klíčových slov jsou označeny součásti důležité pro generátory kódu a simulátor, jako je program counter či instrukční paměť. Dalšími příklady strukturálních jazyků mohou být AIDL a UDL/I. Jejich nevýhodou je, že často obsahují málo informací pro generátory kódu, jako je například syntaxe jazyka instrukční sady a zase pro naopak potřeby rychlé simulace jsou příliš detailní. Umožňují však přesně specifikovat jednotlivé hardwarové součásti.

Odstraněno: -

Kategorie behaviorálních jazyků či jazyků zaměřených na instrukční sadu jsou zaměřeny především na popis chování procesoru. Detailní popis jednotlivých hardwarových součástí neumožňují. Jsou vhodné pro vytváření generátorů kódu a simulátory. Příkladem může být jazyk nMl, který je formálním jazykem založeným částečně na popisu v podobě jazyka RTL (Register Transfer Language, používá se často jako intermediární kód v překladačích vyšších programovacích jazyků). Syntaxe instrukční sady je popsána jako atributová gramatika. Dalšími příklady behaviorálních jazyků jsou ISDL, CSDL, Valen-C.

Smíšené jazyky či jazyky pro popis instrukční sady a architektury rozšiřují behaviorální jazyky o možnost popsat hardwarové zdroje. V poslední době je vývoj soustředěn především na tyto typy jazyků, protože je s jejich pomocí možné popsat všechny informace potřebné pro nástroje jako jsou

generátory kódu, simulátory a syntetizátory. Zástupci smíšených jazyků jsou tyto: ISAC, LISA, Maril, HMDES, TDL, FlexWare, PEAS, RADL, EXPRESSION, MetaCore a ASIA.

1.4 Instrukce a operace v jazycích pro popis architektury

Nejprve si definujme dva pojmy: instrukce a operace. Instrukce je základní jednotka, do které si zakódujeme to, co chceme aby procesor dělal. Ta se pak při spuštění přeloží do jedné či více operací, které jsou vykonány hardwarem procesoru. (blíže viz [6], str. 106).

V článku [4] uvádějí zajímavou možnost popisu operací pomocí trojice obsahující chování, zdrojový prvek a čas. Tu zde nyní uvedu a pokusím se uvést, čemu tato trojice odpovídá v jazyce ISAC.

Každá operace procesoru provádí nějakou změnu stavu v procesoru. Přesný popis takovéhoto stavového přechodu musí obsahovat tři prvky popisující co, kde a kdy se děje. Pro překladač to mohou být tyto: chování, zdroj a čas. Zde, chování odpovídá sémantické akci, která se skládá ze čtení zdrojových operandů, výpočtu a zápisu výsledků, zdroj odpovídá hardwarovému prvku, který je operací využíván, tuto informaci potřebujeme pro určování hazardů (souběžného využívání jednoho zdrojového prvku). Poslední z trojice – čas – určuje například číslo cyklu, kdy se chování vykonává. Obvykle je popsán relativně k času, kdy byla operace načtena (fetch). Používá se pro modelování času.

Pomocí těchto tří základních elementů můžeme jednoduše popsat každou instrukci množinou takovýchto trojic. Popis operace má takovýto tvar:

operace = (chování, zdroj, čas)

Jednotlivé instrukce jsou pak množinou operací:

instrukce = {operace1, operace2, ...}

Například instrukci Add je možné popsat následujícím způsobem, čas je relativní k načtení operace:

Add = {(načti operand *reg[src1]*, pomocí *portu a registrového pole*, v *2. cyklu*),
(načti operand *reg[src2]*, pomocí portu *b* registrového *pole*, v *2. cyklu*),
(proveď součet, v *jednotce alu*, ve *3. cyklu*),
(zapiš operand *reg[dst],* pomocí *zápisového portu registrového pole*, v *5. cyklu*)}

V určité podobě lze tyto trojice nalézt ve všech smíšených jazycích, dokonce i v jazyce ISAC. V sekci ACTIVATION definujeme seznam operací společně s časem, kdy se tyto operace vykonávají. Pak u těchto jednotlivých operací můžeme specifikovat využívaný zdrojový prvek pomocí klíčového slova IN. Samotné operace se mohou skládat z dalších pod-operací, ale obecně definují pomocí sekce BEHAVIOR jejich chování a v sekcích ASSEMBLER a CODING navíc syntaxi jazyka instrukční sady a jeho kódování. Detailní popis využívání ostatních zdrojových prvků jako například specifikaci potru registrového pole však náš jazyk zatím neumožňuje.

1.5 Jazyk ISAC a projekt Lissom

Nyní již zpět k našemu projektu. Projekt Lissom byl v roce 2003 zahájen na Fakultě informačních technologií Vysokého učení technického v Brně. V současnosti je hotový překladač jazyka ISAC, interpretovaný a cycle-accurate simulátor, assembler, disassembler, syntetizátor jistých hardwarových částí procesoru a také plugin do vývojového prostředí Eclipse umožňující tyto nástroje využívat. Všechny uvedené nástroje se průběžně vyvíjejí.

Odstraněno: založen
Odstraněno: ,
Odstraněno: máme
Odstraněno: stále

Jazyk ISAC (Instruction Set Architecture C) vychází z jazyka LISA a umožňuje popsat architekturu procesoru a jeho instrukční sadu. Jde o smíšený jazyk pro popis architektury a instrukční sady (viz kap. 1.3). Syntaxe je popsána v dokumentu Jazyk ISAC – Příručka [7]. Současná verze jazyka je 0.1.

Popis je rozdělen na dvě části:

- část zdrojů, zde jsou popsány zdrojové prvky jako jsou paměti, jednotlivé registry, registrové pole a jednotka zřetězeného zpracování (pipeline) procesoru
- část operací, ta popisuje jednotlivé operace, syntax jazyka instrukční sady, kódování instrukcí a jejich chování

Část zdrojů není pro assembler, který je tématem této diplomové práce, příliš důležitá a tak se zaměřím spíše na část operací.

1.5.1 Jazyk ISAC, verze 0.0

V prvotní verzi jazyka ISAC 0.0 byly operace popsány ve formě připomínající překladovou gramatiku a na základě tohoto pohledu také probíhal překlad mezi jazykem strojových instrukcí a binární podobou instrukcí. V článku Two-Way Deterministic Translation and Its Usage in Practice [8] můžete nalézt definici překladové gramatiky a popis algoritmu překladu, který tyto překladové gramatiky využívá.

Příklad instrukce ADD popsané pomocí operací:

```
OPERATION axreg {
      ASSEMBLER { "AX" }
      CODING { 0b01 }
}

OPERATION bxreg {
      ASSEMBLER { "BX" }
      CODING { 0b10 }
}

GROUP reg = axreg, bxreg;

OPERATION Add {
```

```
    INSTANCE reg ALIAS { src, dest }
    ASSEMBLER { "ADD" dest "," src }
    CODING { 0x1001 src dest}
}
```

Nejprve si definujeme překladovou párovou gramatiku: Překladová párová gramatika je 5-tice $G = (N, \Sigma, \Delta, P, S)$, kde N je konečná množina nonterminálů, Σ je konečná vstupní abeceda, Δ je konečná výstupní abeceda, P je konečná množina pravidel v podobě $A \rightarrow x1 \mid x2$, kde $A \in N$, $x1 \in (N \cup \Sigma)^*$, $x2 \in (N \cup \Delta)^*$ takových, že v $x1$ a $x2$ se vyskytuje stejný počet nonterminálů a je startovní nonterminál.

Pak, gramatika získaná na základě předchozího popisu instrukce ADD pak vypadá takto:

G = (*N*, Σ, Δ, *P*, <add>), kde:

- *N* = { <start>, <add>, <addsrc>, <adddest>, <reg>, <axreg>, <bxreg> }
- Σ = { “ADD”, “,”, “AX”, “BX” }
- Δ = { “0”, “1” }
- *P* = { **1:** <add> → “ADD” <adddest> “,” <addsrc> | “0” “1” “0” “1” <addsrc> <adddest>,
  2: <addsrc> → <reg> | <reg>, **3:** <adddest> → <reg> | <reg>,
  4: <reg> → <axreg> | <axreg>, **5:** <reg> → <bxreg> | <bxreg>,
  6: <axreg> → “AX” | “0” “1”, **7:** <bxreg> → “BX” | “1” “0” }

Ve verzi 0.0 jazyk ISAC popisoval především syntaxi a kódování instrukční sady. Pomocí sekce BEHAVIOR bylo sice také možné pro simulátor popsat chování jednotlivých operací, ale nebylo možné modelovat zřetězené linky a popis časového modelu nebyl možný vůbec.

1.5.2 Jazyk ISAC, verze 0.1

Ve současné verzi 0.1 byla zavedena nová sekce operací ACTIVATION (viz [9]). Ta nám umožňuje popsat model časování. Tím se popis operací poněkud vzdálil od popisu toho, jak se mají instrukce překládat a spíše jde o popis chování procesoru pro simulátor (a i pro překladač vyššího programovacího jazyka).

Další novou konstrukcí je klíčové slovo IN. Pomocí něj můžeme operaci určit, který zdrojový prvek využívá. V současné době je možné určit buď část zřetězené linky (pipeline stage) a nebo nějaký, zatím abstraktní, logický prvek jako například aritmeticko-logickou jednotku.

Díky konstrukcím ACTIVATION a IN můžeme tedy popsat model časování a využívání zdrojových prvků. Tím jsme se přiblížili modelu operací popsatelném pomocí trojice "chování, zdroj a čas" (viz předchozí kapitola 1.4).

Novou sekcí operace je dále EXPRESSION (viz [7], str. 26). Ta docela mění celkový koncept jazyka a umožňuje se dívat na definice jednotlivých operací jako na funkce. Ty můžou vykonat nějakou akci vrátit výsledek specifikovaný sekcí EXPRESSION. Tedy, znovu, nejde ani tak o popis čistě instrukční sady, ale spíše o popis toho, jakým způsobem procesor funguje a jak může být simulován.

Konečně posledními novými konstrukcemi, jsou SWITCH-CASE a IF-THEN-ELSE (viz [10], str. 55-58). Ty byly zavedeny kvůli tomu, že je občas potřeba popsat podobné operace, které se liší pouze v jedné nebo více sekcích a také, například v sekci ACTIVATION je za základě výsledku jedné operace vybrat správnou operaci, která se má provést.

V této první kapitole jsem si stručně řekli něco projektu Lissom a o jazycích pro popis architektury. Nyní se již zaměříme přímo na téma této diplomové práce a tím je assembler. V následující kapitole si uvedeme nějaké informace o instrukčních sadách a o tom, co by výsledný assembler měl umět a k čemu přesně bude sloužit.

2 Assemblery a instrukční sady

2.1 Účel assembleru

Assembler vytváří objektový soubor překladem mnemonických zápisů instrukcí do jejich binární podoby, určením hodnot symbolických názvů a zpracováním dalších konstrukcí jazyka assembleru (direktiv). Assemblery poskytují pouze velice malou automatizaci pro programátory. Je jich potřeba pro psaní nízkoúrovňových rutin, které přímo ovládají hardware. Některé z takovýchto operací nejde pomocí vyššího programovacího jazyka popsat, protože využívají speciální instrukce. U některých architektur navíc není překladač vyššího programovacího jazyka k dispozici a tak programátorovi nezbývá nic jiného, než aplikaci pro assembler psát.

Odstraněno: -

Dalším případem, kde se psaní kódu pro assembler používá, jsou různé DSP procesory. Ty často provádějí různé specifické operace (např. uložení zpracovávaných dat do cyklických bufferů a jiných speciálních adresovacích módů) a pomocí univerzálního programovací jazyka jako je například jazyk C není možné tyto schopnosti procesoru dostatečně efektivně využít.

Odstraněno: u

2.2 Obecné assemblery

Obecně se dá to, co assembler překládá rozdělit na dvě části – instrukce a symboly s direktivami. V našem projektu potřebujeme, aby se část, která umožňuje překlad instrukcí, dala jednoduše změnit. Takovéto assemblery se nazývají retargetabilní assemblery, nebo, v českém překladu, assemblery se změnitelnou cílovou architekturou či obecné assemblery.

K assemblerům se brzy vrátíme, nyní si jenom stručně uvedeme některé související pojmy jako jsou instrukční sady a podíváme se také na to, jak fungují moderní překladače vyšších programovacích jazyků.

2.3 Instrukční sady

2.3.1 Různá přirovnání k instrukčním sadám

V literatuře lze nalézt několik různých přirovnání používaných k popisu vztahy mezi překladači, architekturami instrukčních sad a implementacemi (implementací se zde myslí hardware procesoru).

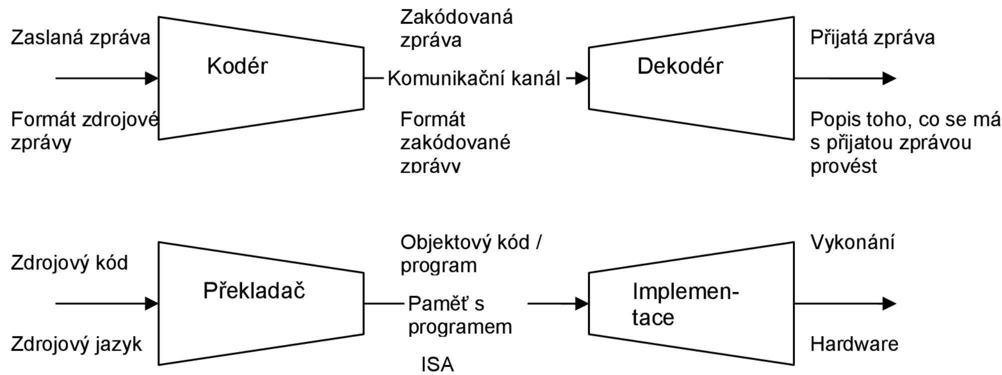
Naformátováno: 1. odstavec

Jedním z nich je *rozhraní*, používané v knize Computer Organization and Design: The Hardware/Software Interface [11]. Dalším může být *dohoda*, kde architektura instrukční sady je brána jako dohoda mezi návrháři softwaru a hardwaru. Třetím přirovnáním je *předpis* či *recept*, kde objektový kód je receptem, podle kterého se vytvoří postup pro vykonávání programu. To, co všem těmto přirovnáním je společné, že každé z nich popisuje nějakou abstraktní vrstvu, kde se prolínají různé obory počítačových věd a musí se nalézt společná řeč pro domluvu.

Naformátováno: 2. odstavec

Odstraněno: .

Dalším přirovnáním může být přirovnání programu ke komunikačnímu kanálu, kde architektura a kódování instrukční sady popisuje *komunikační protokol* mezi překladačem a implementací.



Obrázek 2.1: Analogie mezi architekturou instrukční sady a komunikačním protokolem

Na obrázku 2.1 je znázorněno přirovnání architektury instrukční sady ke komunikačnímu protokolu. Program je zprávou pro hardware procesoru a architektura instrukční sady určuje jeho syntaxi a sémantiku. (Částečně převzato z [6].)

2.3.2 Návrh architektury instrukční sady

Architektura instrukční sady typicky určuje kódování instrukcí, počet a typ registrů viditelných programu a typy operací, které mohou být vykonány jednotlivými funkčními jednotkami procesoru.

Instrukční sada může být navržena s ohledem na více různých kritérií jako jsou kompaktnost, kompletnost a ortogonalita.

V dřívějších dobách se často navrhovaly instrukční sady tak, aby se více blížily vyššímu programovacímu jazyku. Od toho se dnes upouští a převahu získávají RISCové architektury s menším počtem jednoduchých instrukcí.

Zajímavé informace o instrukčních sadách se dají nalézt například v knize Embedded Computing: A VLIW Approach to Architecture, Compilers and Tools [6], kapitole 3, která pojednává především o kódování instrukcí ve spojitosti s architekturami VLIW. Dále pak, ve dvou velice si podobných knihách Computer Organization and Design: The Hardware/Software Interface [11], kap. 2 a Computer Architecture: A Quantitative Approach [12], příloha B, kde se můžete dočíst o klasifikacích instrukčních sad, adresovacích módech a kódování instrukcí. Také zde můžete nalézt výsledky měření ukazující, jaké instrukce, jaké adresní módy a jaké velikost operandů jsou nejvíce využívané u různých testovacích programů a na základě těchto měření jsou uvedeny rady, kterými by se návrháři instrukčních sad měli řídit.

3 Assemblery

V předchozí kapitole jsme si řekli, jak probíhal dosavadní vývoj assembleru. Assembler je část projektu, která by mohla být stálá a příliš se neměnit. Po důkladném návrhu. se už asi těžko objeví něco, s čím jsme nepočítali a z toho důvodu si myslím, že už je na čase s assemblerem postoupit poněkud dále a vytvořit assembler, který bude umět vše, co od něj čekáme.

3.1 Literatura o obecných assemblerech

Oblast výzkumu týkající se retargetabilního překladu postoupila do bodu, kdy jazyk pro popis architektury (ADL) lze použít k modelování mikroarchitetury a překladač může být automaticky generován z takovéto specifikace architektury. Zatímco výzkum týkající se generování překladače se dostává do poměrně vyspělého stádia, pouze málo snahy bylo zatím věnováno automatickému generování ostatních nástrojů jako je assembler či linker. To je z části z důvodu, že jsou tyto úkoly vnímány jako triviální v porovnání s optimalizujícím překladačem. To však nemění nic na tom, že assemblery, ač se v nich nepoužívají žádné sofistikované algoritmy, jsou poměrně komplexními aplikacemi a musí řešit určité specifické úkoly. (částečně podle článku [13])

Asi jedinou knihou pojednávající o architektuře assembleru je, dnes již v některých ohledech poněkud zastaralá, kniha Asssemblers and Loaders [14]. Dále existuje nepřeberné množství knih a příruček o konkrétních assemblerech pro danou cílovou architekturu, ty však bez výjimky popisují, jak pro daný assembler psát kód a ne to, jak je kód překládán.

Assemblery úzce souvisejí s objektovými soubory a linkery. Kniha Linkers and Loaders [15] popisuje a vysvětluje vybrané formáty objektových souborů a funkci linkeru, obsahuje užitečné a aktuální informace.

Dále se dá nalézt několik článků o obecných assemblerech, prvním může být A Generative Approach to Universal Cross Assembler Design [16]. Ten stručně popisuje generování assembleru s využitím nástrojů Yacc a Lex, avšak s tím, že upravitelné části je zapotřebí napsat ručně. Další publikace ([13], [17], [18] a [19]) pojednávají o tom, jak automaticky upravovat GNU assembler z balíčku nástrojů GNU binutils. Zájem autorů posledních článků o napovídá, že právě GNU assembler by mohl být dobrým zdrojem inspirace. K tomuto tématu se vrátíme v následující kapitole 4.

3.2 Jedno a dvouprůchodové assemblery

V této kapitole si ve zkratce řekneme, jak obvykle assemblery fungují. Popis se bude týkat jenom překladu instrukcí a zpracování symbolů. Zpracování direktiv určujících podobu výsledného objektového souboru prozatím vynecháme.

Naformátováno: 2. odstavec

Odstraněno: -

Odstraněno: -

Naformátováno: 1. odstavec

3.2.1 Jednoprůchodový assembler

Jak jeho název naznačuje, čte zdrojový soubor pouze jednou. Během tohoto jediného průchodu zpracuje zároveň definice návěští (symboly) a přeloží instrukce. Problém má však se symboly, které jsou definované později, než byly použity (tzv. budoucí symboly). Tento problém je možné řešit tak, že si zaznamená místa jejich výskytů a ve chvíli, kdy jsou definovány, upraví označená místa tím, že dosadí zjištěné hodnoty symbolů (provede tzv. *fix-upy* či *bitové opravy*), to však znamená další průchod.

Naformátováno: 1. odstavec

Pozn. 1: V knize [14], na straně 26, je uvedeno, že jednoprůchodový assembler může generovat pouze absolutní objektové soubory a už ne soubory relokatibilní. Toto tvrzení zde však není nijak podloženo a osobně nevidím důvod, proč by nemohl vytvářet i soubory relokatibilní.

Pozn. 2: Jednou z nevýhod jednoprůchodového assembleru je, že tzv. listing výpisy, pokud se generují současně během prvního průchodu, nejsou kompletní (nejsou vypsány zakódované hodnoty budoucích symbolů).

3.2.2 Dvouprůchodový assembler

Dvouprůchodový assembler obvykle funguje tak, že v prvním průchodu hledá pouze definice symbolů, ukládá je do tabulky symbolů včetně jejich pozice a instrukce nepřekládá. U instrukcí pouze určuje jejich délku a díky tomu může zjistit hodnoty symbolů (adresy, které tyto symboly definované jako návěští, představují). Na konci prvního průchodu by tedy tabulka symbolů měla obsahovat definice všech návěští definovaných v programu.

Ve druhém průchodu pak překládá instrukce a používá hodnoty symbolů obsažených v tabulce symbolů.

Za dvouprůchodový assembler může být považován i assembler, který čte vstupní soubor pouze jednou a zaznamená si informace o místech, kde se má provést bitová oprava. V druhém průchodu pak tyto bitové opravy vykoná a zapíše výsledný objektový soubor.

3.3 Pojmy sekce, location counter a relokace

3.3.1 Sekce

Pomocí sekcí je možné rozdělit jednotlivé části kódu a dat. Takovéto rozdělení do sekcí je obzvláště důležité pro architektury s oddělenými adresovými prostory pro kód a data (harvardská architektura) a pro architektury se sofistikovanější správou či ochranou paměti.

Naformátováno: 1. odstavec

Obecně existují tři druhy sekcí: kódové, s inicializovanými daty a s neinicializovanými daty. Také se používají specializované sekce, například kvůli potřebám objektově orientovaných programovacích jazyků. Další speciální sekce se využívají pro podporu statických či dynamických sdílených knihoven (detaily viz [15], str. 50-56 a 187-246).

Naformátováno: 2. odstavec

V našem assembleru si však v současnosti vystačíme se třemi uvedenými základními typy: kódové, s inicializovanými a s neinicializovanými daty. Sekci si můžeme představit jako jednoduché

datové pole na jehož konec se při překladu konec přidávají nová data vzniklá přeložením instrukcí či definic dat. Výsledný objektový soubor je pak složen z takto vytvořených sekcí.

Když assembler při překladu narazí na direktivu představující definici nové sekce, přidá aktuální sekci k seznamu sekcí, vytvoří novou sekci a označí ji jako aktuální. Do ní se pak budou přidávat další přeložené instrukce a data.

Naformátováno: Odsazení: První řádek: 0,7 cm

3.3.2 Ukazatel aktuální pozice – location counter (LC)

Důležitou součástí každého assembleru je takzvaný location counter (LC). Ve své podstatě jde pouze o nějakou celočíselnou proměnnou, která určuje aktuální adresu ve zpracovávané (také aktuální) sekci.

Aktuální hodnota LC je využívána ve chvíli, kdy je definováno návěští. Symbolu, který představuje toto návěští, je přiřazena hodnota LC a tím pádem je hodnotou symbolu jeho vzdálenost od začátku sekce.

3.3.3 Symboly

Symboly assembleru mohou ve zdrojovém souboru definovány obecně dvěma způsoby: jako návěští a jako konstanty. Podle toho se určuje jejich typ hodnoty: hodnota návěští je vždy relativní k začátku sekce a hodnota konstantního symbolu je hodnotou absolutní. Dále je také potřeba popsat nedefinované symboly (symboly, na které se ze zdrojového souboru pro assembler odkazujeme, ale jsou definovány v jiném objektovém souboru).

Naformátováno: 1. odstavec

Dále ke každému symbolu potřebujeme uložit jeho název, hodnotu a, u relokatibilních symbolů, informaci o tom, ve které sekci byl definován. Také je občas vhodné vědět, jakého datového typu je jeho hodnota. V naprosté většině případů jde o celočíselný datový typ, a ukládá se pouze informace, zda je typ znaménkový či neznaménkový.

Naformátováno: 2. odstavec

3.3.4 Relokace

Pojem relokace se týká většinou linkerů. Avšak i v assemblerech hraje svou důležitou roli, zde se často označuje jako *fix-up* či v češtině *bitová oprava*.

Naformátováno: 1. odstavec

Poté, co si linker projde všechny objektové soubory, které má spojit, pro všechny sekce určí jejich finální umístění v paměti a vypočítá hodnoty všech symbolů, může provést relokaci. Informace o relokaci jsou v objektových souborech uloženy v podobě relokačních položek.

Naformátováno: 2. odstavec

Relokační položka (relocation entry) je tedy strukturou objektového souboru, obsahuje informace pro linker či loader a slouží ke dvou účelům:

Naformátováno: 2. odstavec

1. Když je sekce kódu přesunuta na jinou bázovou adresu, relokační položky určují místa v kódových či datových sekcích, které musí být upraveny - *lokální relokace*.

2. V linkovatelném souboru jsou také relokační položky, které představují odkazy na nedefinované symboly. Linker, když už zná hodnoty nedefinovaných symbolů, ví, která místa v sekcích má patřičně upravit - *externí relokace*.

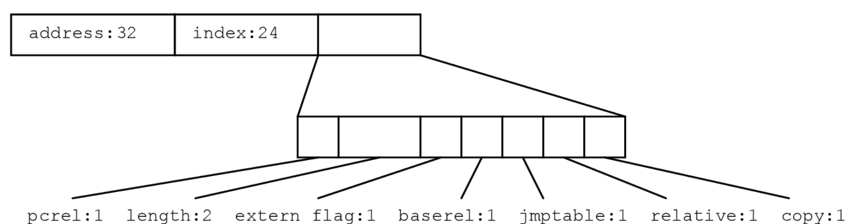
Názvy lokální a externí relokace nejsou součástí obvyklé terminologie, přesto si myslím, že dobře popisují dva možné druhy relokace a v dalším popisu je využívám. Další pojmem je *relokovaná hodnota*, ta představuje je výslednou hodnotu upraveného (relokovaného) místa.
Relokační položka je vždy svázána s nějakou sekcí.

Nyní si popíšeme, jak taková relokační položka vypadá a jaké informace nese. Každá položka obsahuje *adresu* v sekci společně s další informací určující, co se má při relokaci dělat (podle [15]). Tato další informace se skládá dále z *typu relokace*, ten určuje metodu výpočtu relokované hodnoty. Dále z *relokačního operandu* (relocation addend), jež je proměnnou která slouží k uložení informace pro výpočet v případě, že místo v kódu či datech, které má být relokováno není dostatečně veliké pro uložení potřebných informací. Zatímco přesný zdroj této informace se může lišit, ve finále je to většinou číselná hodnota, která je přičtena k relokované hodnotě a z toho vychází její anglický název. V případě češtiny mi však název operand připadá výstižnější. (podle [17], zde také můžete nalézt příklady různých typů relokace pro architektury SPARC a Intel 386).

Podíváme se na několik příkladů relokačních položek, jak jsou definovány v různých formátech objektových souborů. Budeme zde hledat *relokační adresu*, *typ relokace* a *relokační operand*.

Relokační položka formátu a.out

V UNIXOvém formátu a.out má relokační položka následující formát, číslo za dvojtečkou určuje délku v bitech:



Obrázek 3.1: Relokační položka formátu a.out

Položky mají takovýto význam:

- `address` – pozice místa v datové či kódové sekci, které se má upravit, jeho hodnota je počtem bajtů od začátku sekce,
- `index` – v závislosti na extern flag je buď určuje index sekce, které sekce se tato položka týká (relokace je *lokální*, viz první účel relokační položky) a nebo je indexem symbolu pro tuto relokační položku v tabulce symbolů (*externí relokace*),
- `pcrel` – určuje, zda je výsledná relokovaná hodnota brána jako relativní adresa,
- `length` – délka relokované hodnoty, hodnoty 0 – 3 představují délku 1, 2, 4 a 8 bajtů,

- `extern_flag` – určuje interpretaci položky index, pokud je roven 0, tak jde o lokální relokaci, pokud je 1, tak jde o externí relokaci,
- `copy` – pokud je příznak nastaven, říká, že pro relokační hodnotu se má použít přímo hodnota symbolu (zároveň extern flag nesmí být 0),
- `baserel`, `jmptable` a `relative` - souvisí se sdílenými a dynamickými knihovnami, nejsou pro nás důležité, protože prozatím nepředpokládáme, že vygenerovaný kód poběží nad nějakým operačním systémem, který by disponoval těmito vymoženostmi.

V případě tohoto formátu je `address` *relokační adresou*, `index` *relokačním operandem* a ostatní položky určují *typ relokace*.

Formát a.out umožňuje definovat pouze relokaci, jejíž umístění je zarovnáno na úroveň bajtů a jejíž velikost je pouze 1, 2, 4 nebo 8 bitů. Toto je například pro některé 8-bitové mikrokontrolery nepřijatelné omezení. Typ relokace umožňuje pouze velice malou možnost manipulace s relokovanou hodnotou, pro obvyklé architektury je přinejmenším kromě součtu ještě potřeba operací jako je posuv, logický součin a součet. Podrobnější popis formátu můžete nalézt například v [20].

Relokační položka formátu COFF

Virtual Address:32	Symbol Table Index:32	Type:16

Obrázek 3.2: Relokační položka formátu COFF

Prvky mají následující význam:

- `Virtual Address` – Adresa místa, u kterého se bude provádět relokace. Jde o offset od začátku sekce plus hodnota položky sekce RVA/Offset field (adresa začátku této sekce v paměti, zkratka RVA znamená „relative virtual address“). Narozdíl od a.out jde o adresu místa v paměti a ne pouze o vzdálenost od začátku sekce.
- `Symbol Table Index` – Index v tabulce symbolů.
- `Type` - Hodnota určující jaký typ relokace má být proveden, určuje také velikost relokované hodnoty. Je definováno velké množství typů relokace, zvlášť pro každou podporovanou architekturu, např. pro PowerPC je jich 18, pro Intel 386 11, pro Intel Itanium 30.

Rozlišení externí a lokální relokace je určeno typem odkazovaného symbolu v tabulce symbolů. V případě tohoto formátu je určení jednotlivých součástí relokační položky přímočaré: `Virtual Address je` *relokační adresou*, `Symbol Table Index` *relokačním operandem* a `Type` určuje *typ relokace*.

Pomocí typu relokace je možné definovat libovolnou velikost a pozici výsledné relokované hodnoty, nemusí být zarovnána na úroveň bajtů. Podpora daných typů však musí být implementována ručně.

V následující tabulce můžete vidět příklady některých typů relokace pro architekturu MIPS formátu COFF:

Konstanta	Hodnota	Popis
IMAGE_REL_MIPS_ABSOLUTE	0x0000	Relokace je ignorována.
IMAGE_REL_MIPS_REFHALF	0x0001	Horních 16 bitů výsledné virtuální adresy (VA).
IMAGE_REL_MIPS_REFWORD	0x0002	Výsledná 32-bitová VA.
IMAGE_REL_MIPS_JMPADDR	0x0003	Dolních 26 bitů výsledné VA. Zavedeno z důvodu podpory MIPS instrukcí J a JAL.
IMAGE_REL_MIPS_REFHI	0x0004	Horních 16 bitů cílové 32bitové VA. Využíváno pro první instrukci v sekvenci dvou instrukcí sloužících k nahrání celé adresy. Tato relokace musí být následována dvojicí relokací, jejichž SymbolTableIndex obsahuje 16bitový displacement, který je přičten k horním 16 bitům, ty jsou získány z pozice, která je relokována.
IMAGE_REL_MIPS_REFLO	0x0005	Spodních 16 bitů výsledné VA.
IMAGE_REL_MIPS_GPREL	0x0006	16bitový znaménkový displacement výsledné adresy relativní k registru GP.
IMAGE_REL_MIPS_LITERAL	0x0007	The same as IMAGE_REL_MIPS_GPREL.
IMAGE_REL_MIPS_SECTION	0x000A	16bitový index sekce, která obsahuje cíl adresy. Používáno pro podporu ladicích informací.

Tabulka 3.1: Typy relokace formátu COFF pro architekturu MIPS

Podrobný popis formátů PE a COFF se dá nalézt v dokumentu [21].

Naformátováno: 2. odstavec

Příklad dalšího způsobu reprezentace relokační položky, která byla definována definovaná pro potřeby automatického generování knihovny BFD (Binary File Descriptor library, viz [22]) je popsán v článku [17].

Naformátováno: 2. odstavec

Relokační položka formátu objektového souboru používaném v projektu Lissom

Adresa
Typ
Počet slov tvořící adresu
Nejnižší bit adresy
Nejvyšší bit adresy
Index symbolu

Obrázek 3.3: Relokační položka formátu objektového souboru používaného v projektu Lissom

- `Adresa` – adresa relokovaného údaje v datech sekce,
- `Typ` – typ relokace, může být buď A – absolutní, R – relativní nebo F – plochý adresovací model,
- `Počet slov tvořící adresu` – určuje délku relokované hodnoty,
- `Nejnižší a nejvyšší bit adresy` – bitová pozice prvního a posledního bitu relokované hodnoty,

- `Index symbolu` – index v tabulce symbolů.

V našem případě je *relokační adresou* adresa, *relokačním operandem* index a zbylé prvky určují *typ relokace*. Popis formátu tohoto objektového souboru naleznete v dokumentu [23].

Naformátováno: 2. odstavec

Jak je vidět na uvedených příkladech, relokační položka je vždy trojice sestávající z *relokační adresy*, *relokačního operandu* a *typu relokace*. Typ relokace je obecně nějaká funkce, která bere 2 celočíselné parametry a jejím výsledkem je celé číslo. Pokud použijeme operátor "*" ve významu dereference ukazatele (podobně jako v jazyce C), tak předpis pro provedení relokace můžeme zapsat jako:

**relokační_adresa = typ_relokace(*relokační adresa, relokační operand)*

Funkce *typ_relokace*, kromě samotné *operace* určuje navíc bitový offset, bitová velikost a datový typ prvního vstupního parametru a stejné vlastnosti pro výsledek. (Datový typ zde může být pouze znaménkové či neznaménkové celé číslo.)

Naformátováno: 2. odstavec

U výše uvedených formátů objektových souborů je operací typu relokace vždy pouze součet, bitový posuv nebo jejich kombinace.

Pozn. 1: V dokumentu [24] je uvedena možnost, kde namísto jednoho typu relokace se složitější operací (popsaného pomocí jediné relokační položky) lze použít sekvenci typů relokací s primitivními operacemi (více relokačních položek se stejnou relokační adresou). Nevím, jestli kdy bude potřeba používat složité operace při relokaci, pokud však ano, takovýto přístup by mohl být vhodný a zjednodušit linker.

Pozn. 2: V literatuře se pro linker budu používá pojem *relokace* a pro assembler pojem *bitová oprava* (či *fix-up*). Mají téměř stejný význam, pouze s jedním rozdílem. Relokace může být lokální či externí (viz začátek této podkapitoly). Externí relokace odpovídá dosazení hodnoty symbolu, jehož hodnota už je známá a lokální souvisí pouze s posouváním celé sekce v paměti. Lokální relokace nemá u assembleru význam, bitová oprava v assemblerech odpovídá externí relokaci v linkerech.

4 Příklady obecných assemblerů a GNU assembler

4.1 Některé obecné assemblery

Obecných či retargetabilních assemblerů, o nichž by byla dostupná nějaké dokumentace či jejich zdrojové soubory neexistuje příliš mnoho. Objevil jsem tyto:

- CROSS 32 Meta Assembler (http://www.datasynceng.com/c32doc.htm)
- Macro Assembler AS (http://john.ccac.rwth-aachen.de:8000/as/as_EN.html)
- Hobby Cross Assembler (http://home.earthlink.net/~hxa/)

Tyto uvedené assemblery jsou ale víceméně prototypy a nenašel jsem žádné informace o tom, že by byly více využívány nebo, že existuje překladač vyššího programovacího jazyka, který by generoval zdrojové soubory pro tyto assemblery. Ani jejich dokumentace neobsahuje příliš užitečné informace.

Zbývá ještě jeden assembler, který v seznamu uveden není a to je assembler as z balíčku GNU binutils.

4.2 GNU as

GNU assembler (či jenom as) je spíše kolekcí assemblerů než assemblerem jediným. Umožňuje změnit jak instrukční sadu procesoru, pro který se překládá, tak cílový objektový formát.

V současnosti obsahuje balíček binutils verze GNU assembleru pro 54 různých architektur a 8 objektových formátů. Dále byl také přenesen i na různé další další architektury, například na DSP procesory firmy Texas Instruments, nebo na procesor Atmel AVR32, avšak tyto verze nejsou standardní součástí binutils. Něco málo o tom, jak se dá u as změnit cílová architektura naleznete v kapitole 5.6.

Architektura GNU as se postupně vyvíjela tak, aby byla schopná zachytit různé speciality rozličných procesorů a objektových formátů a proto myslím, že by zrovna tento assembler mohl být vhodnou inspirací.

Nevýhodou je však jeho rozsáhlost. Jenom samotné zdrojové kódy mají 7 MB, jsou však poměrně dobře okomentované. Stručný popis jeho architektury se dá nalézt v dokumentu [25]. V několika následujících kapitolách si popíšeme jeho funkci, jaké jsou jeho hlavní součásti a také to, jakým způsobem se dá změnit cílová architektura. Po dokončení popisu se již pustíme do samotného návrhu našeho assembleru.

Pozn.: GNU as je především využíván jako překladač výstupu překladačů vyšších programovacích jazyků z kolekce GCC. Asi jedinou knihou, která pojednává o psaní kódu přímo pro GNU as je tato: Programming from the Ground Up [26].

4.3 Kroky překladu

Překlad pomocí as probíhá v těchto krocích:

1. Assembler se inicializuje voláním několika inicializačních rutin.
2. Otevře vstupní soubor a začne ho překládat.
3. Pro každou řádku, assembler předá návěští funkci `colon` a izoluje první slovo. Pokud vypadá jako pseudooperace, slovo je vyhledáno v tabulce pseudooperací a odesláno rutině, která pseudooperaci zpracuje. Jinak, „cílově architektonicky specializovaná" (target dependent) funkce `md_assemble` je zavolána a ta přeloží instrukci.
4. Pokud jsou výsledkem zpracování pseudo-operace či instrukce data, jsou přidána k současné subsekci.
5. Výsledkem pseudooperací a instrukce mohou být také bitové opravy.
6. Pro jisté cílové architektury, instrukce mohou vytvořit tzv. „variant" subsekce, které jsou použity k uložení informace o relaxaci (viz kap. 4.7).
7. Po dokončení překladu vstupního souboru se zavolá funkce `write_object_file`. Ta přiřadí adresy všem subsekcím (`relax_segment`), vyřeší hodnoty všech symbolů (použitím `resolve_symbol_value`), vyřeší bitové opravy (`fixup_segment`), a nakonec zapíše výsledný objektový soubor.

Pozn. 1: Pro direktivy je v GNU as používán ekvivalentní název pseudooperace.
Pozn. 2: Jedna sekce se skládá z několika subsekcí, viz následující kapitola 4.4.

4.4 Subsekce

Subsekce byly do GNU as zavedeny z důvodu, aby bylo možné ve zdrojovém souboru zapisovat promíchaně kód a data, které patří do jediné sekce. Například překladač GCC tohoto využívá tak, že pro každou funkci definuje vlastní kódovou subsekci a pokud je potřeba i subsekci datovou a nemusí se zabývat rozdělováním kódu a dat tak, aby tvořily souvislé sekce. Uvnitř každé sekce může být až 8192 subsekcí.

Číslo subsekce za zapisuje jako volitelný parametr subsegment u direktiv .**section**, .**text** a .**data**. Pokud tento parametr není zadán, jeho implicitní hodnota je 0. Je možné mít více subsekcí se stejným pořadovým číslem.

Ve výstupním objektovém souboru se pak ze všech těchto subsekcí vytvoří jediná sekce, kde subsekce budou seřazeny nejprve podle čísla subsekce a ty, jejichž pořadové číslo je stejné, jsou řazeny podle jejich původního pořadí ve zdrojovém souboru. Další popis subsekcí naleznete v manuálu GNU as [27].

Ukážeme si chování subsekcí na příkladě. Mějme takovýto zdrojový soubor:

```
.text 0
.ascii "1 This lives in the first text subsection. *"
.text 1
.ascii "2 But this lives in the second text subsection."
.section .my_section, "d"
.ascii "3 This lives in the my_section data section,"
.ascii "in the first data subsection."
```

```
.text 0
.ascii "4 This lives in the first text section,"
.ascii "immediately following the asterisk (*)."
.section .my_section, 1
.ascii "5 And this follows first my_section subsection."
```

Výsledný objektový soubor bude obsahovat dvě sekce s daty - .text a .my_section (výpis byl získán pomocí příkazu objdump –s.):

```
Contents of section .text:
 0000 31205468 6973206c 69766573 20696e20  1 This lives in
 0010 74686520 66697273 74207465 78742073  the first text s
 0020 75627365 6374696f 6e2e202a 34205468  ubsection. *4 Th
 0030 6973206c 69766573 20696e20 74686520  is lives in the
 0040 66697273 74207465 78742073 65637469  first text secti
 0050 6f6e2c69 6d6d6564 69617465 6c792066  on,immediately f
 0060 6f6c6c6f 77696e67 20746865 20617374  ollowing the ast
 0070 65726973 6b20282a 292e3220 42757420  erisk (*).2 But
 0080 74686973 206c6976 65732069 6e207468  this lives in th
 0090 65207365 636f6e64 20746578 74207375  e second text su
 00a0 62736563 74696f6e 2e909090 90909090  bsection........
Contents of section .my_section:
 0000 33205468 6973206c 69766573 20696e20  3 This lives in
 0010 74686520 6d795f73 65637469 6f6e2064  the my_section d
 0020 61746120 73656374 696f6e2c 696e2074  ata section,in t
 0030 68652066 69727374 20646174 61207375  he first data su
 0040 62736563 74696f6e 2e352041 6e642074  bsection.5 And t
 0050 68697320 666f6c6c 6f777320 66697273  his follows firs
 0060 74206d79 5f736563 74696f6e 20737562  t my_section sub
 0070 73656374 696f6e2e                    section.
```

GAS tedy seřadí subsekce podle jejich pořadových čísel. Struktura GAS představující 1 subsekci se jmenuje `frag` Sekce se také občas označují jako segmenty a 1 segment představuje struktura `segment_info_struct(`detaily viz příloha F).

Uvažoval jsem nad tím, zda bude vhodné podporu subsekcí do našeho assembleru zavést (první návrh s tímto dokonce počítal). Po dalším zvážení jsem se rozhodnul podporu nezavádět, protože by zbytečně assembler zkomplikovala. Požadovaná funkcionalita by se dala dosáhnout pomocí preprocesoru, který by subsekce předem seřadil do výsledných sekcí.

4.5 Struktury GNU as

Asi nejdůležitějšími součástmi GNU as jsou struktury, do kterých si ukládá informace získané během překladu.

Jde o tyto:

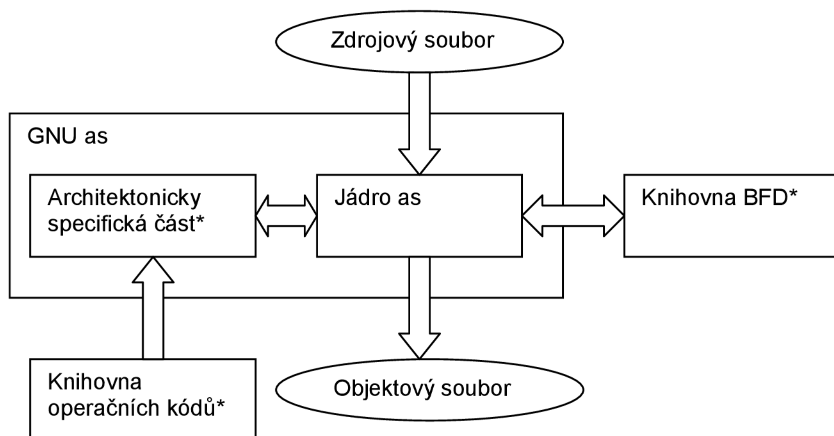
- symbol – obsahuje informace o jednom symbolu, jeho hodnotu představuje instance struktury expressionS
- expressionS – výraz, může se skládat z dalších podvýrazů
- fix – informace o bitové opravě
- frag – jedna subsekce

Jsou pevnými součástmi assembleru, bez ohledu na to, pro jakou architekturu a do jakého objektového formátu assembler překládá.

Tyto struktury a jejich prvky jsou podrobněji popsány v dokumentu [25] a jejich diagram tříd naleznete v příloze F. Inspiroval jsem se jimi při návrhu struktur a architektury assembleru projektu Lissom.

4.6 Změna cílové platformy

Jak již bylo uvedeno v kapitole 4.2, existuje mnoho rozličných verzí GNU as pro různé architektury. Na následujícím obrázku můžete vidět součásti assembleru (označeny hvězdičkou "*"), které je nutné dodat či modifikovat, abychom docílili změny cílové architektury a formátu objektového souboru. (Obrázek byl převzat z [18].)



Obrázek 4.1: Součásti GNU as

Jádro as je kódem, který je nezávislý na cílové architektuře a provádí překlad návěští, direktiv a výrazů. Dále volá funkce architektonicky specifických modulů, ty mají přesně určené rozhraní.

Knihovna operačních kódů obsahuje informace o cílové instrukční sadě. Není přesně určeno, jaký mají mít tyto informace formát, proto se musí vytvářet architektonicky specifická část GNU as, která informace v knihovně operačních kódů využívá a překládá instrukce.

Knihovna BFD (Binary File Descriptor library) poskytuje funkce pro generování výsledného objektového formátu. Je využívána jak assemblerem, tak linkerem a i jinými součástmi balíčku binutils. Její podrobnější popis naleznete v [22], dále článek [17] popisuje, jakým způsobem se dá tato knihovna automaticky generovat na základě popisu binárního aplikačního rozhraní (ABI, Application Binary Interface).

4.7 Relaxace

Relaxace termínem používaným v GNU assembleru a GNU linkeru pro označení případu, kdy velikost či kódování nějaké instrukce závisí na hodnotě symbolu nebo jiných dat. Něco málo o tom, jak je relaxace řešena v GNU as naleznete v [25].

Příkladem, kde je relaxace zapotřebí jsou architektury s tzv. banky paměti. Jejich adresový prostor se může jevit jako souvislý, ale adresy zakódované jako parametry instrukcí mají nižší rozsah a není možné s jejich pomocí adresovat celý prostor. V těchto procesorech existují speciální registry, které vybírají aktuální banky paměti a tímto umožňují vybrat ve kterém banku se adresuje. Příkladem takovéto architektury jsou procesory Microchip řady PIC16. Ty mají až 8KB paměti, ale prostor pro adresu v zakódované instrukci CALL má pouze 11 bitů – tj. může adresovat 2KB. Pro výběr banku jsou ve speciálním registru STATUS vyhrazeny dva bity. Výsledná adresa se pak vytvoří tak, že se tyto dva bity přidají před adresu z instrukce CALL – adresa má pak 13 bitů a takto můžeme adresovat celý adresní prostor.

Problém spočívá v tom, že překladač neví, jestli je potřeba změnit aktuální banku paměti, protože nezná adresu, která je parametrem instrukce CALL. Případně by mohl vygenerovat kód, který vždy správný bank paměti vypočítal, ale vygenerovaný kód by se skládal z několika instrukcí a výsledný program by byl velice neefektivní.

Zde nastupuje relaxace. Linker (a občas i assembler) již znají konkrétní adresu, číslo banku mohou vypočítat sami a v případě potřeby před instrukci CALL přidají jedinou instrukci pro výběr banku.

V dalších případech se relaxace nepoužívá ani tak pro správnou funkci kódu, ale pouze pro optimalizaci. Příkladem mohou být přímý a rozšířený adresní mód mikrokontroleru Motorola HC08. Operační kód instrukce a její délka záleží na tom, zda je parametrem 8-bitové číslo (přímý mód) nebo 16-bitové číslo (rozšířený mód). Zde je možné používat vždy rozšířený mód a výsledný kód bude korektní. Zde jde tedy pouze o optimalizaci velikosti (a částečně i rychlosti) kódu.

Naformátováno: 2. odstavec

Jde o poměrně složitý problém a to především z důvodu, že assembler generujeme automaticky. Myslím, že ho budeme muset v budoucnu nějakým způsobem řešit. V současnosti však tento problém není nijak akutní.

Naformátováno: 2. odstavec

V této kapitole jsem se pokusil vypsat některé důležité informace o GNU assembleru. Tyto poznatky jsem pak využil při návrhu assembleru projektu Lissom.

Naformátováno: 2. odstavec

5 Shrnutí předchozího vývoje, dvojcestné párové automaty

V této kapitole si shrneme předchozí vývoj assembleru a řekneme si něco o překladu pomocí dvojcestných párových automatů.

5.1 Verze 0.1

Verzí 0.1 budu označovat generátor assembleru, který byl vytvořen v rámci mého ročníkového projektu v ak. roce 2004/2005. Touto prací byl, mimo jiné, naznačen způsob, jakým by se assembler mohl generovat.

Následuje stručné shrnutí toho, jakým způsobem generování assembleru a překlad instrukcí funguje. Pro pochopení detailů je potřeba prostudovat odkazované dokumenty.

V principu jde o to, že se na popis instrukční sady procesoru popsané pomocí jazyka ISAC díváme jako na párovou atributovou LR gramatiku (viz kapitola 1.5.1). Tím, že jde o LR gramatiku, se nabízí využití nástroje Bison (Yacc) pro generování překladače jazyka instrukcí. Dále je také nutné nějakým způsobem popsat gramatiku jazyka assembleru. Pro generování kódu překladače jsme také použili nástroj Bison.

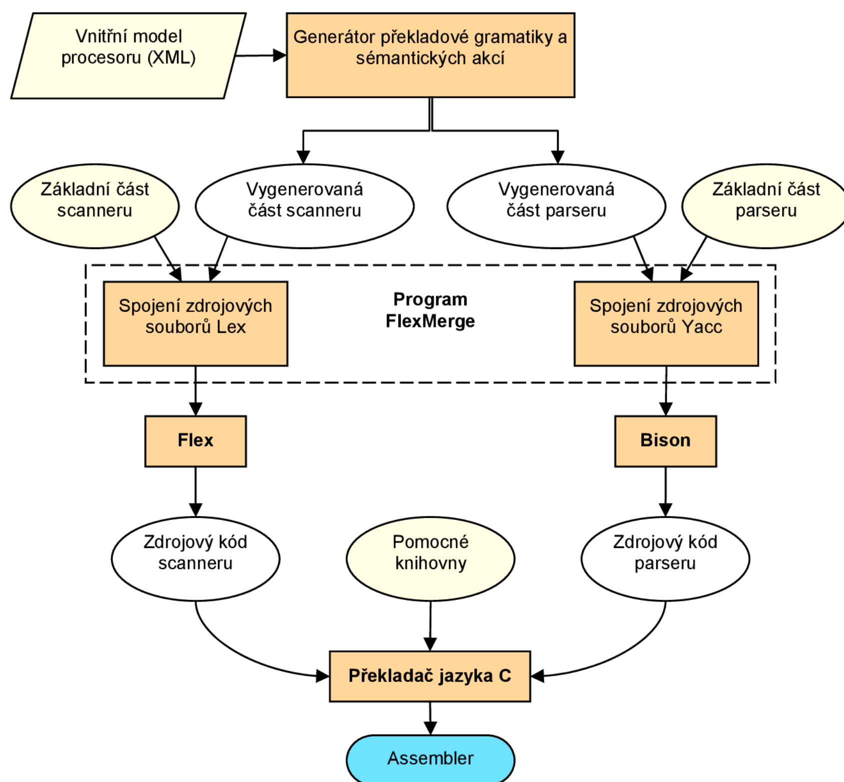
Jako vstup generátoru slouží vnitřní model jazyka ISAC (viz [28] a [29]). Z tohoto modelu se kromě části scanneru vygeneruje také část parseru (jde o zdrojový kód pro Bison) a představuje párovou atributovou gramatiku popisující IS a přepisovací pravidla s odpovídajícími sémantickými akcemi pro překlad do binární podoby.

V článku [8] je popsán algoritmus pro obousměrný deterministický překlad mezi jazykem instrukcí a jeho binární podobou (funguje i pro opačný směr překladu). Postup překladu byl pojmenován „Pletací algoritmus“. V dokumentaci k ročníkovému projektu [30] byl pak tento algoritmus doplněn tak, aby popisoval i tok hodnot atributů. Tento algoritmus je využíván vygenerovaným assemblerem.

Protože jsme se snažili o to, aby se disassembler co nejvíce podobal assembleru (viz např. párové gramatiky), probíhal vývoj assembleru a disassembleru společně.

Následující diagram ukazuje, jakým způsobem se assembler generuje (převzato z [31]).

Naformátováno: 2. odstavec



Obrázek 5.1 Generování assembleru na základě informací z vnitřního modelu procesoru.

Assembler verze 0.1 podporuje:

- jednoduchý překlad instrukcí do binární podoby.
- výstupem je jednoduchý proud nul a jedniček (v textové podobě).

Nepodporuje:

- symboly.
- výrazy.
- návěští.
- generování objektových souborů.

5.2 Verze 0.2

V této podkapitole bude, jakým způsobem byl generovaný assembler rozšířen do verze 0.2. Tímto vylepšením se zabýval Libor Vašíček v rámci jeho bakalářské práce [31].

Naformátováno: 1. odstavec

Zásadním nedostatkem assembleru verze 0.1 bylo, že si neumí poradit se symboly a návěštími a také to, že neumí generovat výsledek překladu v do podoby objektového souboru pro linker.

Do generovaného assembleru byla tedy doplněna podpora návěští. Také byla vytvořena knihovna *objfilelib* poskytující funkce pro vytváření a čtení objektových souborů podle formátu popsaného v [23]. Zájemce o další detaily odkazuji na dokument [31].

Co tedy assembler verze 0.2 umí:

- překlad instrukcí do binární podoby,
- symboly a návěští,
- generování objektových souborů.

A neumí:

- výrazy,
- direktivy pro zarovnávání (aligning).

V následující podkapitole 5.3 se mimo jiné podíváme na jeden z větších nedostatků assembleru verze 0.2 a to podporu výrazů.

Naformátováno: Odsazení: První řádek: 0,5 cm

5.3 Propojení generovaného jazyka instrukcí a jazyka assembleru ve verzích 0.1 a 0.2

5.3.1 Jazyk instrukční sady (JI)

Jde o jazyk popisující syntaxi zápisu jednotlivých instrukcí instrukční sady. Definuje ho uživatel pomocí jazyka ISAC s využitím konstrukcí GROUP a OPERATION.

U jazyka instrukční sady (nebo jenom jazyka instrukcí) zavedeme pojem s ním související a to překladač jazyka instrukcí. Ten bude označován buď jako překladač instrukcí a nebo kodér instrukcí.

5.3.2 Jazyk assembleru (JA)

Jazyk assembleru je jazykem popisujícím „vyšší" vrstvu celého jazyka assembleru. S jeho pomocí je definována struktura vstupního souboru assembleru. Ve verzích 0.1 a 0.2 popisuje zápis direktiv, symbolů, konstantních čísel a sémantickými akcemi jeho přepisovacích pravidel je kromě mimo jiné také popsáno, jak se generuje výstupní objektový soubor.

5.3.3 Propojení jazyků JA a JI

Oba dva jazyky JA i JI jsou popsány bezkontextovými gramatikami. Jazyk instrukcí má jeden startovní nonterminál, pojmenujme ho *isac_instr*. Startovní nonterminál jazyka assembleru nechť se jmenuje *start*. V následující části kódu (ze souboru asmyaccbase.yy) je ukázáno, jakým způsobem jsou tyto dva jazyky propojeny.

```
start
   : initop isac_instr start
   | initop
```

Nástroj Bison očekává, že startovní nonterminál se bude jmenovat *start*. Tím pádem je startovním nonterminálem spojených jazyků nonterminál z jazyka assembleru. Jeho další nonterminál *initop* pak zpracuje veškeré řetězce jazyka assembleru dokud nenarazí na první instrukci (zpracovává direktivy, definice návěští apod.). Nonterminál *initop* je také možné přepsat na prázdný řetězec.

Zpracování nonterminálu jazyka instrukcí *isac_instr* pak způsobí načtení jedné instrukce ze vstupu a její překlad do binární podoby. Tyto dva kroky se opakují, dokud se nedojde na konec vstupu.

5.3.4 Možné řešení přidání podpory výrazů pro assembler verze 0.1 a 0.2

Jelikož jsme způsobem popsaným v předchozí podkapitole spojili jazyk assembleru a jazyk instrukcí do jazyka jediného, je možné poměrně jednoduchým způsobem zavést podporu výrazů jako operandů instrukcí.

Naformátováno: 1. odstavec

Nejprve popíší, jakým způsobem se generovaný assembler vypořádává s konstantními čísly a symboly jako operandy instrukcí. Poté naznačím, jakým způsobem by se dal assembler rozšířit tak, aby podporoval i výrazy.

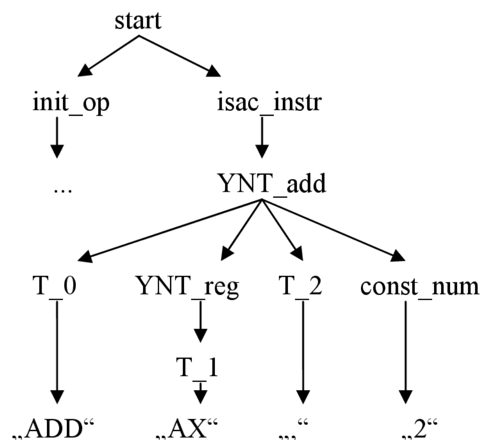
Jazyk assembleru (v. 0.1) obsahuje terminál pojmenovaný *const_num*. Ten představuje konstantní číslo zadané ve zdrojovém souboru pro assembler. JI ví tom, že tento terminál existuje a při generování gramatiky JI se tam, kde se očekává v operandu instrukce konstantní číslo, použije právě tento nonterminál.

Naformátováno: 2. odstavec

Ve verzi 0.2 je k terminálu *const_num* ještě přidán další terminál *symbol*. Ten představuje, jak jeho název napovídá, libovolný symbol JA jako je např. návěští nebo číselná konstanta.

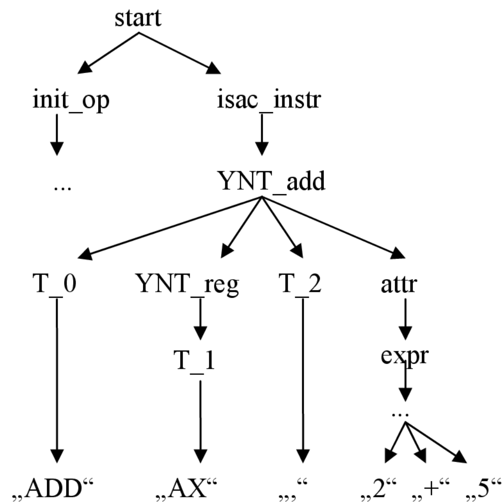
Podívejme se tedy, jak by probíhal překlad (derivační strom) například řetězce `ADD AX, 2` (generované terminály JI mají předponu T a nonterminály předponu YNT):

Naformátováno: 2. odstavec



Obrázek 3.2 Ukázka derivačního stromu pro překlad instrukce s konstantním číslem jako operandem

Jak ale bylo již naznačeno na konci podkapitoly 3.2, jedním z nedostatků assembleru verze 0.2 je absence podpory výrazů. Pokud však nonterminál *const_num* nahradíme jiným terminálem, který se pak pomocí dalších pravidel dá přepsat na výraz, můžeme jednoduše podporu výrazů zavést. Na následujícím obrázku si ukážeme, jak vypadal derivační strom pro řetězec `ADD AX, 2+5`.



Obrázek 5.3 Ukázka derivačního stromu pro překlad instrukce, jejíž operandem je výraz

Jak můžeme vidět na obrázku 5.3, terminál *const_num* můžeme nahradit nějakým obecným nonterminálem *attr* (z gramatiky jazyka assembleru), který zastupuje jak všechny konstantní čísla, symboly a i výrazy (taktéž z gramatiky JA). Gramatika JA i gramatika JI jsou sjednocené a tak není problém si takto výrazy v instrukcích zavést.

Naformátováno: 2. odstavec

Mohly by nastat různé kolize, pokud by byl jazyk instrukcí nadefinován nekompatibilně s jazykem assembleru, ale tyto kolize pravidel by nástroj Bison odhalil a tvůrce assembleru (nebo návrhář instrukční sady) by okamžitě věděl, že někde je problém.

V této kapitole jsem chtěl především vysvětlit rozdíl mezi jazykem assembleru a jazykem instrukcí. Dále také to, jakým způsobem by se pro assembler verze 0.2 daly zavést výrazy.

Naformátováno: 2. odstavec

5.4 Překlad instrukcí pomocí dvojcestných párových automatů

V assembleru verze 0.3 dojde ke změně postupu při překladu instrukcí. Jazyk instrukcí bude namísto jazyka bezkontextového reprezentován jazykem regulárním a instrukce se budou překládat pomocí dvojcestných párových automatů.

Naformátováno: 1. odstavec

5.4.1 Co jsou to dvojcestné párové automaty (DPA)

Jazyk instrukcí popisující víceméně jakoukoli instrukční sadu je ve svém principu jazykem regulárním. Pak se může zdát zbytečné instrukce překládat pomocí LR gramatiky a pro překlad může stačit vhodný konečný automat zpracovávající regulární jazyk vytvořený na základě popisu instrukční sady v jazyce ISAC. Roman Lukáš se touto otázkou zabýval a navrhnul překlad pomocí *dvojcestných párových automatů*. Dokázal také, že všechny instrukční sady, které můžeme pomocí jazyka ISAC popsat, lze reprezentovat regulárním jazykem a pomocí DPA přeložit. V prezentacích [32], [33] a v článku [34] můžete o párových automatech nalézt podrobnější informace.

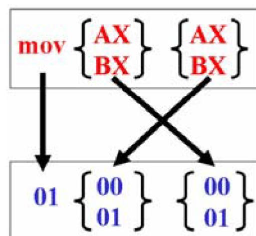
Stále, stejně jako ve verzích 0.1 a 0.2, chceme dodržet co největší podobnost mezi překladem z jazyka strojových instrukcí do binární podoby a nazpět. Tj., aby si assembler a disassembler byly co nejvíce podobní, fungovaly na stejných principech a sdíleli co největší množství kódu.

V další podkapitole pouze stručně naznačím, jakým způsobem bude překlad pomocí DPA probíhat.

5.4.2 Překlad pomocí DPA

Na příkladu překladu řetězce `mov AX, BX` ukáži, jak bude probíhat překlad do binární podoby a zpět. Obrázky jsou převzaty z prezentace [32].

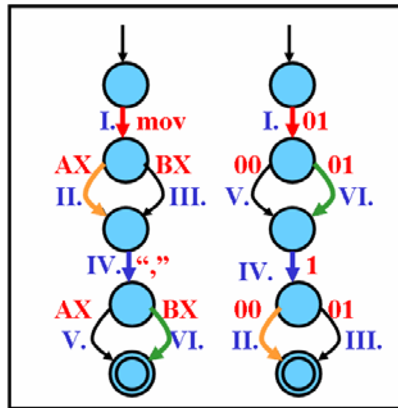
Na obrázku 5.4 můžeme vidět způsob zakódování instrukce `mov` do binární podoby. Operační kód instrukce se ukládá na prvních dvou bitech a pro `mov` je `01`. Za oper. kódem následují dva operandy reprezentující zdrojové a cílové registry.



Obrázek 5.4 Schéma překladu instrukce mov

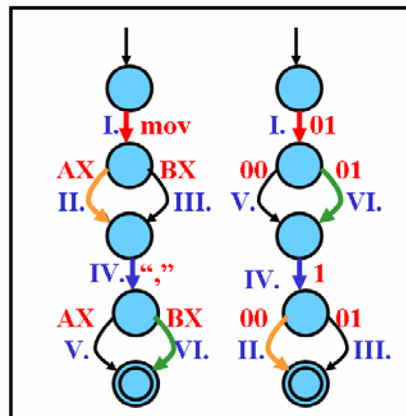
Na následujícím obrázku 5.5 můžeme vidět dva párové automaty – jeden pro textovou podobu instrukce a druhý pro binární. Jednotlivé hrany automatů jsou označeny římskými čísly. Tyto čísla určují vztah mezi hranami automatů.

Necháme tedy řetězec `mov AB, BX` zpracovat prvním automatem pro textovou podobu instrukce. Hrany, přes které se projde jsou tyto: I., II., IV., a VI.. Nyní si označíme ve druhém automatu hrany se stejnými čísly. Vytvořili jsme si cestu druhým automatem. Když teď projdeme po označených hranách od počátečního k finálnímu stavu, vytvoří se nám binární podoba původní instrukce s operandy: `0101100`.



Obrázek 5.5 Párový automat: Překlad ASM→BIN

Analogicky k předchozímu příklady probíhá překlad opačným směrem. Ten můžete vidět na následujícím obrázku 5.6.



Obrázek 5.6 Párový automat: Překlad BIN→ASM

V této podkapitole jsme si ve zkratce ukázali, jak probíhá překlad pomocí DPA.

To, jakým způsobem se automaty vytváří a jak zpracovávají atributy už je z naprosté většiny vymyšleno a základní verze generátoru dvojcestných párových automatů byla implementována. Detaily o implementaci generátoru automatů můžete nalézt v prezentaci [33].

5.4.3 Výhody překladu pomocí dvojcestných párových automatů

Rychlost zpětného překladu je zásadním důvodem, proč se překlad pomocí DPA zavádí. Rychlý simulátor je důležitou částí projektu a návrháři procesoru umožňuje ověřovat, zda má procesor požadované vlastnosti. Pokud by simulace trvala příliš dlouho, snížilo by to efektivitu vývoje.

Dalším problémem, se kterým jsme se potýkali byla nejednoznačnost při zpětném překladu. Ta se použitím DPA vyřešila. Detaily viz [35], kap. 2.3.2.

5.4.4 Jaké jsou hlavní rozdíly při změně algoritmu pro překlad instrukcí

Prvním rozdílem je to, že již nebude možné jednoduše spojit gramatiky jazyka instrukcí a jazyka assembleru (viz podkapitola 5.3) tak, aby překladač vygenerovaný nástrojem Bison zároveň prováděl překlad instrukcí do jejich binární podoby pouze s pomocí jednoduchých sémantických pravidel. Jak již bylo uvedeno, ve verzích 0.1 a 0.2 bylo spojení gramatik jazyka instrukcí a jazyka assembleru ve zdrojovém souboru syntaktického analyzátoru realizováno takto:

Naformátováno: 1. odstavec

```
start
   : initop isac_instr start
   | initop
```

Podobný princip by mohl zůstat, pouze s tím, že nonterminál *isac_instr* by byl nahrazen nějakým mechanizmem pro zpracování instrukcí – kodérem instrukcí. Tento kodér by pracoval na principu dvojcestných párových automatů.

Naformátováno: 2. odstavec

My ale potřebujeme zavést do assembleru i podporu výrazů v operandech instrukcí. Spojení gramatik není možné takto jednoduše použít z důvodu, že máme instrukce popsané regulárním jazykem. Dále, obecné výrazy není možné popsat pomocí regulárního jazyka a výrazy se musí zpracovat předtím, než se celá instrukce kodéru předá k přeložení.

Tento princip *předzpracování* mě dovedl na myšlenku použití preprocesoru. Původní představa byla, že preprocesor by byl samostatný překladač, který by mohl provést víceméně libovolnou transformaci vstupního souboru (viz [35]). Takovýto preprocesor by však byl závislý na cílové architektuře. Musel by znát gramatiku jazyka instrukcí, aby mohl rozlišit mezi konstrukcemi jazyka instrukcí a jazyka assembleru.

Naformátováno: 2. odstavec

Rozumnější asi bude nechat preprocesor nezávislý na cílové architektuře, i když tím omezíme jeho sílu. Současná představa může být taková, že půjde o preprocesor schopný textových náhrad, podobně jako preprocesor jazyka C.

6 Návrh assembleru

V této kapitole si podrobně popíšeme, jakým způsobem bude assembler fungovat a z jakých součástí se bude skládat.

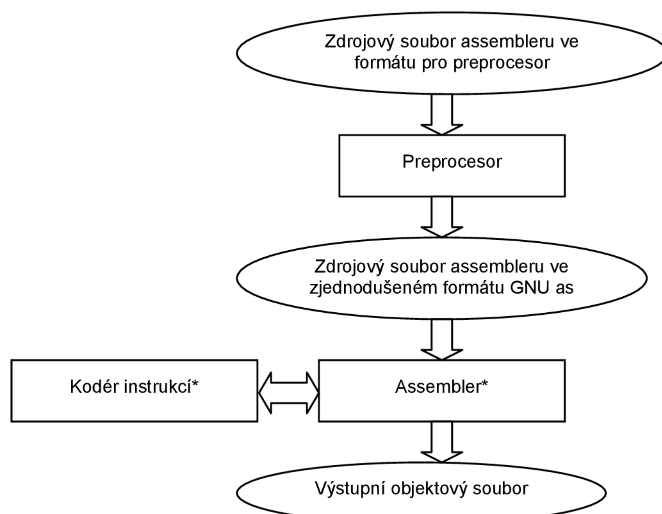
6.1 Cíle assembleru projektu Lissom

Nežli se pustíme do popisu architektury assembleru, stanovíme si požadavky a některé vlastnosti, které na assembler klademe:

- assembler bude umět překládat instrukce instrukční sady popsané pomocí jazyka ISAC, překlad samotných instrukcí musí být nezávislý na zbytku assembleru,
- bude podporovat nejzákladnější direktivy určující podobu výsledného objektového souboru (definice sekcí, určení absolutní pozice v sekce v paměti),
- v případě potřeby musí být možnost snadno definovat nové direktivy, které jinak upravují výsledný objektový soubor,
- bude podporovat symboly a výrazy, symboly mohou být definovány buď jako návěští nebo pomocí direktiv jako konstanty,
- podpora zarovnávání kódu a dat na určené paměťové hranice,
- konzistentní zpracování chyb a jejich výpis,
- možnost jednoduše změnit formát objektového souboru,
- pomocí preprocesoru půjde částečně měnit formát vstupního souboru,
- ukládání ladicích informací do objektového souboru,
- podpora ladicích a informačních výpisů (tzv. listing výpisy, obsahují informace o tom, jak byl vstupní soubor přeložen, obvykle obsahují tři sloupce – adresa, zakódovaná instrukce a řádka původního textu ze zdrojového souboru),
- bude sloužit jako překladač výstupu překladače vyššího programovacího jazyka.

6.2 Postup při překladu

V kapitole 4 jsme si popsali základní architekturu každého assembleru a na jejím konci jsou uvedeny některé závěry, kterými se budeme řídit při návrhu našeho assembleru. Celkový pohled na překlad pomocí assembleru můžete vidět na následujícím obrázku (součásti označené hvězdičkou jsou závislé na cílové architektuře.):



Obrázek 6.1: Schéma překladu assemblerem

Vstupní soubor je nejprve předpřipraven preprocesorem. Ten umožňuje provádět pouze jednoduché textové náhrady, jako je změna názvů direktiv a jejich parametrů, provedení podmíněného překladu a rozbalení maker a také, v případě potřeby, může seřadit subsekce v původním zdrojovém souboru do celých sekcí (subsekce viz 5.4). Vstupní soubor je tedy preprocesorem transformován do souboru ve zjednodušeném formátu GNU as (viz direktivy v příloze B). Assembler pak, s použitím kodéru instrukcí, přeloží jeho vstupní soubor a vygeneruje objektový soubor.

Naformátováno: 2. odstavec

Preprocesorem se již dále nebudeme zabývat, řekněme, že bude umožňovat provádět operace popsané v předchozím odstavci. Zaměříme se spíše na samotný assembler a kodér instrukcí.

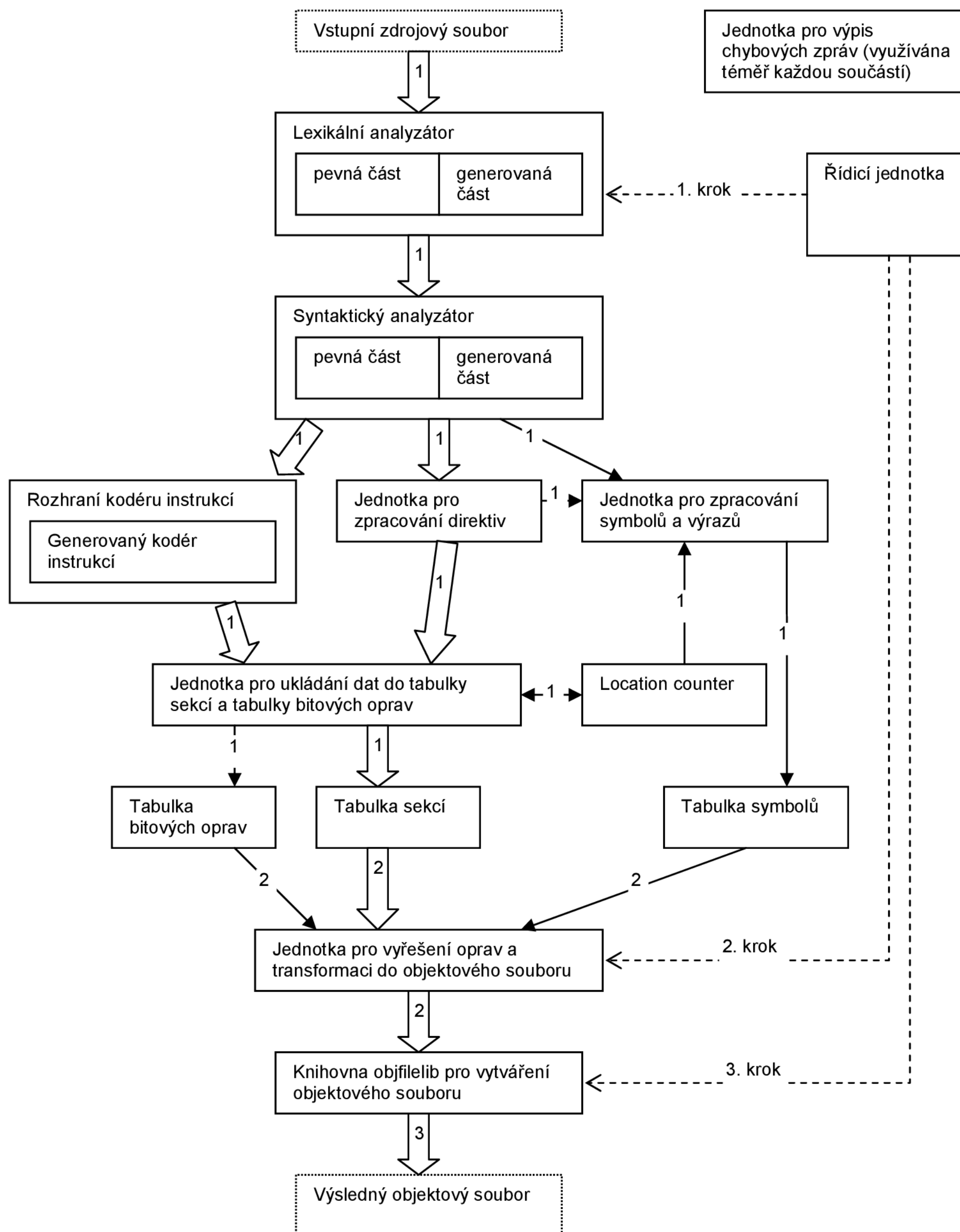
6.3 Assembler a jeho součásti

Překlad assemblerem probíhá ve třech krocích:

1. načtení zdrojového souboru a naplnění tabulek sekcí, symbolů a bitových oprav na základě těchto informací
2. výpočet hodnot symbolů, provedení bitových oprav v sekcích a naplnění struktur objektového souboru
3. uložení objektového souboru na disk

Ve své podstatě se jedná o dvouprůchodový assembler, v prvním kroku načteme informace ze vstupního souboru a v kroku druhém vyřešíme bitové opravy budoucích symbolů. Na následujícím obrázku 6.2 jsou znázorněny jednotlivé součásti assembleru. Dále, pomocí širokých šipek tok kódu a

dat ze vstupního souboru assembleru do výstupního objektového souboru. Pomocí úzkých šipek je pak naznačen tok dalších, doplňujících informací. U každého toku informací je pak pomocí čísla 1, 2 nebo 3 určeno, ve kterém kroku překladu probíhá.



Obrázek 6.2: Jednotlivé součásti assembleru a kroky překladu

Tento diagram znázorňuje funkci assembleru spíše z pohledu toho, jak se vstupní zdrojový soubor překládá. V příloze E naleznete podrobnější diagram, který zobrazuje součásti asembleru spíše z programátorského hlediska a jsou v něm specifikovány vztahy mezi těmito jednotlivými součástmi.

To by k základní představě, jak bude assembler fungovat mohlo stačit a můžeme se pustit do podrobnějšího popisu jeho součástí a vztahů mezi nimi.

6.4 Lexikální a syntaktický analyzátor a kodér instrukcí

Lexikální analyzátor, syntaktický analyzátor a kodér instrukcí jsou jedinými částmi assembleru, které jsou závislé na cílové architektuře, přesněji její instrukční sadě.

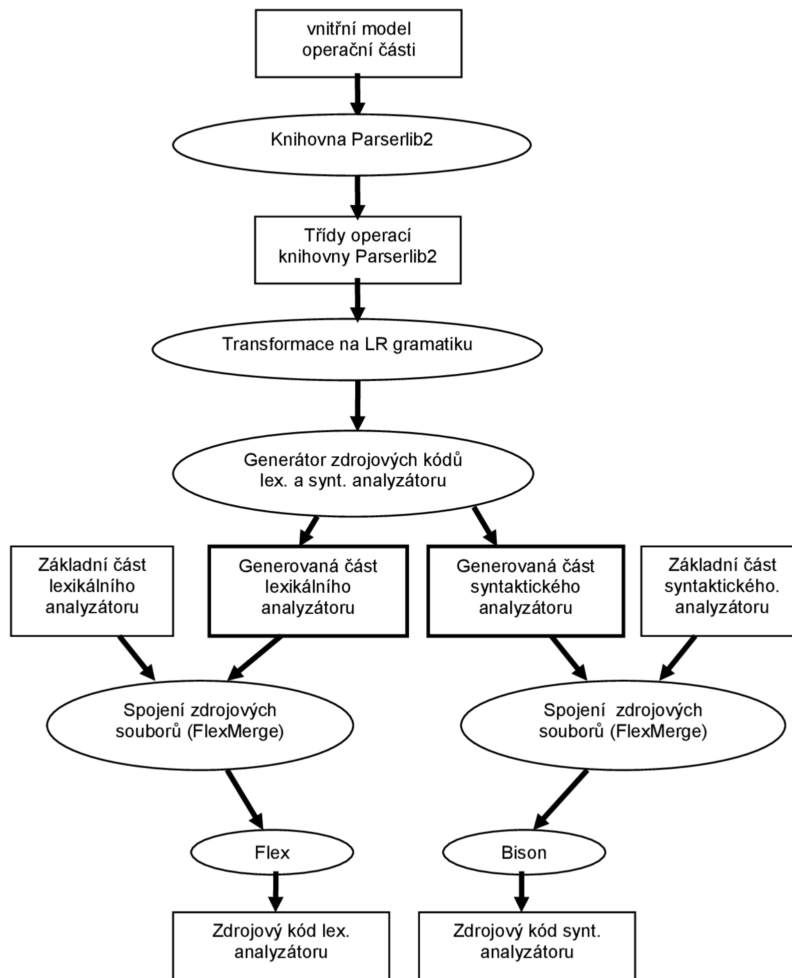
K oddělení překladu jazyka instrukční sady a jazyka assembleru (viz kap. 3.3) jsem se rozhodnul z důvodu, že překlad instrukcí probíhá pomocí dvojcestných párových automatů - překládají regulární jazyk a gramatika jazyka assembleru bude LR gramatikou z důvodu podpory překladu výrazů (ostatní konstrukce jazyka assembleru ale lze popsat pomocí regulárního jazyka). Dalším důvodem pro oddělení překladu je vzájemná nezávislost assembleru a kodéru instrukcí a tak se kodér může vyvíjet nezávisle na zbytku assembleru.

Zatím mějme představu, že kodér instrukcí je konečný automat (což ve skutečnosti opravdu je) a od syntaktického analyzátoru dostává postupně terminály jazyka instrukcí a pomocí přechodů iniciovanými těmito terminály provádí svůj překlad. Syntaktický analyzátor musí tedy vědět, které terminály patří do jazyka instrukcí a u pravidel, které je využívají, provádět sémantickou akci v podobě zaslání tohoto terminálu kodéru instrukcí. To, jakým způsobem se gramatika jazyka instrukcí a jazyka assembleru propojí je naznačeno v kapitole 3.3.3.

Naformátováno: 2. odstavec

Vytváření lexikálního a syntaktického analyzátoru assembleru je částečně podobné jejich vytváření v předchozích verzích. Na následujícím diagramu můžete vidět postup při jejich generování. Detailnější popis naleznete v dokumentu [15], kapitoly 3.2 a 3.4.

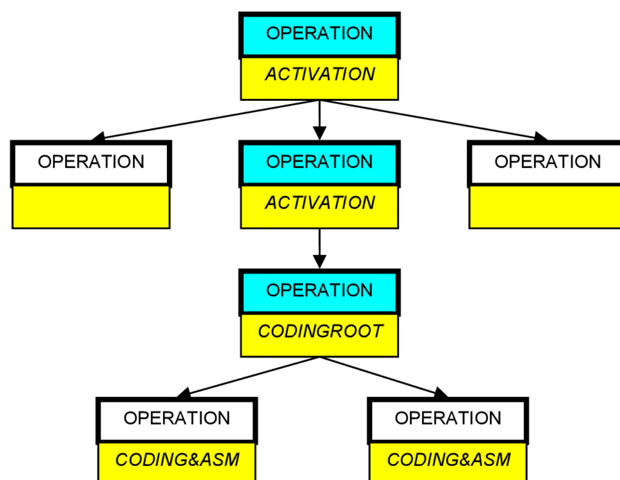
Naformátováno: 2. odstavec



Obrázek 6.3: Generování lexikálního a syntaktického analyzátoru

Algoritmus pro generování vychází z algoritmu používaného v předchozích verzích assembleru. Využívá informace získané z popisu v jazyce ISAC pomocí knihovny Parserlib2. Implementace její části, která se stará o načtení informací o operacích a její zpřístupnění, je popsána v příloze A. Generování lexikálního a syntaktického analyzátoru pak probíhá v těchto krocích:

1. Začneme od operace nazvané "main" a nalezneme všechny sekce CODINGROOT. Ty jsou počátečními body, od kterých začíná popis instrukční sady. Následující obrázek předchozí větu ilustruje (převzato z [9]).



Obrázek 6.4: Význam sekce CODINGROOT jako počátečního bodu, od kterého začíná popis instrukcí instrukční sady.

2. Na počáteční body aplikujeme algoritmus podobný algoritmu popsanému v článku [8]. Ve své podstatě jde o to, že získáme gramatiku jazyka instrukční sady – tj. gramatiku, která určuje, jakým způsobem se instrukce zapisují. Žádné další informace, než je syntaxe jazyka instrukcí, nepotřebujeme.

3. Na základě získané gramatiky vygenerujeme části zdrojových souborů lexikálního a syntaktického analyzátoru pro nástroje Flex a Bison. Postup generování je v podstatě stejný jak byl popsán v [15], kapitolách 3.2 a 3.4., pouze s tím rozdílem, že se sémantická pravidla generují pouze pro pravidla, které mají na levé straně terminály. Pro každý tento terminál se zavolá funkce kodéru instrukcí – předá se mu načtený terminál (token).

Tímto způsobem získáme lexikální a syntaktický analyzátor. Zbývá nám ještě kodér instrukcí. Ten překládá instrukce pomocí dvojcestných párových automatů (viz kap. 3.4) a popis jeho vytváření naleznete v dokumentech [18], [19] a [20]. Na příkladě si ukážeme, jak vypadá výsledná gramatika pro jednoduchou instrukční sadu a jakou podobu mají generované soubory lexikálního a syntaktického analyzátoru.

6.4.1 Příklad generovaných částí lexikálního a syntaktického analyzátoru

V následujícím kódu si pomocí jazyka ISAC nadefinujeme instrukční sadu procesoru. Pro jednoduchost mějme pouze jedinou instrukci *Mov*.

```
OPERATION Axreg {
      ASSEMBLER { "AX" }
      CODING { 0b01 }
}
```

```
OPERATION Bxreg {
      ASSEMBLER { "BX" }
      CODING { 0b10 }
}

GROUP Reg = axreg, bxreg;

OPERATION Mov
{
      INSTANCE Reg ALIAS { src, dst };
      ASSEMBLER { "MOV" src "," dst };
      CODING { 0b1100 src dst };
}

GROUP Instructions = Mov;

OPERATION AnyInstruction IN pipeline.DE
{
      INSTANCE Instructions ALIAS { instr };
      ASSEMBLER { instr };
      CODING { instr };
}

OPERATION DecodeInstr
{
      INSTANCE AnyInstruction ALIAS { anyinstr };

      CODINGROOT {
            //zde začíná popis kódování instrukcí, v hardwaru
            //procesoru se data, která se mají dekódovat, nalézají
            //v registru pipeline označeném jako FE_DE_inst[FE_DE_pc]
            anyinstr(FE_DE_inst[FE_DE_pc]);
      }
}


OPERATION main
{
      INSTANCE DecodeInstr ALIAS { decode };
      ACTIVATION {
            decode;
      };

}
```

V prvním kroku nalezneme všechny výskyty sekce CODINGROOT. V tomto příkladu máme jeden v operaci DecodeInstr. Obecně ale sekcí CODINGROOT může být definováno více.

V druhém kroku použijeme nalezené sekce CODINGROOT a získáme gramatiku jazyka instrukční sady. Počáteční nonterminál si pojmenujeme isac_instr a pro každou operaci Op, ve které se vyskytuje sekce CODINGROOT vytvoříme přepisovací pravidlo ve tvaru <isac_instr> -> <Op>. Zde máme takovéto pravidlo jediné (1). Poté začneme postupně procházet instance operací a vytvoříme pravidla popisující jazyk instrukční sady. Výsledná gramatika pro tento příklad bude mít následující podobu:

G = (*N*, Σ, *P*, <isac_instr>), kde:

- *N* = { <isac_instr>, <DecodeInstr>, <AnyInstruction>, <Instructions>, <Mov>, <Reg>, <Bxreg>, <Axreg> }
- Σ = { “MOV”, “,”, “AX”, “BX” }
- *P* = { **1:** <isac_instr> → <DecodeInstr>,
  2: <DecodeInstr> → <AnyInstruction>,
  3: <AnyInstruction> → <Instructions>,
  4: <Instructions> → <Mov>,
  5: <Mov> → "MOV" <Reg> "," <Reg>,
  6: <Reg> → "AX",
  7: <Reg> → "BX" }

Na základě této gramatiky již můžeme vygenerovat části lexikálního a syntaktického analyzátoru.

Část lexikálního analyzátoru (zdrojový kód pro nástroj Flex):

```
%%
"MOV"/([^_[:alpha:][:digit:]]) return T_0;
","return T_1;
"AX"/([^_[:alpha:][:digit:]]) return T_2;
"BX"/([^_[:alpha:][:digit:]]) return T_3;
```

Část syntaktického analyzátoru (zdrojový kód pro nástroj Bison):

```
%token T_0 /* "MOV" */
%token T_1 /* "," */
%token T_2 /* "AX" */
%token T_3 /* "BX" */

%%
YNT_isac_instr
:YNT_DecodeInstr
;

YNT_DecodeInstr
: YNT_AnyInstruction
;

YNT_AnyInstruction
: YNT_Instructions
;

YNT_Instructions
: YNT_Mov
;

YNT_Mov
: T_0
//kodéru instrukcí odešleme token MOV ihned, jakmile na něj
//při překladu narazíme, podobně i u dalších terminálů
{ SendTokenToInstrCoder(T_0); }
YNT_Reg
T_1
```

```
{ SendTokenToInstrCoder(T_1); }
YNT_Reg
;

YNT_Reg:
: T_2
{ SendTokenToInstrCoder(T_2); }

| T_3
{ SendTokenToInstrCoder(T_3); }
```

Konec příkladu.

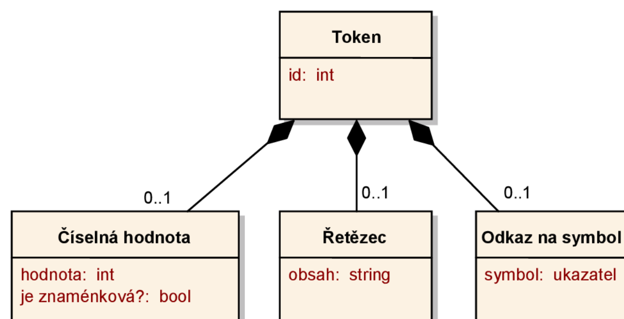
6.4.2 Rozhraní mezi syntaktickým analyzátorem a kodérem instrukcí, jednotkou pro zpracování direktiv a jednotkou pro zpracování symbolů výrazů

V této kapitole si popíšeme, jaké informace posílá syntaktický analyzátor pomocí sémantických pravidel jednotlivým součástem assembleru.

Naformátováno: 1. odstavec

Základní jednotkou komunikace je *token*, ten má stejný význam jako terminál gramatiky vzniklé sjednocením gramatik jazyka instrukcí a jazyka assembleru (s výjimkou symbolů z tabulky symbolů). Tokeny jsou identifikovány unikátní celočíselnou hodnotou, jde o stejnou hodnotu, jakou tomuto tokenu přiřadil nástroj Flex při generování lexikálního analyzátoru. Token může mít navíc nějakou hodnotu atributu, atributem hodnota je buď celé číslo, znakový řetězec a nebo odkaz na symbol. Typ atributu se rozliší podle identifikátoru tokenu.

Naformátováno: 2. odstavec



Obrázek 6.5: Token s atributem typu číslo, řetězec a nebo odkaz na symbol

Pomocí sémantických pravidel je určeno, které součásti se tokeny budou posílat. Syntaktický analyzátor tedy buď jednotlivé tokeny, seznamy a nebo stromy tokenů.

- Kodéru instrukcí se posílají jednotlivé tokeny.
- Jednotce pro zpracování direktiv se posílají seznamy tokenů, ty představují direktivy s jejich parametry.
- U jednotky pro zpracování symbolů a výrazů jsou dvě možnosti:
  - Při překládání výrazu se pomocí sémantických pravidel vytváří strom tokenů a ten je pak po přeložení celého výrazu zaslán. Tato jednotka vytvoří z výrazu symbol a vrátí syntaktickému analyzátoru token představující tento výraz - pokud je možné výraz

vyčíslit, tak vrátí token s číselnou hodnotou, pokud ne, tak vytvoří nový symbol představující tento výraz a vrátí token s odkazem na tento symbol.
- Při definici návěští se zasílají jednotlivé tokeny s řetězcovým atributem, ten určuje název symbolu (tento jediný token je vlastně také strom, avšak pouze s kořenovým prvkem).

6.5 Jednotka pro zpracování direktiv

Jak bylo uvedeno v předchozí kapitole, tato jednotka dostává od syntaktického analyzátoru seznamy tokenů. První z těchto tokenů vždy určuje, o kterou direktivu se jedná. Na základě této direktivy a jejích parametrů provede požadovanou akci. Jednotka pro zpracování direktiv provádí následující úkoly:

- vytváření nových sekcí a nastavování jejich vlastností,
- generování inicializovaných dat,
- zpracování direktiv řídicích zarovnávání na paměťové hranice,
- definice symbolů s konstantní hodnotou.

Odstraněno: ¶

Při vykonávání prvních tří úkolů komunikuje pouze s jednotkou pro ukládání dat do tabulky sekcí a tabulky bitových oprav.

Seznam direktiv, které budou podporovány v počátečních fázích vývoje assembleru naleznete v příloze B.

6.6 Jednotka pro zpracování symbolů a výrazů

Tato jednotka provádí:

- zpracování a vytváření symbolů definovaných jako návěští
- vytváření symbolů z výrazů na základě stromu tokenů od syntaktického analyzátoru a kontrolu, zda jsou tyto výrazy platné

6.6.1 Symboly

Symboly mohou být definovány ve zdrojovém kódu buď jako návěští a nebo pomocí direktiv jako konstanty. Dále, i pro výrazy se budou vytvářet symboly. Popíšeme si, jaké informace si o symbolech potřebujeme ukládat. Nejprve si uvedeme do jakých kategorií se dají symboly rozdělit.

První druh členění symbolů může být podle typu jejich hodnoty:

- relokatabilní - Hodnota je vzdáleností (offsetem) od začátku sekce, kde se daný symbol vyskytuje. (Dá se také říci, že jde o adresou relativní k dané začátku sekce, to se ale může plést s relativními adresami využívanými pro relativní skoky.)
- absolutní – Hodnota symbolu je pevně daná.

- nedefinovaný – Hodnota symbolu není zatím známa, tento typ budeme používat pro "budoucí symboly" (viz kap. 3.2.1), hodnota nemusí být ani známa po přeložení celého vstupního souboru.

Druhý typ členění je podle toho, zda jsou viditelné pro linker:

- lokální – Standardně je každý symbol lokální, to znamená, že se na symbol můžeme odkazovat pouze v rámci zrovna zpracovávaného zdrojového souboru.
- globální – Pomocí speciální direktivy (např. .global) je možné změnit typ symbolu z lokálního na globální. To znamená, že je po překladu uložen v tabulce symbolů a jiné objektové soubory mohou využívat jeho hodnotu.

U symbolů dále potřebujeme jeho název, hodnotu a u relokatibilních symbolů, informaci o tom, ve které sekci byl definován.

Následující příklad ilustruje různé typy symbolů (jsou použity direktivy popsané v příloze B):

```
1    .section kód, "t" ;definice kódové sekce
2
3    ;do registru AX uložíme adresu, kde řetězec začíná
4    MOV AX, zpráva
5
6    ;do registru BX uložíme délku řetězce
7    MOV BX, dvojnásobná_délka/2
8
9    ;zavoláme rutinu pro výpis řetězce, ta očekává v
10   ;registrech AX a BX uvedené hodnoty
11   CALL vypiš_řetězec
12
13   .section data, "d" ;definice datové sekce
14
15   ;návěští zpráva představuje adresu, kde začínají data této zprávy
16  zpráva:
17   .bit 8, 'A', 'h', 'o', 'j'
18
19   ;návěští konec_zprávy pak představuje adresu těsně za
20   ;posledním bajtem zprávy
21  konec_zprávy:
22
23   ;pomocí direktivy .equiv si nadefinujeme symbol, jehož hodnotou
24   ;je délka zprávy v bajtech vynásobená dvěma (dvojnásobek hodnoty je
25   ;zde pouze demonstračně)
26   .equiv dvojnásobná_délka, (zpráva - konec_zprávy)*2
```

V tomto příkladě jsou uvedeny snad všechny možné typy symbolů, tyto symboly se vytvářejí pomocí jednotky pro zpracování symbolů a výrazů ve chvíli, kdy jsou definovány či poprvé využity:

- `zpráva` je vytvořena jako nedefinovaný symbol (na řádku 4), pak, na řádku 16 je její typ změněn na relokatibilní a je nastavena jeho hodnota na 0 (je úplně na začátku sekce data),
- pak máme výraz `dvojnásobná_délka/2`, pro ten se vytvoří pseudosymbol s unikátním názvem #expr_7_8 (číslo řádku a sloupce v textu, kde se nachází) a hodnota tohoto symbolu je popsána výrazem `dvojnásobná_délka/2` (o výrazech viz dále),

- `vypiš_řetězec` je nedefinovaným symbolem,
- zápis konstanty 8 v prvním parametru direktivy **.bit** je z pohledu překladače výrazem, ten však lze vyčíslit a pseudosymbol se pro něj nevytváří,
- `konec_zprávy` je vytvořen jako relokatibilní symbol s hodnotou 4,
- `dvojnásobná_délka` je symbolem, který zastupuje výraz `(zpráva – konec_zprávy)*2`, tento výraz je možné, ve chvíli, kdy se překládá, vyčíslit a tak není potřeba pro tento výraz vytvářet pseudosymbol a jeho vypočítaná hodnota 8 (=(4-0)*2) se nastaví jako hodnota symbolu `dvojnásobná_délka`.

6.6.2 Symboly, výrazy a jejich atributy

Na konci minulé kapitoly jsme se již o vyčíslování výrazů zmínili. Nejprve si řekneme, z čeho se výrazy skládají a jakým způsobem se vypočítávají.

Výrazy si můžeme definovat pomocí následujícího zápisu:

<výraz> ::= operátor <operand> | <operand> operátor <operand> | <operand>
<operand> ::= <výraz> | symbol | konstanta

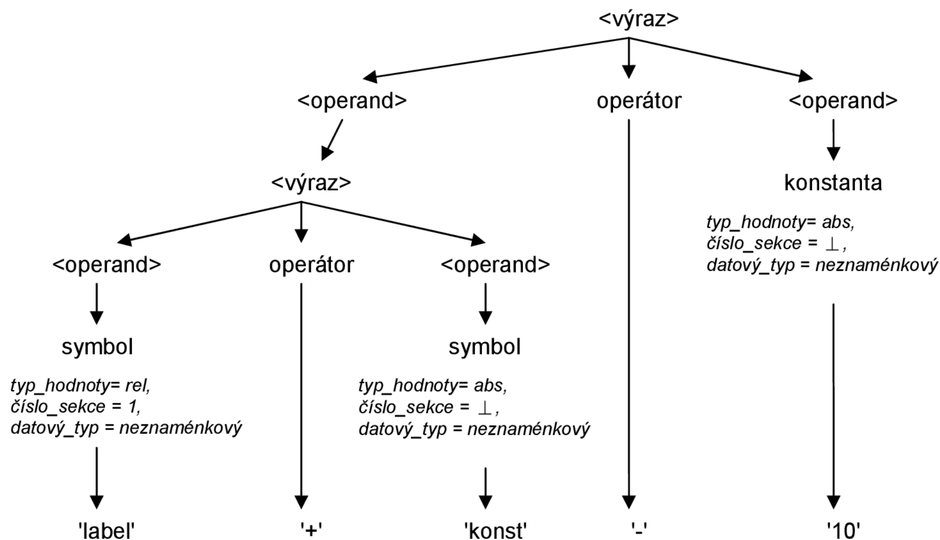
Popíšeme si atributy, které budeme potřebovat pro vyhodnocování výrazů. Symboly mohou mít tři typy hodnot: *relokatibilní (rel)*, *absolutní (abs)* a *nedefinované (nedef)*. U relokatibilních symbolů navíc potřebujeme číslo sekce. Dále datové typy konstant mohou být celočíselné *znaménkové* a celočíselné *neznaménkové*. Zavedeme si tedy tři atributy, které se týkají jejich typu – *typ hodnoty, číslo sekce* a *datový typ*. Dalším atributem je navíc samotná *hodnota*, tento atribut však pro nás není nyní důležitý.

typ_hodnoty $\in$ *{rel, abs, nedef}*
číslo_sekce $\in \{\mathbf{Z} \cup \perp\}$
datový_typ $\in$ *{ neznámý, znaménkový, neznaménkový}*
hodnota $\in \mathbf{Z}$

Pro hodnoty atributů platí následující pravidla:

typ_hodnoty = rel $\Leftrightarrow$ *datový_typ = neznaménkový,*
typ_hodnoty = abs $\Leftrightarrow$ *datový_typ* $\neq$ *neznámý* a
číslo_sekce $\neq \perp \Leftrightarrow$ *typ_hodnoty = rel* .

Na příkladě jednoho výrazu si ukážeme jak se vytváří strom pomocí pravidel pro popis výrazu.



Obrázek 6.6: Příklad vytváření pseudosymbolu pro výraz (label + konst) - 10

6.6.3 Operátory a tok atributů

Toto jsou operátory, které bude assembler podporovat:

- binární operátory: *, /, %, <<, >>, |, &, !, +, -
- unární operátory: -, ~

Jde o stejné operátory, jaké podporuje GNU assembler a mají naprosto stejný význam jako v jazyce C (viz [33], kapitola Integer Expressions).

Nyní si popíšeme typový systém, který nám definuje povolené operace a tok atributů nad prvky, ze kterých jsou výrazy složeny. Atribut *hodnota* nyní není důležitý a tak ho zanedbáme. Pro stručnost zápisu si zavedeme atribut *atributy*, ten sdružuje všechny atributy prvků. Zápis *chyba* značí, že popisovaná operace nad atributy není povolena a při vyčíslování výrazu assembler vypíše chybu.

Aplikace unárních operátorů

<výraz> ::= operátor <operand>

Unární operátory nejsou nad prvky s *relokatabilním typem hodnoty* povoleny. Pokud je unárním operátorem mínus ('-'), automaticky mění *datový typ* na *znaménkový*.

if <operand>.*typ_hodnoty = rel*
then *chyba*
else
    if operátor = '-'

```
        then <výraz>.datový_typ = znaménkový, <výraz>.typ_hodnoty =
                <operand>.typ_hodnoty, <výraz>.číslo_sekce = <operand>.číslo_sekce
        else  <výraz>.atributy = <operand>.atributy
        endif
    endif
```

Aplikace binárních operátorů

```
<výraz>     ::= <operand1> operátor <operand2>
```

Pokud je *typ hodnoty* jednoho z operandů *nedefinovaný*, pak i celý výraz má *nedefinovaný* typ *hodnoty*. Dále, pokud oba operandy mají *relokatibilní typ hodnoty*, pak jediný povolený operátor je mínus. Pokud pouze jeden z operandů má *relokatibilní typ hodnoty* a druhý má *absolutní,* pak jsou povolenými operátory pouze +, - a bitové posuvy, i výsledný výraz má *relokatibilní typ hodnoty.* Pokud mají oba operandy *absolutní typ hodnoty,* tak výsledek má také *absolutní typ hodnoty* a pokud alespoň jeden z operandů má *znaménkový datový typ*, tak i výsledek má *znaménkový datový typ.*

```
if <operand1>.typ_hodnoty = nedef or <operand2>.typ_hodnoty = nedef
then <výraz>.typ_hodnoty = nedef, <výraz>.číslo_sekce = ⊥,
        <výraz>.datový_typ = neznámý
else
        //ani jeden operand nemá nedefinovaný typ hodnoty
        if <operand1>.typ_hodnoty = rel  or <operand2>.typ_hodnoty = rel
        then
                //alespoň jeden operand má relokatibilní typ hodnoty
                if <operand1>.typ_hodnoty = rel  and <operand2>.typ_hodnoty = rel
                then
                        //oba operandy mají relokatibilní typ hodnoty
                        if operátor = '-' and <operand1>. číslo_sekce = <operand2>. číslo_sekce
                        then <výraz>.typ_hodnoty = abs, <výraz>.číslo_sekce =
                                <operand1>.číslo_sekce, <výraz>.datový_typ = znaménkový
                        else  chyba
                        endif
                else
                        // právě jeden operand má relokatibilní typ hodnoty a druhý absolutní
                        if operátor = '-' or operátor = '<<' or operátor = '>>'
                        then
                                //u těchto operátorů musí být rel. na levé straně
                                if <operand1>.typ_hodnoty = rel
                                then <výraz>.typ_hodnoty = rel, <výraz>.číslo_sekce =
                                        <operand1>.číslo_sekce, <výraz>.datový_typ =
                                        <operand2>.datový_typ
                                else  chyba
                                endif
                        else
                                if operátor = '+'
                                then
                                        //u operátoru + může být rel. na levé i pravé straně
                                        if <operand1>.typ_hodnoty = rel
                                        then   <výraz>.typ_hodnoty = rel, <výraz>.číslo_sekce =
                                                <operand1>.číslo_sekce, <výraz>.datový_typ =
                                                <operand2>.datový_typ
                                        else   <výraz>.typ_hodnoty = rel, <výraz>.číslo_sekce =
```

```
                                        <operand2>.číslo_sekce, <výraz>.datový_typ =
                                        <operand1>.datový_typ
                                    endif
                                else  chyba
                                endif
                            endif
                        endif
                    else
                        //oba operandy mají absolutní typ hodnoty
                        if <operand1>.datový_typ = znaménkový or <operand2>.datový_typ = znaménkový
                        then  <výraz>.typ_hodnoty = abs, <výraz>.číslo_sekce = ⊥,
                              <výraz>.datový_typ = znaménkový
                        else  <výraz>.typ_hodnoty = abs, <výraz>.číslo_sekce = ⊥,
                              <výraz>.datový_typ = neznaménkový
                        endif
                    endif
                endif
```

Ostatní pravidla

U ostatních pravidel pro vytváření výrazů se již žádné speciální operace nad atributy neprovádějí, hodnoty atributů se pouze kopírují.

<výraz> ::= <operand> :

<výraz>.*atributy* = <operand>.*atributy*

<operand> ::= <výraz> :

<operand >.*atributy* = <výraz>.*atributy*

<operand> ::= symbol :

<operand >.*atributy* = symbol.*atributy*

<operand> ::= konstanta :

<operand >.*atributy* = konstanta.*atributy*

Příklad

Mějme například výraz, který je složen z prvků, které mají uvedené hodnoty atributu *typ_hodnoty*. Řekněme, že zde *relokatibilní* prvky výrazu mají stejné číslo sekce. Ukážeme si, jak by se postupně zjistila hodnota *typu hodnoty* výsledku:

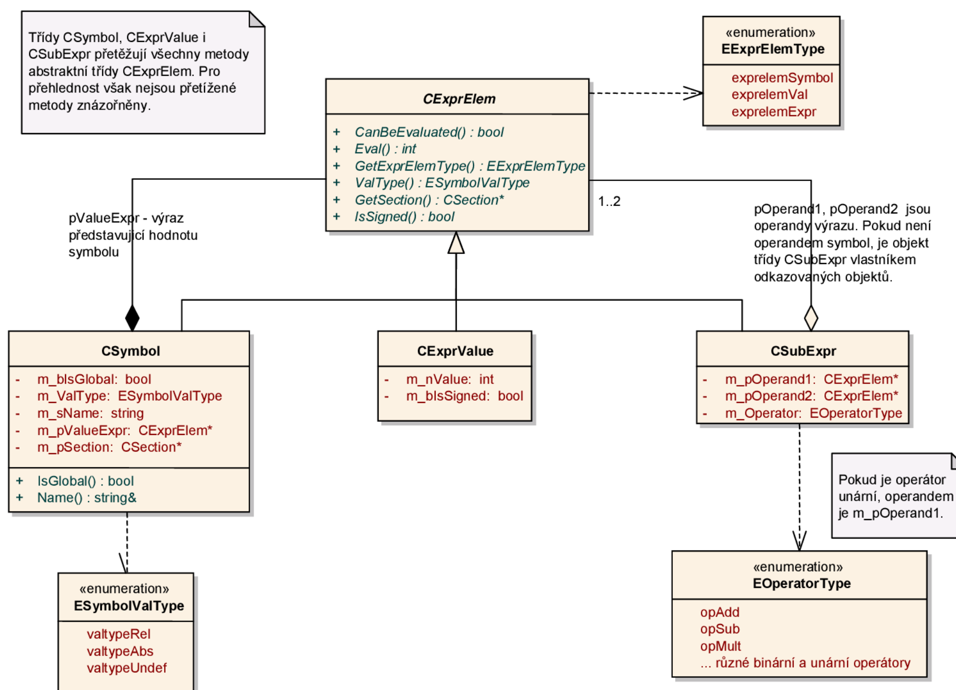
(rel – ((rel – rel) + abs)) – rel **->** *(rel – (abs + abs)) – rel* **->** *(rel – abs) – rel* **->** *rel – rel* **->** *abs*

Možná si říkáte, proč se vůbec takovýto typový systém vlastně zavádí. Je to z následujících důvodů:

- Především při překladu potřebujeme nějakým systematickým způsobem zjistit, zda je výraz platný a zda ho můžeme vypočítat.
- Často, při psaní programu pro assembler, potřebujeme zjistit velikost nadefinovaného pole dat. Ukázkou může být symbol `dvojnásobná_délka` z příkladu v předchozí kapitole 6.5.1. Z tohoto důvodu musíme povolit odčítání *relokatibilních* prvků výrazu (podobně jako u ukazatelové aritmetiky jazyka C).
- Musíme omezit také to, z čeho se výraz používající *relokatibilní* symboly může skládat. Některé výrazy nemohou být v době překladu vyhodnoceny proto potřebujeme zajistit, aby měli pouze určitou podobu, která zajistí, že linker jejich podobu může vypočítat (viz relokační položky v kapitole 3.3.4).

6.6.4 Návrh reprezentace symbolů a výrazů

Na následujícím diagramu můžete vidět návrh tříd, které budou využívány k reprezentaci výrazů a k jejich vypočítávání. jejich metodu budou implementovat typový systém popsaný v předchozí kapitole.



Obrázek 6.7: Diagram tříd reprezentující symboly a výrazy

Třída `CExprElem` je abstraktní třídou a definuje rozhraní všech prvků, ze kterých se výrazy skládají. Její tři metody `ValType`, `GetSection` a `IsSigned` umožňují přístup k hodnotám atributů *typ_hodnoty*, *číslo_sekce* a *datový_typ*. Metoda `CanBeEvaluated` zredukuje výraz na jeho minimální podobu (tj. vypočítá vypočítatelné podvýrazy), a zjistí jestli je v tomto okamžiku možné

výraz vypočítat, tzn. jestli výsledná hodnota atributu *typ_hodnoty* celého výrazu je *absolutní* či *relokatibilní*. Pomocí `Eval` pak získáme hodnotu výrazu.

Tabulka symbolů obsahuje ukazatele na všechny symboly a pseudosymboly (zástupné symbol pro výrazy), ty jsou reprezentovány třídou `CSymbol`.

6.6.5 Funkce a rozhraní jednotky pro zpracování výrazů

Jednotka pro zpracování výrazů je využívána pouze syntaktickým analyzátorem a jednotkou pro zpracování direktiv. Operace, které provádí jsou popsány v následujících odstavcích.

Nejprve si řekneme, jak je jednotka využívána pro zpracování výrazů. Od syntaktického analyzátory přijme výraz reprezentovaný stromem tokenů (viz kapitola 6.4.2). Ten přetransformuje do objektů tříd z obrázku 6.7 a pokusí se ho vypočítat. Pokud to není možné, vytvoří namísto výrazu nový pseudosymbol, přidá ho do tabulky symbolů a vrátí token s odkazem na tento pseudosymbol (ve skutečnosti s reálnými symboly a pseudosymboly pracuje stejně, jsou reprezentovány tou samou třídou viz předchozí kapitola 6.5.4). Pokud je možné výraz vypočítat, pseudosymbol nevytváří a vrátí syntaktickému analyzátoru token s vypočítanou číselnou hodnotou. Vrácený token tedy představuje výraz a tento token pak syntaktický analyzátor používá při komunikaci s kodérem instrukcí a nebo jednotkou pro zpracování direktiv. Pro symboly, jejichž názvy se ve výrazu objevily, ale nevyskytují se v tabulce symbolů vytvoří nové symboly a jejich atribut *typ_hodnoty* se nastaví jako *nedefinovaný*. Dále, pokud se výraz skládá pouze z jediného symbolu (ten se případně vytvoří a přidá do tabulky symbolů, pokud vytvořen nebyl), pseudosymbol se nevytváří a jednotka pro zpracování symbolů a výrazů vrací přímo token buď s jeho hodnotou a nebo s odkazem na tento symbol. Tímto způsobem je zajištěno, že se s výrazy, symboly a konstantami pracuje naprosto stejně.

Pro definice symbolů existují dvě možnosti. Pokud je symbol je definován jako návěští, tak syntaktický analyzátor pošle jednotce pro zpracování symbolů a výrazů jeden token s hodnotou v podobě řetězce (řetězec tokenu představuje název symbolu). Jednotka si zjistí z location counteru jeho relativní adresu a sekci, a vytvoří nový symbol se zadanými hodnotami a atributem *typ_hodnoty* se *relokatibilní*. Pokud je symbol definován pomocí direktivy .equiv, pak jednotka pro zpracování direktiv zašle dva tokeny – první představuje jméno nového symbolů a druhý je pak token s hodnotou a nebo s odkazem na symbol.

6.7 Jednotka pro ukládání dat do tabulky sekcí a bitových oprav

Tato jednotka slouží k zpracování a uložení informací vytvořených kodérem instrukcí a jednotkou pro zpracování direktiv. Je využívána pouze těmito dvěma součástmi assembleru.

Od kodéru instrukcí dostává zakódované instrukce včetně informací o bitových opravách – tzn. na která místa v zakódované instrukci se mají dosadit hodnoty symbolů (nebo pseudosymbolů představujících nějaký vyraz). Jedna informace o bitových opravách sestává ze tří položek – první dvě jsou celá čísla určující bitový offset prvního a posledního bitu v předaných datech, kde se bitová

oprava má provést a třetí položka je ukazatel na symbol do tabulky symbolů, který se při výpočtu hodnoty pro dosazení použije.

Jednotka pro ukládání dat do tabulek sekcí a bitových oprav si po přijetí zakódované instrukce a seznamu bitových oprav nejprve z location counteru zjistí aktuální adresu v sekci, pak na konec aktuální sekce přidá zakódovaná data. Dále k jednotlivým položkám seznamu bitových oprav uloží informace o sekci a adrese, kde se vyskytují a s těmito přidanými informacemi je přidá do tabulky bitových oprav.

Jednotka pro zpracování direktiv jednotku pro ukládání dat využívá pro následující čtyři operace: Vytváří nové sekce a nastavuje jejich vlastnosti, pri zpracování direktivy .org nastavuje absolutní pozici sekce, zařizuje zarovnávání na paměťové hranice zadané pomocí direktivy .align a do tabulky sekcí přidává inicializovaná a neinicializovaná data. S tabulkou bitových oprav se v tomto případě nijak nemanipuluje.

V dalších podkapitolách si popíšeme tabulku sekcí a tabulky bitových oprav.

6.7.1 Tabulka sekcí

Tabulka sekcí je seznamem sekcí a umožňuje je vyhledávat podle názvu. Každá sekce se skládá z následujících prvků:

- *název*,
- *typ* - ten může být buď *text* (kód), *data* (inicializovaná data) a nebo *bbs* (neinicializovaná data),
- *data sekce* – jde o jednoduché datové pole,
- *velikost dat*,
- informace o tom, *zda má nastavenu absolutní adresu*,
- *absolutní adresa*, ta má význam pouze pokud jí má nastavenu,
- *bitová šířka* slova v této sekci,

- číslo určující, *po kolika slovech se dají data v sekci adresovat.*
- informace, zda jsou čísla v této sekci uložena jako *little* či *big endian,*

Pozn.: Poslední dvě uvedené položky zatím v prvních fázích vývoje assembleru nebudou podporovány.

Data sekce jsou uložena jako standardní binární data a ne v textové podobě, jak je tomu u formátu objektového souboru (viz 23). Pro ladění to zde nemá velký význam (narozdíl od objektového souboru) a funkce pracující nad daty sekcí budou jednodušší a v případě přechodu na binární podobu objektového souboru je nebude potřeba upravovat. V případě, že bitová šířka slova v sekci se bude lišit od hostitelského počítače, vybere se takový počet buňek paměti, aby se slovo cílové architektury do nich vešlo. (např. pokud je bitová šířka cílové architektury 13, použijí se pro uložení na PC 16bitové úseky).

6.7.2 Tabulka bitových oprav

Tabulka bitových oprav je seznamem všech bitových oprav. Její položky vycházejí z formátů relokačních položek popsaných v kapitole 3.3.4 a obsahují následující prvky:

- *index symbolu v tabulce symbolů* – ten se využívá při výpočtu výsledné hodnoty,
- *offset prvního a posledního bitu* bitové opravy – na toto místo se dosadí vypočtená hodnota
- *adresa slova, ke kterému se vztahuje offset prvního bitu v sekci* a
- *číslo sekce*, které se tato bitová oprava týká.

6.8 Jednotka pro vyřešení bitových oprav a pro transformaci přeložených informací do objektového souboru

Doposud jsme si popisovaly součásti assembleru, které se do překladu zapojují v jeho prvním kroku. Po jeho dokončení nastupuje jednotka pro vyřešení bitových oprav a transformaci přeložených informací do objektového souboru.

V tabulkách sekcí, bitových oprav a symbolů jsou již všechny informace, které bylo možné získat ze vstupního souboru. V druhém kroku potřebujeme dosadit hodnoty budoucích symbolů a výrazů, jejichž hodnota nebyla v prvním kroku známá. Tato jednotka postupně čte informace z tabulky bitových oprav a ty, které dokáže vyřešit také vyřeší. V tabulce bitových oprav pak zůstanou pouze položky, které se odkazují na nedefinované symboly.

Poté uloží data v tabulkách sekcí do struktur knihovny *Objfilelib*, projde si zbylé položky v tabulce bitových oprav a nalezne všechny související symboly v tabulce symbolů. Na základě toho pak nastaví relokační položky a tabulku symbolů výsledného objektového souboru tak, aby linker měl všechny potřebné informace pro lokální i externí relokaci (viz kapitola 3.3.4).

Třetí poslední krok překladu pak spočívá už pouze v uložení vytvořeného objektového souboru na pevný disk.

6.9 Zhodnocení návrhu assembleru

Assembler vykonává několik různých operací jako jsou především kódování instrukcí, zpracování direktiv a generování objektového souboru. Ty se podařilo v návrhu rozdělit na samostatné součásti s jednoduchým jednoúčelovým rozhraním. Zde v několika bodech shrnu základní prncipy návrhu:

- Preprocesor assembleru nám umožní definovat odlišnou syntaxi jazyka assembleru (viz 6.1) a zavést podporu podmíněného překladu a maker tak, aby assembler měl schopnosti moderních makroassemblerů. Vývoj preprocesoru může probíhat nezávisle na samotném assembleru.

- Oddělením překladu jazyka instrukcí a překladu jazyka assembleru jsme dosáhli toho, že samotný překlad instrukcí je na assembleru nezávislý. Jediné, co assembler potřebuje znát je gramatika jazyka instrukcí (bez jakýchkoli dalších informací), a dokáže pak odlišit, co má zpracovat sám a co má poslat k překladu kodéru instrukcí (5.3.3 a 6.4).

- Rozhraní mezi syntaktickým analyzátorem a kodérem instrukcí, jednotkou pro zpracování direktiv a jednotkou pro zpracování symbolů a výrazů je velice jednoduché a spočívá pouze v zasílání *tokenů* (6.4.2, 6.5 a 6.6.5).

- Systematickým způsobem je zavedena podpora výrazů a definován typový systém pro operace nad výrazy. Ty pak slouží k tomu, aby bylo možné rozhodnout, zda se výraz dá vypočítat, zda je správně vytvořen a jakého typu je jeho výsledek (6.6.2 a 6.6.3).

- Assembler je, podobně jako GNU assembler, částečně nezávislý na formátu objektového souboru. V případě jeho změny stačí změnit pouze jednu součást a to jednotku pro vyřešení bitových oprav a transformaci přeložených informací do objektového souboru.

- Postup při překladu je rozdělen na tři samostatné kroky a v jednotlivých krocích se používají jiné funkční součásti assembleru.

7 Závěr

V rámci mé diplomové práce jsem se nejprve seznámil s různými jazyky pro popis architektury mikroprocesorů jako jsou LISA, nMl, MIMOLA a částečně i dalšími. Poté jsem studoval architektury instrukčních sad různých mikroprocesorů. Také jsem se naučil nové věci o architekturách procesorů. To vše s cílem udělat assembler co nejkvalitnější. Zkoumal jsem různé obecné assemblery a především GNU assembler. Ukázalo se, že vytvořit opravdu použitelný obecný assembler, který by bylo možné dále vyvíjet a rozšiřovat, je náročný úkol, který spojuje obory překladačů, architektur počítačů a také je potřeba perfektně znát funkci linkerů. Práce navazuje na můj ročníkový projekt a bakalářskou práci Libora Vašíčka.

S ohledem na to, že bych rád na práci v projektu Lissom dále pokračoval, jsem assembler detailně navrhnul především s plánem do budoucna. Myslím, že návrh je zdařilý a odděluje jednotlivé součásti assembleru jako je preprocesor, pevná část assembleru a kodér instrukcí, ty se mohou vyvíjet nezávisle na sobě. Preprocesor nám umožní konfigurovat vstupní formát zdrojového souboru assembleru a také zavést podporu schopností moderních makroassemblerů. Pevná část je pak jádrem assembleru, který přijímá výstup preprocesoru, řídí překlad, zpracování direktiv a výrazů a vytváří objektový soubor. Pevná část je dále rozdělena na několik jednoúčelových součástí, které mají pouze jednoduché rozhraní. Dále, kodér instrukcí je využíván pevnou částí assembleru k překladu mnemotechnického zápisu instrukcí do jejich binární podoby. Dále bylo navrženo zpracování výrazů, ty se v assemblerech zpracovávají poněkud neklasickým způsobem z důvodu, že symboly nemusí být definovány nebo jejich hodnota může být relativní k začátku sekce a pod.. To, jak lze výrazy tvořit a vypočítávat je formálně popsáno pomocí typového systému. Vývoj assembleru půjde rozdělit do několika fází, kde výsledkem každé fáze bude použitelný assembler. Tyto fáze jsou popsány v příloze C.

Dále jsem implementoval část projektu Lissom, která je využívána jak assemblerem tak i jinými nástroji či jejich generátory. Jedná se o část operací knihovny Parserlib2, která čte informace o operacích z vnitřního modelu jazyka ISAC a na základě popisu vytváří instance tříd, které jsou pak snáze využitelnější v programech než samotný vnitřní model. Také jsem přehledným způsobem graficky popsal aktuální formát vnitřního modelu, tento popis naleznete v příloze F.

Projekt Lissom mě velice zaujal, ostatně jako i další členy týmu. Věřím, že se nám společně podaří vytvořit v praxi využitelné vývojové prostředí, které výrazně zefektivní proces vývoje nových aplikačně specifických procesorů a rád bych na projektu pracoval i během mého doktorského studia.

Komentář [th1]: takto se odkazy nedělají

Literatura

[1] Hoffmann, A., Meyr, H., Leupers, R.: *Architecture Exploration for Embedded Processors with LISA*. Kluwer Academic Publishers, 2002

[2] Knoll, A.: *The NML Machine Description Formalism*. TU Berlin 1991. Dostupné z WWW: <http://atknoll1.informatik.tu-muenchen.de:8080/tum6/publications/documents/Index/machine/list_topic_documents?f_typ=bereich>

[3] Fauth, A., Praet, J., Freericks, M.: *Describing Instruction Set Processors Using nML*. TU Berlin, 1995. Dostupné z WWW: <http://citeseer.ist.psu.edu/fauth95describing.html>

[4] Qin, W., Malik, S.: *Architecture Description Languages for Retargetable Compilation*. CRC Press, 2002. Dostupné z WWW: <http://www.gigascale.org/pubs/199.html>

[5] Masařík, K.: *Jazyky pro popis architektury počítačových systémů*., FIT VUT, 2006

[6] Fisher, J. A. a kol.: *Embedded Computing: A VLIW Approach to Architecture, Compilers and Tools*, Morgan Kaufmann, 2004

[7] Hruška, T. a kol.: Jazyk ISAC – Příručka, FIT VUT 2007

[8] Lukáš, R., a kol.: *Two-Way Deterministic Translation and Its Usage in Practice.*. FIT VUT 2006

[9] Masařík, K.: *Obohacení popisného jazyka o nové konstrukce*. FIT VUT 2006

[10] LISATek: *LISA 2.0 Language Reference Manual*. LISATek 2001

[11] Patterson, D., Hennesy, J.: *Computer Organization and Design: The Hardware/Software Interface*. Morgan Kaufmann, 2004

[12] Patterson, D., Hennesy, J.: *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 2007

[13] Abbaspor, M., Zhu, J.: *Retargetable Binary Utilities*. University of Toronto, 2002. Dostupné z WWW: <http://citeseer.ist.psu.edu/abbaspour02retargetable.html>

[14] Salomon, D.: *Assemblers and Loaders*. Ellis Hortwood, 1993. Dostupné z WWW <http://www.davidsalomon.name/assem.advertis/AssemAd.html>

[15] Levine, R.: *Linkers and Loaders*. Morgan Kaufmann Publishers, 2000

[16] Chiu, P. P. K., Fu, S. T. K.: *A Generative Approach to Universal Cross Assembler Design*, Hong Kong Polytechnic, 1990. Dostupné z WWW: <http://doi.acm.org/10.1145/74105.74111>

[17] Abbaspor, M., Zhu, J.: *Automatic Porting of Binary File Descriptor Library.* University of Toronto, 2001. Dostupné z WWW: <http://www.eecg.toronto.edu/~jzhu/publications/doc/tr-09-01.pdf>

[18] Baldassin, A. a kol.: *Extending the ArchC language for Automatic Generation of Assemblers.* Institute of Computing, University of Campinas, 2005. Dostupné z WWW: <http://www.lsc.ic.unicamp.br/publications/>

[19] Taglietti, L. a kol.: *Automatic ADL-Based Assembler Generation for ASIP Programming Support.* Federal University of Santa Catarina, Brazil, 2005. Dostupné z WWW: <http://www.springerlink.com/index/10WBTCV1079DM183.pdf>

[20] *Manual Reference Pages - A.OUT (5).* FreeBSD Man Pages, 1993. Dostupné z WWW: <http://www.gsp.com/cgi-bin/man.cgi?section=5&topic=a.out>

[21] *Microsoft Portable Executable and Common Object File Format Specification*, Microsoft, 2006. Dostupné z WWW: <http://www.microsoft.com/whdc/system/platform/firmware/PECOFF.mspx>

[22] *LIB BFD, the Binary File Descriptor Library.* Free Software Foundation, 1999, Dostupné z WWW: <http://www.gnu.org/software/binutils/manual/bfd-2.9.1/>

[23] Kolář, D.: *Návrh výstupního formátu pro assembler a linker, verze. 0.00.1.* FIT VUT, 2004.

[24] Shann,R., Chepstow M.: *Assembling an Object Code Module.* ST Microelectonics Limited, 2001. Dostupné z WWW: <http://www.freepatentsonline.com/6901584.html>

[25] *Assembler Internals*, Free Software Foundation,2006. Dostupné z WWW: <http://www.gnu.org/software/binutils/>

[26] Bartlett, J.: *Programming from the Ground Up*. Bartlett Publishing, 2003. Dostupné z WWW: <download.savannah.nongnu.org/releases/ pgubook/ProgrammingGroundUp-1-0-booksize.pdf>

[27] *Using as*. Free Software Foundation, 2006. Dostupné z WWW: <http://www.gnu.org/software/binutils/manual/gas-2.9.1/html_chapter/as_toc.html>

[28] Hruška, T.: *Vnitřní model ISAC 0.1.* FIT VUT, 2006.

[29] Lukáš, R.: *ISAC 0.0:Vnitřní model operační části*. FIT VUT, 2004.

[30] Husár, A.: *Obecný assembler pro mikroprocesory*. Ročníkový projekt, FIT VUT, 2005.

[31] Vašíček, L.: *Návrh struktury obecného assembleru a zpětného assembleru*, Bakalářská práce, FIT VUT, 2006.

[32] Lukáš, R: *Dvojcestné párové automaty.* FIT VUT, 2006.

[33] Lukáš, R.: *Implementace dvojcestných párových automatů.* FIT VUT, 2007.

[34] Lukáš. R., a kol.: *Two-Way Coupled Finite Automata*, FIT VUT, 2006.

[35] Husár, A.: *Implementace obecného assembleru*. Semestrální projekt, FIT VUT, 2007.

[36] Masařík, K.: *Parserlib – verze 2*. FIT VUT, 2007.

[37] Carlson, D.: *Modelling XML applications with UML*, Addison-Wesley, 2001.

[38] *The GNU Assembler*. Texas Instruments, 2003. Dostupné z WWW: <http://tigcc.ticalc.org/doc/gnuasm.html>

Příloha A
Knihovna Parserlib2, třídy pro uložení informací o operacích a skupinách

Tato kapitola obsahuje popis tříd knihovny Parserlib2 a samotného procesu ukládání informací do těchto tříd pomocí XML parseru. Na základě informací z tříd knihovny Parserlib2 se bude generovat kodér instrukcí, lexikální a syntaktický analyzátor assembleru. Knihovna je využívána i dalších částech projektu Lissom. Jedná se o hlavní implementační část řešení.

Popis v této kapitole je určen především těm, kteří budou s knihovnou Parselib2 pracovat a i z tohoto důvodu je poněkud obsáhlá. Pokoušel jsem se uvést všechny informace potřebné pro pochopení toho, jakým způsobem se s třídami obsahujícími informace o operacích a skupinách pracuje. Pro čtenáře, který nepotřebuje pracovat s Parserlib2, bude kapitola nejspíše nezajímavá a může jí přeskočit.

A.1 Nová verze Parserlib2

Knihovna Parserlib je součástí projektu, která se stará o načtení dat z vnitřního XML modelu do struktur (nebo spíše tříd) této knihovny. Data uložená v třídách Parserlib2 pak využívají další nástroje generující assembler, disassembler, simulátor, a další.

- Třídy knihovny Parserlib slouží k uložení informací z původního zdrojového souboru v jazyce ISAC (resp. vnitřního modelu) k dalšímu zpracování.

V následujícím textu se budu zabývat především částí operací a skupin. V části HW zdrojů nedošlo oproti minulé verzi jazyka ISAC 0.0 (do roku 2005) k příliš velkým změnám. Část jazyka ISAC, stejně jako část vnitřního modelu, popisující operace a skupiny budu nazývat *část operací*.

A.1.1 Požadavky kladené na třídy Parserlib2

- Korespondence s konstrukcemi jazyka ISAC
- Možnost srozumitelně uložit všechny informace
- Pokud možno, jednoduchost a jednotnost
- Jednotné zacházení s instancemi operací, skupin a odkazy na zdrojové prvky
- Rychlé zjištění chyby při špatném použití (například při pokusu přistupovat k volitelně zadané informaci, která však zadána nebyla)

A.1.2 Důvody pro novou verzi

Předchozí verze Parserlib pro část operací byla původně navržena čistě pro uložení informací o instrukční sadě tak, aby představovaly párovou bezkontextovou gramatiku. Této podoby dat pak využívaly generátory assembleru a disassembleru pro generování překladové gramatiky. Na tomto principu překladu fungoval assembler verze 0.1 a 0.2.

Postupným přidáváním nových sekcí však navržená architektura přestala vyhovovat a informace o sekcích ACTIVATION, CODINGROOT a SWITCH byly uloženy jinak a nejednotně s původními třídami pro ASSEMBLER a CODING z důvodu jejich přílišné specializovanosti. Proto jsme se rozhodli vytvořit novou verzi knihovny Parserlib2, která by se pokusila splnit požadavky uvedené v kapitole A.1.1.

Jaké části se mění a které zůstávají stejné:

Naformátováno: Odrážky a číslování

- Část pro uložení informací o zdrojích zůstává v podstatě stejná bez nějakých výrazných změn.
- Část operací a skupin se kompletně nahrazuje.

Návrh nové architektury architekturu pro část operací a její popis můžete nalézt v dokumentu Parserlib – verze 2 [36].

Naformátováno: Odrážky a číslování

A.2 Vnitřní model jazyka ISAC

A.2.1 Účel vnitřního modelu

Vnitřní model byl navržen jako výstupní formát překladače jazyka ISAC. Slouží jako předzpracovaný zdroj informací pro generující nástroje. Zde je uvedeno několik důvodů proč byl zaveden, jakými zásadami se jeho formát řídí a k čemu souží:

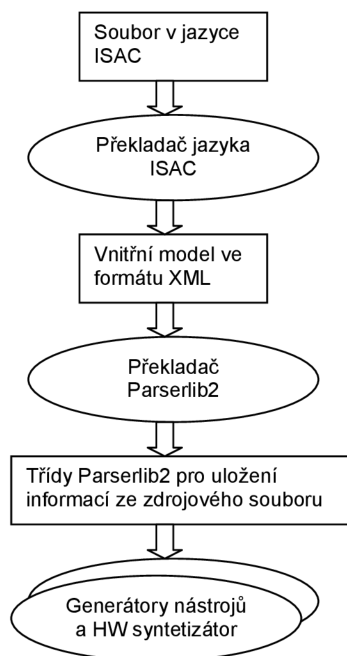
- Generující nástroje mohou předpokládat, že data ve vnitřním modelu jsou již jak syntakticky tak i částečně sémanticky zkontrolovány překladačem jazyka ISAC.

- Překladač může předem provést některé transformace jako seřazení definic různých symbolů a zaručí, že využívaný symbol byl již v předchozím textu definován (to je například jeden z požadavků algoritmu překladu pomocí dvojcestných párových automatů).

- Je ve formátu XML, s tím, že se pouze využívají elementy a jejich textové data. Atributy elementů nejsou využívány.

- Původně byl zamýšlen jako zdroj dat pro generátory nástrojů, nyní je využíván pouze knihovnou Parserlib2, která data načte, částečně je upraví (například propojí místa, jde jsou některé identifikátory využívány a jejich definice) a poté je poskytne generátorům nástrojů.

Na následujícím obrázku můžete vidět tok informací ze zdrojového souboru v jazyce ISAC k generátorům nástrojů.



Obrázek A.1: Tok informací ze zdrojového souboru v jazyce ISAC ke generátorům nástrojů

A.2.2 Současný stav formátu vnitřního modelu, popis jeho diagramu

Naformátováno: Odrážky a číslování

V přílohách G.1 a G.2 je znázorněn současný vnitřní formát (duben 2007). K popisu jsem použil UML notaci podobnou diagramu tříd. Při popisu jsem se snažil o co největší přesnost. Také jsou částečně ukázány vztahy mezi jednotlivými textovými daty. Také vlastnost, že element může obsahovat pouze jeden z možných dětských elementů je zde zobrazena, avšak pouze pomocí poznámky.

Nyní si popíšeme si grafickou notaci, která byla pro znázornění formátu vnitřního modelu použita.

Mějme například element GROUP, který popisuje skupiny operací jazyka ISAC.

Popis v jazyce ISAC:

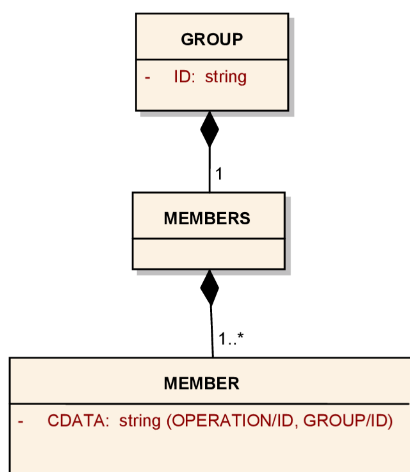
```
GROUP group = C,D,E;
```

Vnitřní model:

```
<GROUP>
   <ID>group</ID>
   <MEMBERS>
      <MEMBER>C</MEMBER>
   <MEMBER>D</MEMBER>
```

```
    <MEMBER>E</MEMBER>
    </MEMBERS>
  </GROUP>
```

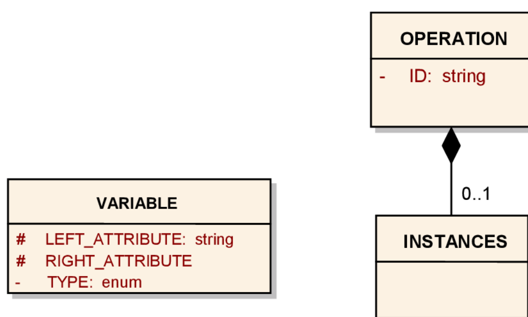
Formát vnitřního modelu konstrukce GROUP popsaný pomocí diagramu tříd UML:



Obrázek A.2: Popis elementu GROUP pomocí diagramu tříd UML

Element GROUP obsahuje (z objektového pohledu jde o kompozici) jak element MEMBERS tak i ID. ID je jednoduchý element, který může obsahovat pouze data – to je naznačeno datovým typem *string*. Element MEMBERS pak obsahuje alespoň 1 element MEMBER a ten už přímo obsahuje data. MEMBER obsahuje řetězec, který by měl být stejný jako nějaký jiný řetězec elementu OPERATION/ID nebo GROUP/ID. Tímto způsobem jsou naznačeny relace mezi textovými daty.

Dále je možné, že některý jednoduchý dětský element nemusí být zadán. Například element VARIABLE, který nese informace o atributu v sekci ASSEMBLER.



Obrázek A.3: Volitelný výskyt dětských elementů

Pokud je některý dětský element znázorněn jako *protected* (#), znamená to, že jeho výskyt je volitelný. Druhou možností naznačení takovéhoto vztahu, je uvést násobnost kompozice jako *0..1*.

A.2.3 Zhodnocení současného formátu vnitřního modelu, návrhy pro vylepšení

Formát vnitřního modelu se vyvíjel postupně a byly do něj přidávány další typy elementů pro nové konstrukce jazyka ISAC. Z tohoto důvodu je poněkud nekonzistentní, podobné věci jsou popisovány jinak a objevují se v něm stejné názvy elementů pro různé konstrukce jazyka ISAC s různým významem (CODE, INSTANCE a EXPRESSION).

Návrhy pro vylepšení:

- Jednou z věcí, které bude potřeba sjednotit, jsou odkazy na instance operací, (OPERATION/INSTANCES), operace, skupiny a zdrojové prvky. Jejich formát je teď víceméně náhodný. Také bude potřeba přesně popsat, jaký název zdrojového prvku, operace či jiného symbolu se může kde vyskytnout.

- V textových datech se mohou vyskytnout znaky jako operátor menší „<". Tyto znaky by mohly zmást XML parser. Těmto případným problémům se bude potřeba vyhnout použitím standardních elementů XML pro náhradu speciálních znaků. V souvislosti s tím by bylo možná výhodné aby i překladač jazyka generující XML soubor využíval XML parser pro jeho generování a nechat tedy práci se zajištěním správného formátu na parseru.

- Používat pro různé konstrukce různé názvy elementů.

- Dále by bylo vhodné přesně popsat strukturu vnitřního formátu buď pomocí XML Schema a nebo RelaxNG.

Popis formátu vnitřního modelu pomocí UML nebo XML Schema

Například v knize [37] se dá nalézt postup, jak UML popis transformovat do popisu v XML Schema. Dále, modelovací program Rational Rose dokáže z UML diagramu automaticky vygenerovat XML Schema. Také existují nástroje pro automatické generování kódu XML překladače a tříd pro uložení získaných informací na základě XML Schema:

- pro Javu: Stylus Studio (www.stylusstudio.com) a
- pro C++: CodeSynthesis XSD (http://www.codesynthesis.com/products/xsd/).

Oba nástroje jsou poskytovány zdarma. Generátor překladače pro C++ jsem trochu zkoušel a vypadá docela použitelně, avšak před skutečným použitím je potřeba dostatečně ověřit, zda vyhovuje našim potřebám.

- Hlavní výhodou při použití automaticky generovaných překladačů by byla možnost jednoduchých a rychlých úprav formátu vnitřního modelu a možnost, na základě jednoho mnohem abstraktnějšího popisu, generovat jak XML parser pro Javu tak pro C++. (C++ potřebujeme pro generátory nástrojů, Javu pro vývojové prostředí – například pro zvýrazňovač syntaxe.)

- Další věcí, se kterou by použití XML Schema snad mohlo pomoci, by byl systematičtější vývoj formátu vnitřního modelu.

A.3 Architektura tříd pro uložení informací o operacích a skupinách

A.3.1 Principy

Zde je vypsáno několik principů, kterými se řídí architektura tříd operací Parserlib2 (viz také Požadavky kladené na třídy Parserlib2 v podkapitole 7.1.1:

- Podobnost s konstrukcemi jazyka ISAC
- Malý počet tříd (vzhledem k tomu, kolik různorodých informací potřebujeme uložit)
- Nevyužívá se dědění tříd, namísto toho obecnější třídy obsahují ukazatel(e) na objekty tříd specializovanějších (např. viz CElem). Dědění by do návrhu spíše zavedlo další složitost a příliš by nepomohlo.
- Prvky tříd jsou veřejné. Pouze pro kontrolu, zda nejsou využívány chybně, jsou obaleny speciálními šablonami, které správné využití kontrolují (viz dále v kapitole A.3.3). Toto „obalení“ ale nemění způsob, se kterým se k prvkům přistupuje.

- A jeden z hlavních principů: „Všechno, co je obsahem sekcí, je element.“. Jak textový řetězec, jakákoli instance, atribut sekce ASSEMBLER a nebo CODING a nebo řídicí struktura (SWITCH) jsou elementy.

Naformátováno: Odrážky a číslování

A.3.2 Architektura tříd

V této kapitole si popíšeme architekturu tříd operací a to, jakým způsobem se instance t2chto tříd navzájem spojují.

Na následující obrázku 7.4 můžete vidět zjednodušený diagram tříd, které slouží pro uložení informací o operacích a skupinách. Některé nepříliš důležité třídy byly vynechány. V příloze H.1 pak můžete najít kompletní diagram.

Několik poznámek, které by se mohli hodit pro pochopení architektury:

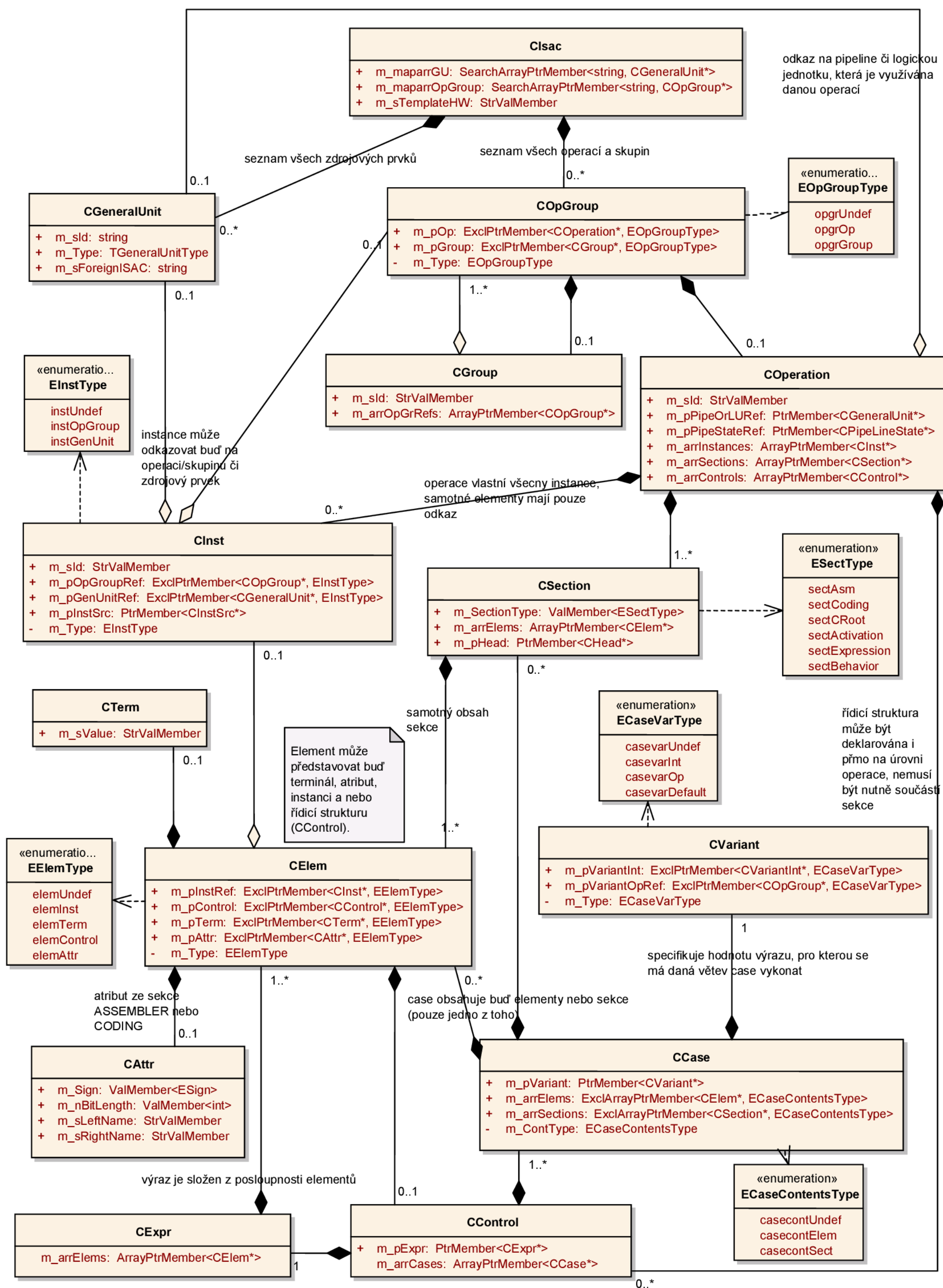
- Jedna třída odpovídá jednomu typu konstrukce v jazyce ISAC

- V podstatě každá třída obsahuje ukazatele na třídy jiné. Pomocí těchto ukazatelů se mezi nimi definují relace. Podobně jako u vnitřního modelu si můžeme konkrétní stav (vytvořené objekty tříd s nastavenými hodnotami a ukazateli) představit jako orientovaný graf. U vnitřního modelu jde dokonce o strom, ale zde máme navíc vyřešené relace, které byly u vnitřního modelu definovány pouze textově (viz příloha F.1).

  - Relace, které byly dané ze struktury vnitřního modelu, jsou modelovány jako kompozice.
  - Relace, které byly vyřešeny pomocí parseru vnitřního modelu (pomocí Parserlib2), jsou naznačeny jako agregace.

- Pozn.: Hledejte spíše podobnost s konstrukcemi jazyka ISAC než s formátem vnitřního modelu. Třídy byly navrženy tak, aby se blížili více k jazyku ISAC. Vnitřní model se bude nejspíše měnit a nebyl na něj brán příliš velký ohled.

- Pozn. 2: Pokud jste se již podívali na diagram třídna obrázku A.4 a vrtá vám hlavou, co jsou to ty šablony ValMember, Ptrmember, SearchArrayPtrMember a pod., vysvětlení naleznete v navazující kapitole A.3.3. Tyto šablony definují určité chování prvků tříd – kdy lze jejich hodnoty zapisovat a číst. Toto pro nás teď ale není důležité a namísto šablon *obalujících* datové typy prvků si můžete představit jednoduché datové typy. U těch šablon, které obsahují v názvu *Array* jde o pole proměnných daného typu. Příklad: Třída COpGroup má prvek m_pOp typu ExclPtrMember<COperation*, EOpGroupType>. Když zanedbáme šalonu, m_pOp je ukazatel na COperation.



Obrázek A.4: Zjednodušený diagram tříd operací

A.3.3 Rozhraní tříd operací

V předchozí kapitole jsem si ukázali jak spolu souvisí jednotlivé třídy operací. Zde se zaměříme především na to, jakým způsobem funguje rozhraní těchto tříd.

Metody rozhraní byly navrženy tak, aby v co největší míře pomohlo se vyhnout se chybám při používání. Jeden z důvodů je, že architektura tříd je navržena velmi flexibilně, ale při špatném použití by mohla být náchylná k chybám. Avšak nabízí zase jednoduchost, díky malému počtu potřebných tříd obsahujících informace z vnitřního XML modelu. Například třída CElem (ELEMENT) může obsahovat odkaz buď na třídu CTerminal, CInstance, Control a na CGeneralUnit. Abychom například nepracovali se špatným z těchto ukazatelů, dovolí nám získat pouze ten správný.

Zde je vypsáno několik principů, jimiž se rozhraní tříd řídí. Detailněji jsou popsány dále.

- Vytváření instancí tříd a nastavení jejich hodnot se provádí pouze jednou (pomocí parseru vnitřního XML formátu). Po vytvoření není možné relace mezi vytvořenými objekty a hodnoty jejich členů měnit.

- Prvky tříd jsou veřejné, avšak díky šablonám ValMember, PtrMember, ArrayValMember a dalším (viz dále) není umožněno získat hodnotu prvku, který nebyl zadán. U polí je použita kontrola platného indexu.

- Dalším pravidlem je to, že členské proměnné, jež jsou ukazatelé na jisté objekty, ale nejsou jejich vlastníky, mají příponu „Ref“. Pro toto relaci používám slovo reference, v UML se značí agregací. (Reference a nebo spíše odkaz zde má odlišný význam než v jazyce C++.)

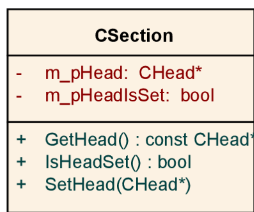
- U některých tříd je možné, aby pouze jedna z členských proměnných nesla platnou hodnotu (např. třída CElem, viz dále). Pokud již byla jedna z těchto „exkluzivních“ proměnných nastavena, není možné nastavit jinou.

Zábrany proti nepovolenému použití jsou implementovány jak konstrukcemi jazyka C++, tak pomocí assertů.

V jazyce ISAC (a tím pádem i ve vnitřním modelu) je spousta různých volitelných typů informací. Příkladem může být informace o atributu v sekci ASSEMBLER nebo CODING. Ten má dva volitelné názvy, LEFT_ATTRIBUTE a RIGHT_ATTRIBUTE. Dalším příkladem volitelného prvku může být *hlavička sekce*.

```
OPERATION operace_A IN alu{
   BEHAVIOR ( [ R, mem ] [ R ] ){
         A_ALU( R, mem);
       };
};
```

Informaci o tom, zda byla informace vůbec zadána, je nutné někam uložit. Jednou možností, jak automaticky zařídit, že se při nastavení hodnoty nastaví i to, že tato hodnota byla zadána by mohla být trojice metod IsSet/Get/Set. Na následujícím obrázku můžete vidět příklad možné třídy CSection. Ostatní prvky, které by CSection měla obsahovat, jsou pro názornost vynechány.

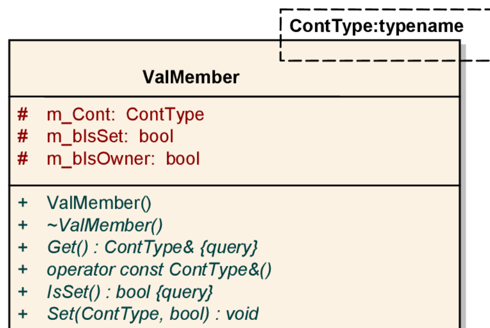


Obrázek 7.5: Hypotetická podoba třídy CSection, která může či nemusí obsahovat hlavičku (CHead)

Pozn.: Příklad byl vybrán kvůli jeho jednoduchosti. Možná vás napadlo, že ukazatel může být i NULL a podle toho pak poznáme, jestli je nebo není nastaven – u celého čísla nebo u řetězce (std::string) to však již tak jednoduché není.

Některé třídy mají takovýchto prvků jako je pHead více a počet jejich metod IsSet/Get/Set by mohl být velký. Z tohoto důvodu jsem zavedl šablony, které „obalují" prvky metod a poskytují požadované chování.

První z obalujících šablon je šablona ValMember. Parametrem šablony ContType je datový typ prvku třídy. Na následujícím obrázku ji můžete vidět.



Obrázek A.6: Šablona ValMember

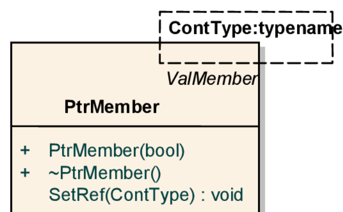
Metody IsSet/Get/Set mají obvyklý význam. Dále definuje operátor pro přetypování na konstantní referenci typu ContType, který nám umožní s obalenou proměnnou pracovat stejně jako kdyby obalená nebyla (pro čtení její hodnoty).

Popis jednotlivých členských proměnných:

- m_Cont – Obsažená (obalená) proměnná

- m_bIsSet – Booleovská proměnná určující, zda byla hodnota nastavena, inicializována na "false"
- m_bIsOwner – V případě, že je obsažen ukazatel, určuje, zda je instance ValMember vlastníkem objektu, na který ukazatel ukazuje. Tzn., že se při zavolání destruktoru odkazovaný objekt uvolní. Pro samotnou šablonu ValMember není tato hodnota důležitá, je využívána zděděnými třídami PtrMember a ExclPtrMember.

Šablona ValMember obaluje proměnnou držící nějakou hodnotu. Pro ukazatel není vhodná, protože z důvodu typové kontroly by se nedalo v C++ čistě vyřešit, aby se v destruktoru použil na proměnnou m_Cont operátor delete. K uložení ukazatelů slouží zděděná třída PtrMember. Ta dědí všechny metody z ValMember a přidává jednu novou metodu SetRef.



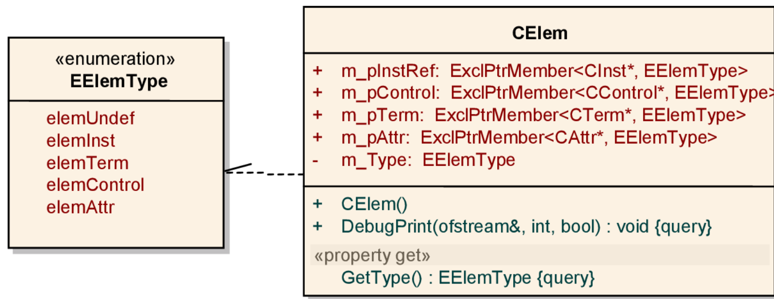
Obrázek A.8: Šablona PtrMember

Dále také definuje jediný konstruktor, který jako parametr bere inicializační hodnotu pro m_bIsOwner.

Rozdíl mezi Set a SetRef souvisí s vlastnictvím objektu na který uložený ukazatel ukazuje. Set slouží k nastavení ukazatele a přebírá vlastnictví odkazového objektu. SetRef si pouze hodnotu ukazatele uloží, ale v destruktoru odkazovaný objekt neruší.

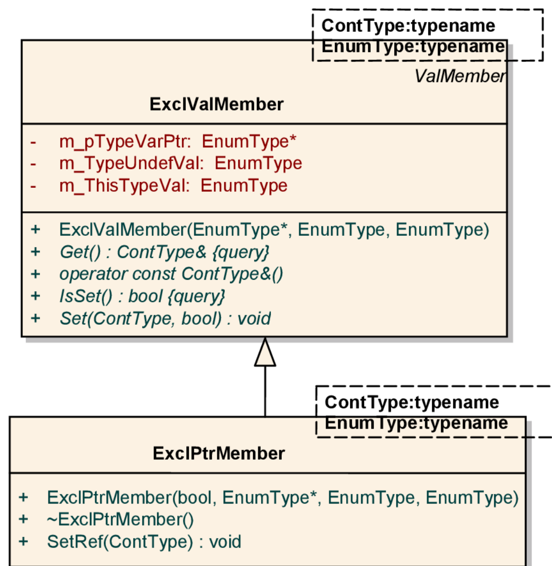
Nyní se vrátíme zpět k hodnotě proměnné m_bIsOwner. Ta se po vytvoření objektu již nedá změnit. Pomocí assertů je pak kontrolováno, zda voláme správnou funkci pro nastavení – pro metodu Set musí být m_bIsOwner „true" a pro SetRef „false". Tímto způsobem je kontrolováno, že opravdu voláme funkci, kterou jsme chtěli a nesletli jsme se.

Další šablonou zděděnou od ValMember je ExclValMember a ExclPtrMember – členská proměnná držící exkluzivní hodnotu/ukazatel. Pro vysvětlení se podíváme na třídu CElem:



Obrázek A.9: Třída CElem

Třída CElem obsahuje čtyři ukazatele. Každý na jiného typu – CInst*, CControl*, CTerm* a CAttr*. Pouze jeden (exkluzivní) ukazatel z těchto čtyř může držet platnou hodnotu (element může být pouze jednoho typu). To, že nebudeme pracovat s neplatným (nenastaveným) ukazatelem nám zařídí šablona PtrMember. Avšak pro kontrolu, že můžeme nastavit pouze jeden z těchto ukazatelů nám její schopnosti nevystačí. Budeme potřebovat nějakou sdílenou proměnnou, která nám řekne, který z ukazatelů drží exkluzivní hodnotu. Ta je prvkem rodičovské třídy – zde CElem::m_Type. Nyní si ukážeme šablonu ExclValMember a od ní zděděnou ExclPtrMember.



Obrázek A.10: Šablony ExclValMember a ExclPtrMember

Šablona ExclValMember má následující prvky, jejich hodnoty jsou nastaveny v konstruktoru šablony.

- m_pTypeValPtr – Ukazatel na sdílenou proměnnou obsahující aktuální typ exkluzivní proměnné – tedy to, který z ukazatelů drží platnou hodnotu. V případě třídy CElem je v jejím konstruktoru nastaven na adresu proměnné CElem::m_Type.
- m_TypeUndefVal – Členská proměnná v konstruktoru inicializována na výčtovou hodnotu označující nedefinovaný typ exkluzivní proměnné. U prvků CElem je nastavena vždy na elemUndef.
- m_ThisTypeVal - Členská proměnná v konstruktoru inicializována na výčtovou hodnotu označující tento typ exkluzivní proměnné (u CElem::m_pInstRef jde o elemInst, u m_pControl o elemControl a pod.).

Nyní ještě zbývá vysvětlit, k čemu nám tyto prvky jsou. To si ukážeme tělech metod Set a Get.

```
/**
 * Returns constant reference to contained item.
 * Asserts if item is not set - when m_bIsSet is false.
 * Sets m_bIsSet to true.
 * Asserts when parent class type variable (*m_pTypeVarPtr)
 * value differs from this variable type (m_ThisTypeValue). That
 * means, that exclusive value was not set or was set but another
 * parent class class member was set
 * before.
 */
virtual const ContType& Get() const
{
	ASSERT(*m_pTypeVarPtr == m_ThisTypeVal);
	return ValMember<ContType>::Get();
}


/**
 * Sets item value for this class member object.
 * Asserts if value was already set. (if m_bIsSet is true)
 * Asserts also if another exclusive class member value of same type
 * was already set before.
 */
virtual void Set(ContType val, const bool bTestOwnerhip = true)
{
	ASSERT(*m_pTypeVarPtr == m_TypeUndefVal);

	//set current type
	*m_pTypeVarPtr = m_ThisTypeVal;

	ValMember<ContType>::Set(val, bTestOwnerhip);
}
```

Poslední šablonou je ExclPtrMember. Ta vychází z ExclValMember a rozšiřuje její chování o případné uvolnění objektu na který obalený ukazatel ukazuje (naprosto stejně, jako PtrMember rozšiřuje chování ValMember).

V přechozím textu této kapitoly jsme si popsali šablony ValMember, PtrMember, ExclValMember a ExclPtrMember. Ty nám, slouží k reprezentaci jednoho prvku třídy s tím, že od tohoto prvku ještě vyžadujeme určité chování. Ve třídách operací potřebujeme kromě prvků držících jednotlivé hodnoty či ukazatele i prvky představující pole hodnot/ukazatelů. K tomu slouží šablony

ArrayValMember, ArrayPtrMember, ExclArrayValMember, ExclArrayValMember a SearchArrayMember. V principu jsou stejné jako šablony pro jednotlivé hodnoty a nejsou zde detailněji popsány.

Kompletní diagram tříd těchto šablon naleznete v příloze G.2.

Naformátováno: Odrážky a číslování

A.4 Architektura parseru vnitřního modelu pro operace a skupiny

V této kapitole si popíšeme, jakým způsobem funguje parser, který načítá data z vnitřního modelu, vytváří objekty tříd operací (dále jen objekty operací), nastavuje hodnoty jejich prvků a také nastavuje relace mezi nimi.

Pokud se podíváte na příklad vytvořených tříd v příloze G.3 a odmyslíte si všechny *nahoru vedoucí* relace typu agregace, tak si vytvořené třídy můžete představit jako strom. Podobně, informace ve vnitřním modelu (resp. jednotlivé elementy), už z podstaty formátu XML, tvoří strom.

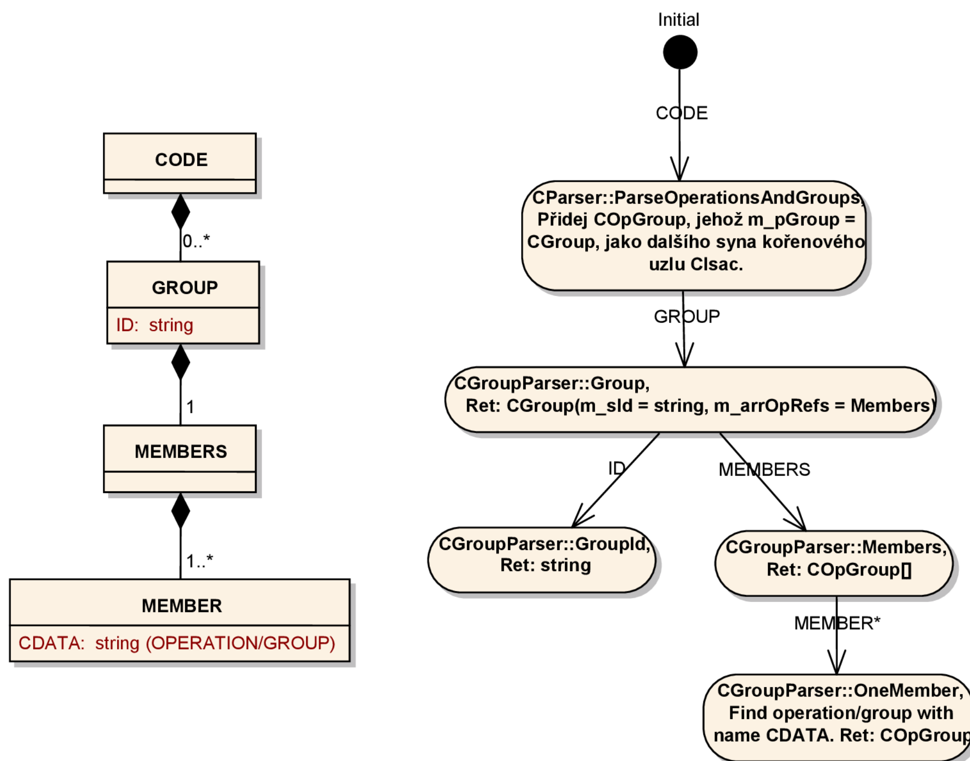
- Parser operací si můžeme představit jako program, který prochází strom vnitřního modelu a tvoří jiný strom objektů operací.
- Strom vnitřního modelu se prochází v preorder průchodem. Pouze s tím rozdílem, že u některých uzlů není pořadí jejich synů důležité a průchod syny nemusí být vykonán v pořadí daném vnitřním modelem (avšak u některých uzlů pořadí důležité je).
- Průchod jedním uzlem stromu vnitřního modelu nám pak vytvoří buď jeden listový uzel, podstrom a nebo více uzlů či podstromů výsledného stromu objektů operací.

Jde tedy o transformaci jednoho stromu na strom jiný. Toto je hlavní idea, kterou se parser řídí a byl takto i navržen. Jedinou věcí navíc, kterou se zabývá, je vyřešení textových odkazů ve vnitřním modelu – to se týká instancí operací, skupin a zdrojových prvků. Vyřešením těchto odkazů sice do stromu operací přidáme zpětné hrany, takže přestane být stromem, ale to nám až tak nevadí a v následujícím popisu tuto skutečnost zanedbávám.

Další věcí, kterou je potřeba zmínit je důraz na modularitu a udržovatelnost. Jak formát vnitřního modelu, tak třídy operací, se mohou a asi i v budoucnu budou měnit, proto bylo při návrhu důležité myslet na to, aby se parser operací dal jednoduše upravit. Z tohoto důvodu je zde podrobně popsán a v přílohách naleznete různé diagramy. Tyto diagramy je potřeba udržovat stále aktuální.

V příloze G.4 můžete vidět diagram popisující parser operací, jeho část můžete vidět v následujícím příkladě. Ve své podstatě jde o stavový automat. Přechody specifikují název elementu vnitřního modelu a stavy jsou funkcemi parseru, které jsou volány v případě vykonání odpovídajícího přechodu. Funkce pak mají specifikované návratové typy. Ty určují, jaký typ objektu je vracen danou funkcí. V následujícím textu budu označovat tento stavový diagram spíše názvem diagram volání funkcí.

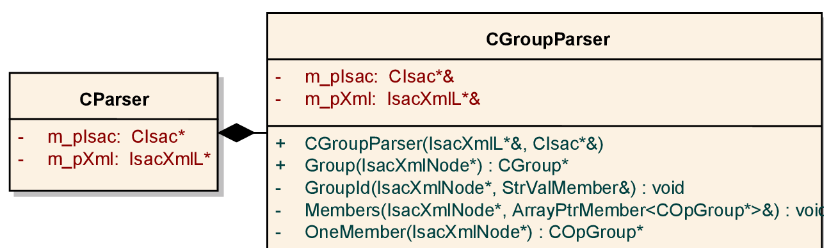
Na příkladě skupin (GROUP) jazyka ISAC si ukážeme, jak parser funguje.



Obrázky A.11 a A.12: Formát vnitřního modelu pro skupiny operací a odpovídající fragment diagramu volání funkcí pro převod stromu vnitřního modelu na strom objektů operací.

Popis diagramu volání funkcí: Jak jsme si již řekli, jeden *stav* diagramu volání funkcí představuje jednu funkci (resp. metodu třídy). Název metody je uveden jako první, pak může následovat stručný popis toho, co metoda dělá. Jako poslední je uvedeno, jaký typ objektu metoda vrací a případně jakým způsobem se nastaví členské prvky (synovské uzly) vraceného objektu. Metoda může vracet i pole objektů, jak můžete vidět v CGroupParser::Members (obrázek A.13).

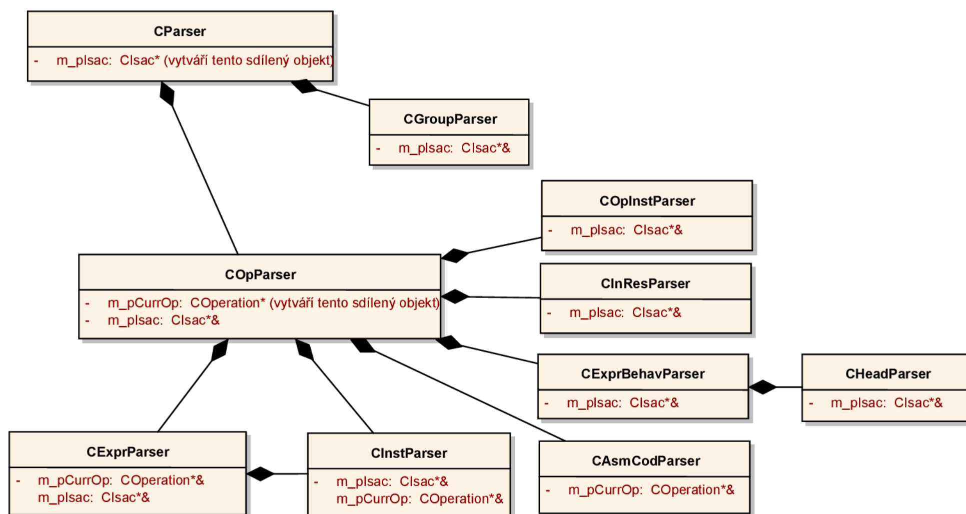
Při návrhu jsme se snažil rozdělit související metody parseru do nezávislých částí. Každá část je pak implementována jako jedna třída. Třídu CGroupParser můžete vidět na následujícím obrázku, zjednodušený diagram tříd na obrázku A.14 a kompletní diagram tříd parseru pak v příloze G.5.



Obrázek A.13: Třída parseru CGroupParser

Třída CGroupParser má dva prvky m_pIsac a m_pXml. První je referencí na ukazatel na kořenový uzel třídy CIsac. Ten je vytvářen třídou CParser a kvůli tomu, že objekt třídy CGroupParser je vytvářen dříve než objekt třídy CIsac (že při vytváření CGroupParser neznáme adresu objektu třídy CIsac), je CGroupParser::m_pIsac referencí na ukazatel a ne pouze jednoduchým ukazatelem. Dalo by se v podstatě říci, že objekt vlastněný CParserem je sdílený mezi dalšími objekty parseru. Podobně to platí i pro prvek CGroupParser::m_pXml, ten ukazuje na sdílený XML parser.

Dalším kritériem pro rozdělení celého parseru samostatných částí bylo to, které ze sdílených objektů využívají. XML parser potřebují všechny, kořenový uzel m_pIsac se využívá pro vyhledávání již zpracovaných zdrojových prvků, operací a skupin podle názvu. Dále existuje ještě jeden, zde nezobrazený sdílený objekt a tím je aktuální operace (třídy COperation). Ten slouží k vyhledávání *deklarovaných instancí* (pomocí konstrukce INSTANCE) a k přidávání nově vytvořených instancí k seznamu využívaných instancí operace. V následujícím diagramu můžete vidět všechny třídy parseru operací s tím, které ze sdílených objektů – kořenový objekt m_pIsac a aktuální operaci m_pCurrOp využívají.



Obrázek A.14: Třídy parseru operací se zobrazenými sdílenými objekty m_pIsac a m_pCurrOp

Nyní bychom si mohli uvést konkrétní příklad, na kterém si ukážeme jednotlivé kroky překladu jedné skupiny popsané v jazyce ISAC.

Mějme skupinu *group*, kterou překládáme a dále skupinu C a operace D a E:

```
GROUP C = ...;
OPERATION D { ... }
OPERATION E { ... }
GROUP group = C,D,E;
```

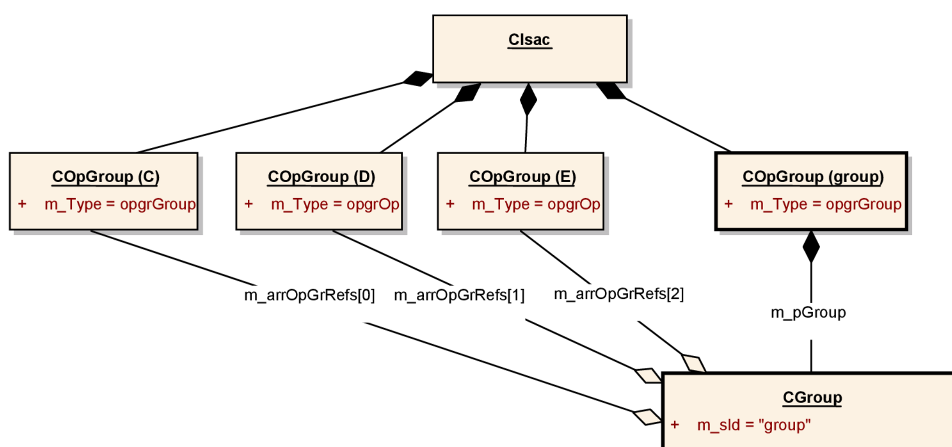
Ve vnitřním modelu jsou informace ze zdrojového ISAC souboru uloženy takto:

```
<CODE>
   <GROUP>
```

```
      <ID>C</ID>
      ...
   </GROUP>
   <OPERATION>
      <ID>D</ID>
      ...
   </OPERATION>
   <OPERATION>
      <ID>E</ID>
      ...
   </OPERATION>

   ...
   <GROUP>
      <ID>group</ID>
      <MEMBERS>
         <MEMBER>C</MEMBER>
         <MEMBER>D</MEMBER>
         <MEMBER>E</MEMBER>
      </MEMBERS>
   </GROUP>
   ...
</CODE>
```

Nyní si skupinu *group* z vnitřního modelu zpracujeme pomocí diagramu volání funkcí na obrázku A.12 a získáme výsledné objekty tříd operací Parserlib2, které představují podstrom kořenového uzlu třídy CIsac (zvýrazněn silnějším rámečkem):



Obrázek A.15: Příklad skupiny operací

Pozn.: Příklad s operacemi byl vybrán z důvodu jeho jednoduchosti. Bohužel příliš nedemonstruje to, že překlad je v podstatě transformací z jednoho stromu do druhého, avšak jiný konkrétní příklad by byl příliš rozsáhlý.

To by bylo o části operací knihovny Parserlib2 asi vše, různé diagramy a další detaily pak v případě zájmu naleznete v přílohách F a G.

Příloha B

Direktivy jazyka assembleru, jejich význam a syntaxe

V této příloze jsou uvedeny direktivy jazyka assembleru, které budou assemblerem podporovány. Pokusil jsem se vybrat minimální podmnožinu direktiv GNU assembleru, tj. ty direktivy, které mají jedinečný význam a nedají se nahradit direktivami jinými. Plánuje se, že assembler bude umět překládat výstup překladače GCC, všechny jím generované direktivy je možné nahradit uvedenými direktivami. U každé z nich je uvedena syntaxe a význam. Jednotlivé direktivy jsou seřazeny podle významu.

Při popisu jsem čerpal z příruček TI The GNU Assembler [38] a z Using as v. 2.17 [27]. Druhou z uvedených příruček budu v následujícím textu označovat jako "manuál GAS".

Některé direktiv mohou obsahovat větší množství volitelných parametrů. Ty se často se týkají přímo určitých formátů objektových souborů. Byly zjednodušeny/upraveny tak, aby odpovídali našemu formátu objektového souboru (zde ho budu označovat jako LOFF, viz [23]), ale zároveň se co nejméně vylučovali s existujícími direktivami pro GNU as. Náš formát vychází z formátu COFF a proto u některých direktiv, které jsou specifické pro určitý objektový formát byla vybrána verze pro COFF. (V manuálu [27] jsou uvedeny verze pro COFF a ELF, v [38] pouze pro COFF.)

Pro co největší budoucí kompatibilitu s výstupem překladače GCC nejsou žádné direktivy nijak zásadně upraveny. V budoucnu, až nám nebudou dostačovat možnosti formátu LOFF, se tyto direktivy budou upravovat tak, aby více odpovídaly direktivám například pro formát pro COFF.

U každé direktivy je popsána její syntaxe, význam a to co se při jejím zpracování assemblerem děje. Dále také jsou uvedeny různé poznámky.

Pozn.: Namísto pojmu direktiva se často také používá ekvivalentní název pseudooperace. Má naprosto stejný význam. V dalším textu bude vždy použit název direktiva.

Při popisu syntaxe jsou použity následující nonterminály s uvedeným významem:

- `Identifier` – Identifikátor, může se skládat ze stejných znaků jako v jazyce C, navíc může obsahovat znak tečka '.' (pro kompatibilitu s GNU as).
- `DefinedExpr` – Výraz, jehož hodnota je po ukončení prvního kroku překladu (po načtení celého souboru) výčíslitelná - atribut *typ_hodnoty* výrazu je *absolutní* nebo *relokatibilní* při započatí druhé fáze překladu (nesmí být nedefinovaný), viz kapitola 6.6.2 a 6.6.3.
- `EvaluableAbsExpr` – Výraz, jehož hodnota přímo při jeho překladu vyčíslitelná a atribut *typ_hodnoty* tohoto výrazu je *absolutní.*

Sekce

.section

Definice syntaxe podle manuálu GAS

```
.section name [, "flags"]
```

Syntaxe

```
SectionDecl:
      '.section' Identifier [',' '"'SectFlag'"']
SectFlag:
      SectOneFlag{SectOneFlag}

SectOneFlag:
      'b' | 'd' | 'x'
```

Význam

Uvedením této direktivy začíná nová pojmenovaná sekce. Následující kód a data budou uložena do sekce s názvem *name*.

Pokud je uveden volitelný parametr *flags*, nastavuje vlastnosti sekce.

b – bbs sekce (neinicializovaná data)
d – datová sekce (inicializovaná data)
x – spustitelná sekce (kód)

Zde uvedené příznaky se navzájem vylučují, ale v budoucnu přibudou další a bude je možné kombinovat.

Akce

Ukončí vytváření předchozí sekce (pokud nějaká byla). Vytvoří novou sekci s požadovaný mi vlastnostmi. Pokud už byla vytvořena sekce se stejným názvem, zahlásí chybu "vícenásobně definovaná sekce".

Nastaví ukazatel pro přidávání nových dat/kódu (location counter) na začátek této nové sekce.

Příznak 'b' koresponduje s příznakem sekce 'B' ve formátu LOFF, 'd' s 'D' a 'x' s 'T' (viz Návrh výstupního formátu pro assembler a linker [17]).

Poznámky

1) V budoucnu nějaké příznaky ještě přibudou a půjdou kombinovat navzájem. Proto je direktiva **.section** definována složitěji, než by bylo potřeba.
2) Příznaky jsou definovány poněkud odlišně od syntaxe pro formát COFF.
3) Gas nijak nekontroluje to, co se může v sekci dále objevit. Například je možné psát kód do bbs sekce. Ten pak vygeneruje sice nesmyslný, ale platný objektový soubor. V tomto případě obsahuje sekci, která je správně tvořena zadaným kódem, ale tato sekce se při spuštění nenahrává do paměti. Osobně bych kontrolu toho, co se může v které sekci objevit zavedl a pokud se objeví něco nekalého, tak by se zahlásilo varování (ne chyba). Hlášení tohoto varování by mělo jít zapnout či vypnout z důvodu, že pro různé architektury třeba může být potřeba kombinovat data a kód v jedné sekci.

Symboly

.global

Definice syntaxe podle manuálu GAS

```
.global symbol
```

Syntaxe

```
SymbolDirective:
      '.global' Identifier
```

Význam

Nastaví, aby symbol s názvem *symbol* byl viditelný pro linker.

Akce

Pokud již v tabulce symbolů existuje symbol s názvem *symbol*, nastaví jeho typ na *globální*. Pokud symbol ještě definován nebyl, vytvoří nový symbol, nastaví, že je globální a má zatím nedefinovanou hodnotu.

Poznámky

1) GCC verze 3.4.2 generuje namísto ".global" ekvivalentní ".globl". Náhradu provede preprocesor. Podobně, některé překladače také používají direktivu .xdef se stejným významem, její náhradu také necháme na preprocesoru.

.comm, .lcomm

Definice syntaxe podle manuálu GAS

.**comm** *symbol*, *length*
.**lcomm** *symbol*, *length*

Syntaxe

```
SymbolDirective:
      '.comm' Identifier ',' EvaluableAbsExpr
      '.lcomm' Identifier ',' EvaluableAbsExpr
```

Význam

Direktiva .**comm** definuje globální společný symbol a .**lcomm** lokální společný symbol. V podstatě jde o deklaraci ukazatele na data o velikosti *length* v bajtech a *symbol* je návěštím označujícím adresu prvního bajtu těchto dat.

Pro globální symboly (.**comm**):

Pokud není symbol definován (jeho pozice není definována), linker alokuje místo o velikosti *length* v neinicializované datové sekci. Při linování linker spojí dvě nebo více společných deklarací stejného společného symbolu. Pokud mají různou velikost, vybere se ta největší.

Pro lokální symboly (.**lcomm**):

Alokuje v neinicializované datové sekci místo o velikosti *length* a definuje symbol s hodnotou, kde se alokované místo nachází. *Symbol* je lokální – není viditelný pro linker.

GCC generuje tyto direktivy pro globální proměnné. Mějme například dvě globální proměnné - řetězec a znak:

V jazyce C:

```
char retezec[20];
int cislo;
```

po překladu:

```
.comm _retezec, 32
.comm _cislo, 16
```

Podobně pro statické proměnné:

V jazyce C:

```
static char retezec[20];
static int cislo;
```

a po překladu:

```
.lcomm _retezec, 32
.lcomm _cislo, 16
```

Překladač GCC používá tyto direktivy pro alokaci místa v sekci s neinicializovanými daty. (Pozn. Překladač proměnné zarovnal na hranice 16 bajtů.)

Tato direktiva by se měla používat pouze v bbs sekci.

Akce

Definuje lokální či globální symbol s hodnotou rovnou aktuální hodnotě location counteru (LC). V aktuální subsekci rezervuje místo o velikosti *length*. Posune LC na konec alokovaného místa.

.equiv

Definice syntaxe podle manuálu GAS

```
.equiv symbol, expression
```

Syntaxe

```
SymbolDirective:
      '.equiv' Identifier ',' DefinedExpr
```

Význam

Definuje symbol s absolutní hodnotou, tento symbol nesmí být definován znovu.

Akce

Pokud již symbol s názvem *symbol* existuje v tabulce symbolů, zahlásí se chyba "Vícenásobně definované návěští/symbol".

Pokud ještě definován nebyl, v tabulce symbolů se vytvoří nový symbol s názvem *symbol* a s hodnotou odpovídající výrazu *expression*.

Poznámky

1) GNU as používá dále direktivy ".equ" a k ní ekvivalentní ".set". Ty mají stejný význam jako ".equiv", avšak definují symbol, který je možné předefinovat. Osobně příliš nevím, k čemu toto může sloužit a ani ve výstupech gcc jsem neobjevil případ, kdy by bylo předefinování symbolu použito.

Definice inicializovaných dat

.bit
.signbit

Definice syntaxe podle manuálu GAS

```
.bit n, values
.signbit n, signpos, values
```

Syntaxe

```
DataDefDirective:
      '.bit' EvaluableAbsExpr ',' DefinedExpr {',' DefinedExpr}
      '.signbit' EvaluableAbsExpr ',' DefinedExpr {',' DefinedExpr}
```

Význam

Definují inicializovaná data o bitové šířce *n* a pro znaménková čísla se znaménkovým bitem na bitové pozici *signpos*. Hodnot *values* může být uvedena 1krát až libovolněkrát.

Akce

Na konec aktuální sekce v tabulce sekcí se přidají uvedené hodnoty v odpovídajícím formátu. Pokud mají zadané hodnoty v binární podobě větší počet bitů, než je povoleno, zahlásí se varování a hodnoty se zleva oříznou.

Příloha C

Fáze vývoje assembleru

Vývoj assembleru půjde rozdělit do několika fází. Výsledkem každé fáze je použitelný assembler, který se pak může používat v ostatních částí projektu Lissom. Samotný kodér instrukcí a preprocesor se budou moci vyvíjet nezávisle na assembleru.

1. fáze

Výsledkem 1. fáze assembleru bude jednoduchý assembler bez podpory direktiv, symbolů a výrazů. Vytvoří se:

- specifikace rozhraní kodéru instrukcí, jednotky pro zpracování direktiv a jednotky pro zpracován symbolů a výrazů
- alespoň částečná specifikace rozhraní a struktur generátoru objektového souboru
- kompletní lexikální analyzátor, generátor generované části lex. analyzátoru, v dalších fázích se budou pouze přidávat různá klíčová slova (jako jsou direktivy) a např. operátory pro výrazy.
- syntaktický analyzátor, bez podpory direktiv, a výrazů, generátor generované části synt. analyzátoru
- kodér instrukcí
- jednotka pro výpis zpráv o chybách

2. fáze

Ve druhé fázi zavedeme podporu direktiv pro vytváření objektového souboru a assembler bude umět generovat objektový soubor pro linker.

- vytvoření jednotky pro zpracování direktiv
- vytvoření generátoru objektového souboru s využitím knihovny objfilelib
- rozšíření lex., a synt. analyzátoru o podporu zavedených direktiv
- podpora direktiv .section a .org

3. fáze

Ve třetí fázi rozšíříme assembler o podporu symbolů a výrazů a jejich relokaci.

- vytvoření jednotky pro zpracování symbolů a výrazů
- vytvoření tabulky symbolů a tabulky relokací
- rozšíření generátoru obj. souboru o podporu symbolů a jejich relokaci
- podpora návěští a direktiv .global a .equiv

4. fáze

Ve čtvrté fázi zavedeme podporu pro definování inicializovaných a neinicializovaných dat.

- rozšíření jednotky pro zpracování direktiv o podporu direktiv .comm, .lcomm, .space, dále také o podporu .bit *n*.
- jediná součást, co se bude muset upravit je kromě lex. a synt. analyzátoru pouze jednotka pro zpracování direktiv

5. fáze

Listing výpisy.

- specifikace různých parametrů assembleru pro určení formátu listing výpisů
- zavedení direktiv pro listing výpisy

6. fáze

Podpora ladicích informací.

- vytvoření jednotky zpracovávající ladicí informace
- zavedení direktiv pro ladicí informace

7. fáze

V sedmé fázi rozšíříme assembler o podporu direktivy ".endianess". I když většina architektur používá pouze jediný způsob uložení čísel, existují architektury, kde je tento mód možné přepnout, či existují speciální instrukce pro práci s opačnou číselnou reprezentací. U architektur, kde se dá mód přepnout však většinou pro operandy instrukcí používají stále stejný mód a změna se týká pouze dat (např. SPARC či Atmel AVR32). Proto je nutné umožnit, aby se mohlo ve zdrojovém souboru assembleru mohli určit úseky, kde se který mód bude používat. (informace o vybraném módu budou muset nést i symboly ve výsledném objektovém souboru).

- zavedení podpory direktivy .endianess
- úprava metod pro převod čísel tak, aby převáděly čísla do vybraného formátu

8. fáze

Zavedení podpory "attribute displacement". V jazyce ISAC je možné popsat operaci, která se má provést s operandem instrukce před tím, než je hodnota operandu dosazena do zakódované instrukce

(viz Jazyk ISAC – Příručka [7], str. 20). Tato operace se označuje buď jako speciální typ relokace a nebo "displacement" atributu.

9. fáze

Podpora relaxace. Tu bude ještě potřeba promyslet, viz kapitola 4.7.

10. fáze

Přidávání a vylepšování různých kontrol sloužících například k tomu, abychom mohli ověřit, že překladač vyššího jazyka generuje korektní kód.

Příloha D

Diagram komponent assembleru

Příloha E

Struktury GNU as

Příloha F
Formát vnitřního modelu procesoru

Příloha F.1

Vnitřní model – část operací

Příloha F.2

Vnitřní model – část zdrojů

Příloha G
Diagramy knihovny Parserlib2

Příloha G.1

Diagram tříd operací Parserlib2

Příloha G.2

Diagram pomocných tříd představujících členské prvky tříd operací Parseerlib2

Příloha G.3

Příklad uložení informací o jedné operaci do tříd operací Parserlib2

Příloha G.4

Diagram volání funkcí XML parseru knihovny Parserlib2 pro část operací

Příloha G.5

Diagram tříd XML parseru knihovny Parserlib2 pro část operací