VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

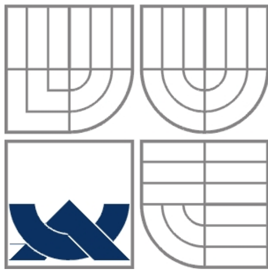
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

IMPLEMENTACE OBECNÉHO ZPĚTNÉHO ASSEMBLERU

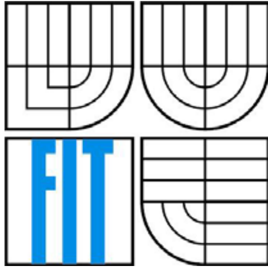
DIPLOMOVÁ PRÁCE
MASTER‘S THESIS

AUTOR PRÁCE Bc. ZDENĚK PŘIKRYL
AUTHOR

BRNO 2007



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 ZPĚTNÉHO ASSEMBLERU

IMPLEMENTATION OF GENERAL DISASSEMBLER

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE Bc. ZDENĚK PŘIKRYL
AUTHOR

VEDOUCÍ PRÁCE Ing. ROMAN LUKÁŠ, Ph.D.
SUPERVISOR

BRNO 2007

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav informačních systémů Akademický rok 2006/2007

Zadání diplomové práce

Řešitel: **Přikryl Zdeněk, Bc.**
Obor: Informační systémy
Téma: **Implementace obecného zpětného assembleru**
Kategorie: Překladače

Pokyny:

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 zpětného assembleru, přičemž využijte navržené struktury z bakalářského projektu pod názvem: "Návrh struktury obecného assembleru a zpětného assembleru".
4. Diskutujte případné možné další rozšíření implementovaného modelu obecného zpětného assembleru.

Literatura:

- Meduna, A: Automata and Languages, Springer, 2000

Při obhajobě semestrální části diplomového projektu je požadováno:

- Splnění bodů 1. a 2.

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

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

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

Vedoucí: **Lukáš Roman, Ing., Ph.D.**, UIFS FIT VUT
Datum zadání: 28. února 2006
Datum odevzdání: 22. května 2007

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

L.S.

doc. Ing. Jaroslav Zendulka, CSc.
vedoucí ústavu

LICENČNÍ SMLOUVA
POSKYTOVANÁ K VÝKONU PRÁVA UŽÍT ŠKOLNÍ DÍLO

uzavřená mezi smluvními stranami

1. Pan

Jméno a příjmení: **Bc. Zdeněk Přikryl**
Id studenta: 47633
Bytem: Bořetická 4134/8, 628 00 Brno
Narozen: 26. 07. 1983, Boskovice
(dále jen "autor")

a

2. Vysoké učení technické v Brně
Fakulta informačních technologií
se sídlem Božetěchova 2/1, 612 66 Brno, IČO 00216305
jejímž jménem jedná na základě písemného pověření děkanem fakulty:

....
(dále jen "nabyvatel")

Článek 1
Specifikace školního díla

1. Předmětem této smlouvy je vysokoškolská kvalifikační práce (VŠKP):
diplomová práce

Název VŠKP: Implementace obecného zpětného assembleru
Vedoucí/školitel VŠKP: Lukáš Roman, Ing., Ph.D.
Ústav: Ústav informačních systémů
Datum obhajoby VŠKP: ....

VŠKP odevzdal autor nabyvateli v:
tištěné formě počet exemplářů: 1
elektronické formě počet exemplářů: 2 (1 ve skladu dokumentů, 1 na CD)

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

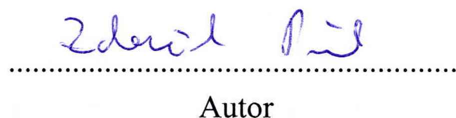
1. Autor touto smlouvou poskytuje nabyvateli oprávnění (licenci) k výkonu práva uvedené dílo nevýdělečně užít, archivovat a zpřístupnit ke studijním, výukovým a výzkumným účelům včetně pořizování výpisů, opisů a rozmnoženin.
2. Licence je poskytována celosvětově, pro celou dobu trvání autorských a majetkových 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: ....



....	....
Nabyvatel	Autor

Abstrakt

Tato práce popisuje proces vytváření disassembleru pro nově navrhované procesory. Kritériem při vytváření je jeho automatické vygenerování. Instrukční sada pro procesor je modelována pomocí specializovaného jazyka ISAC, který obsluhuje prostředky pro popis této instrukční sady, jako je například formát instrukce v jazyku symbolických instrukcí, binární zápis instrukce a chování instrukce. Vnitřním modelem je párový konečný automat, který formálně popíše vztah mezi textovou reprezentací instrukce a binárním kódováním instrukce. Z tohoto vnitřního modelu je generován kód překladače – disassembleru. Ten na vstupu přijímá program ve strojovém kódu a generuje ekvivalentní program v jazyce symbolických instrukcí.

Klíčová slova

Konečný automat, párový konečný automat, assembler, disassembler, simulace, jazyky pro popis procesorů, vestavěný systém.

Abstract

This thesis presents the process of creating disassembler for new designed processors. We demand automatic generation of the disassembler. Instruction set for processor is modeled by specialized language ISAC, which offers resources for description of the instruction set. For example it describes format of instruction in the assembly language or format of instruction in the binary form or behavior of this instruction. Internal model is coupled finite automata, which describes relation of textual form of the instruction and binary form of the instruction in formal way. The code of disassembler is generated from the internal model. This disassembler accepts program in binary code at the input and generate equivalent program in assembly language at the output.

Keywords

Finite automata, coupled finite automata, assembler, disassembler, simulation, languages for specification processors, embedded systems.

Citace

Zdeněk Přikryl: Implementace obecného zpětného assembleru, diplomová práce, Brno, FIT VUT v Brně, 2007

Implementace obecného zpětného assembleru

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením Ing. Romana Lukáše, Ph.D.

Další informace mi poskytli Ing. Karel Masařík, Prof. Ing. Tomáš Hruška, Csc. a Doc. Dr. Ing. Dušan Kolář.

Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

Diplomová práce byla vypracována na základě veřejně přístupného kódu programu TinyXml a Eclipse.

……………………
Zdeněk Přikryl
20. 5. 2007

Poděkování

Chtěl bych poděkovat Ing. Romanu Lukášovi, Ph.D., vedoucímu diplomové práce, za odbornou pomoc při konzultacích, za ochotu a čas, který mi věnoval při tvorbě této diplomové práce. Dále bych chtěl poděkovat Prof. Ing. Tomáši Hruškovi, Csc., pod jehož vedením projekt vznikl, za jeho odborné konzultace.

© Zdeněk Přikryl, 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
1 Úvod 4
2 Základní pojmy 6
2.1 Definice z formálních jazyků 6
2.1.1 Základní pojmy 6
2.1.2 Regulární jazyky a jejich modely 7
2.1.3 Bezkontextové jazyky a jejich modely 10
2.1.4 Překladač 12
2.2 Procesory a jejich architektury 13
2.2.1 Procesor 13
2.2.2 Sériové a zřetězené zpracování instrukcí 14
2.2.3 Architektury 15
2.3 Základní pojmy ze simulací 19
2.3.1 Simulační model 19
2.3.2 Simulace 19
2.3.3 Simulátor 20
2.4 Zpětné inženýrství 20
2.4.1 Disassembler 22
2.4.2 Dekompilátor 22
2.5 Metody disassemblování 22
2.5.1 Zarovnání paměti 22
2.5.2 Relokační údaje 23
2.5.3 Základní metody 24
2.5.4 Pokročilé metody 26
2.6 Další pojmy 27
2.6.1 Strojový kód 27
2.6.2 Jazyk symbolických instrukcí 27
2.6.3 Assembler 27
2.6.4 Linker 27
2.6.5 Vestavěný systém 27
2.6.6 XML 28
3 Vývoj a návrh procesorů 29
3.1 Jazyky pro popis procesoru 30
3.1.1 Jazyky zaměřené na popis architektury 30

3.1.2 Jazyky zaměřené na popis instrukční sady ... 30
3.1.3 Jazyky zaměřené na popis instrukční sady a architektury ... 30
4 Cíl a metodika práce ... 31
5 Projekt Lissom ... 32
5.1 Jazyk ISAC ... 32
5.1.1 Popis zdrojů ... 32
5.1.2 Popis operační části jazyka ISAC ... 33
5.1.3 Překladač pro operační část jazyka ISAC ... 35
6 Schéma překladu popisu procesoru v XML na disassembler ... 36
6.1 Stávající vnitřní model a metodologie vytváření disassembleru ... 36
6.1.1 Formální popis algoritmus pro převod XML na PPG ... 36
6.1.2 Překlad pomocí PPG ... 38
6.1.3 Zhodnocení ... 38
6.2 Výběr nového vnitřního modelu ... 39
6.2.1 Teorém 1 ... 39
6.2.2 Teorém 2 ... 39
6.2.3 Teorém 3 ... 41
6.2.4 Závěr ... 41
6.3 Nová metodologie vyváření disassembleru ... 42
6.3.1 Převod XML na PKA ... 42
6.3.2 Poloformální popis algoritmu pro převod XML na PKA ... 43
6.3.3 Ukázka algoritmu ... 45
6.3.4 Překlad pomocí PKA ... 47
7 Implementace disassembleru ... 51
7.1 Jazyk Bison-- ... 51
7.2 Proces vytváření disassembleru ... 52
7.3 Ilustrativní příklad ... 54
8 Problémy k dalšímu řešení a využití disassembleru ... 58
8.1 Větvící konstrukce ... 58
8.2 Sekce ACTIVATION a CODINGROOT ... 59
8.3 Rozšíření překladače jazyka Bison-- ... 60
8.4 Podpora vstupního formátu ... 60
8.5 Využití disassembleru v simulátorech ... 61
8.5.1 Instrukční simulátor ... 61
8.5.2 Kompilovaný simulátor ... 62
9 Závěr ... 63
Literatura ... 64

Příloha č. 1 66

1 Úvod

V současné době je trendem řešit výpočetní jednotku v aplikacích, pomocí tzv. vestavěných systémů. Jde o systémy, do kterých je zpracování informací vloženo/vestavěno a vše je řízeno obecným či specializovaným procesorem. Jejich použití je mnohdy vhodnější a efektivnější, než užití klasického desktopového počítače. Důvodem může byt potřebná využitá plocha, zahřívání apod. Velký rozvoj probíhá hlavně v multimediálních a síťových aplikacích.

U obou aplikací se začíná mluvit o tzv. Very Large instruction word (VLIW) Application Specific Instruction set Processors (ASIP). Jedná se o silně paralelní systémy, které mají uzpůsobenou instrukční sadu pro specifikou množinu aplikací.. Stejně tak je upraven i hardwarový návrh. Díky tomuto se pak dosahuje velmi vysokých výkonů při velmi nízké spotřebě energie.

Téměř ve všech průmyslových odvětvích dochází k postupnému nahrazování výkonných jednojaderných procesorů systémy paralelními. Jednojaderné CPU jsou velké, mají vysoký příkon a s tím spojený ztrátový výkon, složité zapojení periferií atp. Naproti tomu jsou postaveny systémy využívající velké množství funkčních jednotek pro výpočet, každá vysoce optimalizovaná na jeden druh operací (VLIW procesory) nebo systémy vícejaderné. Ty se fyzicky jeví jako jeden procesor, avšak uvnitř je tvořen až *n* procesory – výpočetními jádry. Vícejaderné procesory vykazují daleko lepší vlastnosti než monolitický procesor.

Proto je nutné mít silný prostředek na návrh, popis, testování a simulaci těchto procesorů. Zvláštní důraz je kladen na simulaci běhu jednoho procesoru na jiném. K tomu je nutné mít výkonný a univerzální procesor, který může být popsán obecným programovacím jazykem, např. C/C++, nebo jazykem specializovaným. Obecný jazyk není příliš vhodný, protože spolu s námi požadovanými vlastnostmi musíme do popisu zahrnout i skutečnosti, které nejsou relevantní a jsou dány syntaxí jazyka. Specializovaný jazyk s sebou přináší čistější popis a syntax jazyka je konstruována pro popis vytvářeného procesoru. Na téma specializovaných jazyků již běží několik projektů, např. Shankya, Lissom[1] a jiné.

Druhým jmenovaným projektem se budeme zabývat blíže. Lissom si klade za cíl vytvořit komplexní prostředek pro vývoj procesorů. Je zde využito modelovacího jazyka ISAC, kterým lze úplně popsat procesor. Popis je rozdělen na dvě nezávislé části. První slouží pro popis zdrojů a hardwarové stránky procesoru, jako jsou např. paměti, registry a další součásti. Druhá je pak určena pro instrukce – jak se zapisují v jazyku symbolických instrukcí, způsob zakódování do binární podoby, je zde popsáno i jejich chování při zpracovávání procesorem.

Z těchto popisů jsou na základě podpůrných programů, jako jsou překladače jazyka ISAC, následně vytvářeny základní součásti procesoru, kterými jsou např. assembler a disassembler (zpětný

[1] Více informaci o projektu na http://lissom.aps-brno.cz

assembler). Nepostradatelný je také linker, který spojí přeložené moduly z assembleru a vytvoří výsledný spustitelný program. Nepostradatelný je také simulátor, který za pomoci modelu procesoru a vstupních dat (program, strojový kód, ...) dokáže na jiném procesoru simulovat chování modelovaného procesoru. Kritickou veličinou zde většinou bývá rychlost simulace.

V dalším textu se budeme věnovat způsobu popisu instrukcí, jejich zpracováním a vytvořením disassembleru pro popsanou instrukční sadu a jeho využití v simulátorech.

2 Základní pojmy

V této kapitole jsou vysvětleny pojmy, které usnadní pochopení nebo ilustraci dalšího textu.

2.1 Definice z formálních jazyků

2.1.1 Základní pojmy

V následující části jsou popsané pojmy, které jsou nutné pro vysvětlení modelů, na kterých je vystavěna implementace obecného zpětného assembleru.

2.1.1.1 Definice abecedy

Abeceda je neprázdná konečná množina prvků, které nazýváme symboly.

2.1.1.2 Definice řetězce nad danou abecedou

Nechť Σ je abeceda. Potom:

- ε je řetězec nad abecedou Σ.
- Jestliže x je řetězec nad abecedou Σ, $a \in \Sigma$, potom xa je řetězec nad abecedou Σ.

Poznámka:

1) Symbol ε značí prázdný řetězec. Prázdný řetězec je takový řetězec, který neobsahuje žádný symbol.
2) Symbolem Σ^* budeme značit množinu všech řetězců nad abecedou Σ.

2.1.1.3 Definice délky řetězce

Nechť x je řetězec nad abecedou Σ. Délka řetězce x, $|x|$, je definována následovně:

- Pokud $x = \varepsilon$, potom $|x| = 0$.
- Pokud $x = a_1 \ldots a_n$, kde pro $\forall i$: $i = 1..n$: $a_i \in \Sigma$, potom $|x| = n$.

2.1.1.4 Definice binární operace konkatenace

Nechť x, y jsou dva řetězce nad abecedou Σ. Konkatenací řetězce x s řetězcem y vznikne řetězec xy připojením řetězce y za řetězec x. Operace konkatenace je asociativní, ale není komutativní.

2.1.1.5 Definice permutace řetězce

Nechť x je řetězec nad abecedou Σ. Pro každé $x \in \Sigma^*$, je $perm(x)$ značí konečnou množinu všech permutací řetězce x., kde:

- $\text{perm}(\varepsilon) = \{\varepsilon\}$,

- pro všechny $a \in \Sigma$, $x \in \Sigma^*$, $perm(ax) = \{qap : qp = z, z \in perm(x)\}$.

2.1.1.6 Definice množiny alph

Nechť x je řetězec nad abecedou Σ. Pro každé $x \in \Sigma^*$, $alph(x)$ značí množinu všech symbolů vyskytující se v řetězci x.

2.1.1.7 Definice formálního jazyka

Nechť je dána abeceda Σ. Potom množinu L, pro kterou platí: $L \subseteq \Sigma^*$, nazveme formálním jazykem nad abecedou Σ.

2.1.2 Regulární jazyky a jejich modely

2.1.2.1 Definice regulárního výrazu

Regulární výrazy nad abecedou Σ a jazyky, které značí, definujeme:

- $\emptyset$ je regulární výraz značící prázdnou množinu (prázdný jazyk),
- ε je regulární výraz značící jazyk $\{\varepsilon\}$,
- a, kde $a \in \Sigma$, je regulární výraz značící jazyk $\{a\}$,
- nechť r a s jsou regulární výrazy značící po řadě jazyky Lr a Ls, potom
  a) (r.s) je regulární výraz značící jazyk $L = LrLs = \{xy: x \in Lr \wedge y \in Ls\}$,
  b) (r+s) je regulární výraz značící jazyk $L = Lr \cup Ls = \{x: x \in Lr \vee x \in Ls\}$,
  c) (r*) je regulární výraz značící jazyk $L = Lr^*$.

2.1.2.2 Definice regulárního jazyka

Nechť L je jazyk. L je *regulární jazyk*, pokud existuje regulární výraz r, který jazyk L značí.

2.1.2.3 Definice konečného automatu

Konečný automat je pětice $M = (Q, \Sigma, R, s, F)$, kde:

- Q je konečná množina stavů,
- Σ je vstupní abeceda,
- R je konečná množina pravidel ve tvaru $pa \rightarrow q$, kde $p,q \in Q$, $a \in \Sigma \cup \{\varepsilon\}$,
- $s \in Q$ je startovací stav,
- $F \subseteq Q$ je množina koncových stavů.

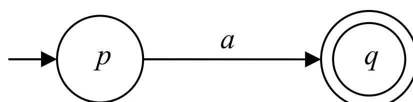
Poznámka

Pokud R obsahuje pravidla ve tvaru $pa \rightarrow q$, kde $p,q \in Q$, $a \in \Sigma$ a dále platí pro každé $pa \rightarrow q \in R$, že množina $R - \{ pa \rightarrow q \}$ neobsahuje žádné pravidlo s levou stranou pa, pak se M nazývá deterministický konečný automat.

Často se místo textové reprezentace automatu používá grafická. Vizualizací pomůže ke snadnější orientaci v pravidlech a stavech.

Platí následující pravidla:

- startovací stav je uvozen orientovanou neohodnocenou hranou, která nevede z žádného stavu
- koncové stavy jsou v dvojitém orámování
- stavy jsou propojeny orientovanými ohodnocenými hranami



Obr. 1: Grafické znázornění konečného automatu

2.1.2.4 Definice konfigurace KA

Nechť $M = (Q, \Sigma, R, s, F)$ je konečný automat, pak konfigurace C automatu M je uspořádaná dvojice

$$C = (q, w), (q, w) \in Q \times \Sigma^*$$

kde q je aktuální stav, w je doposud nezpracovaná část vstupního řetězce

Poznámka

Počáteční konfigurace je konfigurace $(s, a_1a_2\ldots a_n)$ kde $n \geq 0$ a $a_i \in \Sigma$ kde $i = 1..n$.

Koncová konfigurace je konfigurace (q_f, ε), $q_f \in F$

Místo relačního zápisu $(q, w) \in Q \times \Sigma^*$ budeme používat zápis $C = qw$, kde $q \in Q$, $w \in \Sigma^*$.

2.1.2.5 Definice přechodu, posloupnosti přechodů v KA

Přechod

Nechť $M = (Q, \Sigma, R, s, F)$ je konečný automat, a pax a qx jsou dvě konfigurace, kde $p,q \in Q$, $a \in \Sigma \cup \{\varepsilon\}$, $x \in \Sigma^*$. Nechť $r = pa \rightarrow q \in R$ je pravidlo. Potom M může provést přechod z pax do qx za použití r, zapsáno $pax \vdash qx\ [r]$ nebo zjednodušeně $pax \vdash qx$.

Posloupnost přechodů

Nechť χ je konfigurace. M provede nula přechodů z χ do χ; zapisujeme: $\chi \vdash^0 \chi\ [\varepsilon]$ nebo zjednodušeně $\chi \vdash^0 \chi$.

Nechť $\chi_0, \chi_1, \ldots, \chi_n$ je sekvence přechodů konfigurací pro $n \geq 1$ a $\chi_{i-1} \vdash \chi_i\ [r_i]$, $r_i \in R$ pro všechna $i \in 1,..,n$, což znamená:

$$\chi_0 \vdash \chi_1\ [r_1] \vdash \chi_2\ [r_2] \ldots \vdash \chi_n\ [r_n]$$

Pak M provede n-přechodů z χ_0 do χ_n; zapisujeme:

$$\chi_0 \vdash \chi_n\ [r_1 \ldots r_n] \text{ nebo zjednodušeně } \chi_0 \vdash \chi_n.$$

Pokud $\chi_0 \vdash^n \chi_n\ [\rho]$ pro nějaké $n \geq 1$, pak $\chi_0 \vdash^+ \chi_n\ [\rho]$.

Pokud $\chi_0 \vdash^n \chi_n\ [\rho]$ pro nějaké $n \geq 0$, pak $\chi_0 \vdash^* \chi_n\ [\rho]$.

2.1.2.6 Definice jazyka přijímaného KA

Nechť $M = (Q, \Sigma, R, s, F)$ je konečný automat, pak jazyk přijímaný M, $L(M)$ je definován takto:

$$L(M) = \{w \mid w \in \Sigma^*, sw \vdash^* f, f \in F\}.$$

2.1.2.7 Definice líného konečného automatu

Líný konečný automat je pětice $M = (Q, \Sigma, R, s, F)$, kde:

- Q je konečná množina stavů,
- Σ je vstupní abeceda,
- R je funkce konečná množina pravidel ve tvaru $pa \rightarrow q$, kde $p,q \in Q$, $a \in \Sigma^*$,
- $s \in Q$ je startovaní stav,
- $F \subseteq Q$ je množina koncových stavů.

Poznámka

Nechť M je konečný automat. Potom existuje líný konečný automat M', pro který platí, že $L(M) = L(M')$. Tedy existují transformace, jak převést M' na ekvivalentní M.

2.1.2.8 Definice konfigurace LKA

Nechť $M = (Q, \Sigma, R, s, F)$ je líný konečný automat, pak konfigurace C automatu M je uspořádaná dvojice

$$C = (q, w), (q, w) \in Q \times \Sigma^*$$

kde q je aktuální stav, w je doposud nezpracovaná část vstupního řetězce

Poznámka

Počáteční konfigurace je konfigurace $(s, a_1a_2\ldots a_n)$ kde $n \geq 0$ a $a_i \in \Sigma$ kde $i = 1..n$.

Koncová konfigurace je konfigurace (q_f, ε), $q_f \in F$

Místo relačního zápisu $(q, w) \in Q \times \Sigma^*$ budeme používat zápis $C = pw$, kde $p \in Q$, $w \in \Sigma^*$.

2.1.2.9 Definice přechodu, posloupnosti přechodů v LKA

Přechod

Nechť $M = (Q, \Sigma, R, s, F)$ je líný konečný automat, a pax a qx jsou dvě konfigurace, kde $p,q \in Q$ a a, $x \in \Sigma^*$. Nechť $r = pa \rightarrow q \in R$ je pravidlo. Potom M může provést přechod z pax do qx za použití r, zapsáno $pax \vdash qx\ [r]$ nebo zjednodušeně $pax \vdash qx$.

Posloupnost přechodů

Nechť χ je konfigurace. M provede nula přechodů z χ do χ; zapisujeme: $\chi \vdash^0 \chi\ [\varepsilon]$ nebo zjednodušeně $\chi \vdash^0 \chi$.

Nechť $\chi_0, \chi_1, ..., \chi_n$ je sekvence přechodů konfigurací pro $n \geq 1$ a $\chi_{i-1} \vdash \chi_i\ [r_i]$, $r_i \in R$ pro všechna $i \in 1,..,n$, což znamená:

$$\chi_0 \vdash \chi_1\ [r_1] \vdash \chi_2\ [r_2] \ldots \vdash \chi_n\ [r_n]$$

Pak M provede n-přechodů z χ_0 do χ_n; zapisujeme:

$$\chi_0 \vdash \chi_n\ [r_1 \ldots r_n] \text{ nebo zjednodušeně } \chi_0 \vdash \chi_n.$$

Pokud $\chi_0 \vdash^n \chi_n\ [\rho]$ pro nějaké $n \geq 1$, pak $\chi_0 \vdash^+ \chi_n\ [\rho]$.

Pokud $\chi_0 \vdash^n \chi_n\ [\rho]$ pro nějaké $n \geq 0$, pak $\chi_0 \vdash^* \chi_n\ [\rho]$.

2.1.2.10 Definice jazyka přijímaného LKA

Nechť $M = (Q, \Sigma, R, s, F)$ je líný konečný automat, pak jazyk přijímaný M, $L(M)$ je definován takto:

$$L(M) = \{w \mid w \in \Sigma^*, sw \vdash^* f, f \in F\}.$$

2.1.2.11 Definice párového konečného automatu

Párový konečný automat je trojice $\Gamma = (M_1, M_2, h)$, kde:

- $M_i = (Q_i, \Sigma_i, R_i, s_i, F_i)$ je líný konečný automat pro $i \in \{1, 2\}$,
- h je bijektivní zobrazení z R_1 do R_2.

  Nechť h^* je zobrazení z R_1^* do R_2^* a je definováno takto:

  - $h^*(\varepsilon) = \{\varepsilon\}$
  - pro $r_1, r_2, \ldots, r_n \in R_1$ je $h^*(r_1, r_2, \ldots, r_n) = h(r_1)(r_2)\ldots(r_n)$, kde $n \geq 1$.

2.1.2.12 Definice překladu PKA

Nechť $\Gamma = (M_1, M_2, h)$ je párový konečný automat, pak překlad automatu Γ, $T(\Gamma)$ je definován takto:

$$T(\Gamma) = \{(w_1, w_2) : w_1 \in \Sigma_1^*, w_2 \in \Sigma_2^*, s_1 w_1 \vdash^* f_1\ [\rho_1] \text{ v } M_1, s_2 w_2 \vdash^* f_2\ [\rho_2] \text{ v } M_2, f_1 \in F_1, f_2 \in F_2, \rho \in perm(\rho_1), \rho_2 = h^*(\rho)\}.$$

2.1.3 Bezkontextové jazyky a jejich modely

2.1.3.1 Definice gramatiky

Gramatika G je čtveřice $G = (N, \Sigma, P, S)$, kde:

- N je konečná množina nonterminálních symbolů,
- Σ je konečná množina terminálních symbolů, přičemž $N \cap \Sigma = \emptyset$,
- P je konečná podmnožina kartézského součinu: $(N \cup \Sigma)^* N (N \cup \Sigma)^* \times (N \cup \Sigma)^*$ nazývaná množina přepisovacích pravidel, prvek $(\alpha, \beta) \in P$ je přepisovací pravidlo a zapisuje se ve tvaru $\alpha \rightarrow \beta$,
- S je startovací nonterminální symbol.

Poznámka

Pokud v gramatice G obsahuje množina přepisovacích pravidel P pouze prvky tvaru: $(N \cup \Sigma)^* N (N \cup \Sigma)^* \times (N \cup \Sigma)^+$, potom řekneme, že gramatika G neobsahuje ε-pravidla.

2.1.3.2 Definice derivace v gramatice

Nechť $G = (N, \Sigma, P, S)$ je gramatika a nechť $\lambda, \mu \in (N \cup \Sigma)^*$. Mezi λ, μ platí binární relace $\Rightarrow$ zvaná přímá derivace, můžeme-li řetězce λ a μ vyjádřit ve tvaru: $\lambda = \chi\alpha\delta$, $\mu = \chi\beta\delta$, kde $\chi, \delta \in (N \cup \Sigma)^*$ a $\alpha \to \beta \in P$. Pak píšeme $\lambda \Rightarrow \mu$. Relace $\Rightarrow^+$ označuje tranzitivní uzávěr relace $\Rightarrow$. Relace $\Rightarrow^*$ označuje tranzitivní a reflexivní uzávěr relace $\Rightarrow$.

Poznámka

Pokud v gramatice G existuje nonterminál $A \in N$, pro který platí: $A \Rightarrow^+ A$, potom řekneme, že gramatika G obsahuje cyklus.

2.1.3.3 Definice jazyka generovaného gramatikou G

Nechť $G = (N, \Sigma, P, S)$ je gramatika. Potom jazyk generovaný gramatikou G, $L(G)$, je definován jako:

$$L(G) = \{w : w \in \Sigma^* \wedge S \Rightarrow^* w\}$$

Předchozí kapitoly jsou citovány v [2].

2.1.3.4 Definice překladové párové gramatiky

Překladová párová gramatika G je pětice $G = (N, \Sigma, \Delta, P, S)$ kde:

- N je neprázdná konečná množina nonterminálních symbolů,
- Σ je konečná množina vstupních terminálních symbolů,
- Δ je konečná množina výstupních terminálních symbolů,
- P je konečná množina pravidel takových, že $A \to \alpha \mid \beta$, kde $A \in N$, $\alpha \in (N \cup \Sigma)^*$, $\beta \in (N \cup \Delta)^*$ a nonterminální symboly v řetězci β jsou permutací nonterminálních symbolů z řetězce α.
- S je startovní nonterminální symbol.

2.1.3.5 Definice derivace v PPG

Nechť $G = (N, \Sigma, \Delta, P, S)$ je překladová párová gramatika a nechť $\lambda, \mu \in (N \cup \Sigma)^*$ a $\sigma, \tau \in (N \cup \Delta)^*$ a nonterminální symboly v λ, μ jsou permutací nonterminálních symbolů z σ, τ a nechť $A \to \alpha \mid \beta \in P$. Potom $\lambda A\mu \mid \sigma A\tau$ přímo derivuje $\lambda\alpha\mu \mid \sigma\beta\tau$ kterou zapisujeme jako $\lambda A\mu \mid \sigma A\tau \Rightarrow \lambda\alpha\mu \mid \sigma\beta\tau$. Relace

$\Rightarrow^+$ označuje tranzitivní uzávěr relace $\Rightarrow$. Relace $\Rightarrow^*$ označuje tranzitivní a reflexivní uzávěr relace $\Rightarrow$.

2.1.3.6 Definice překladu pomocí PPG

Nechť $G = (N, \Sigma, \Delta, P, S)$ je překladová párová gramatika. Potom překlad definovaný párovou gramatikou G, $T(G)$, je definován jako:

$$T(G) = \{(w_1, w_2)\ w_1 \in \Sigma^*,\ w_2 \in \Delta^* \wedge S \Rightarrow^* w_1 \mid w_2)$$

2.1.3.7 LL(k) gramatika

Aby byla gramatika typu $LL(k)$, musíme být schopni rozpoznat pravidlo při znalosti prvních k symbolů derivovaných z jeho pravé strany. První L označuje, že se vstupní text zpracovává zleva doprava, druhé L představuje vytváření levého rozkladu.

2.1.3.8 LR(k) gramatika

Aby byla gramatika typu $LR(k)$, musíme být schopni rozpoznat pravou stranu přepisovacího pravidla, přičemž je nám známo vše, co lze z této pravé strany derivovat a navíc ještě k dalších vstupních symbolů. L označuje, že vstupní řetězec se zpracovává zleva doprava, R představuje vytváření pravé derivace v opačném pořadí.

Z toho plyne, že *LR* gramatiky popisují mnohem více jazyků, než *LL* gramatiky.

2.1.4 Překladač

Překladač je obvykle program, který čte zdrojový program a převádí ho do ekvivalentního cílového programu. Zdrojový program je napsaný ve zdrojovém jazyce, cílový program je v cílovém jazyce. Důležitou částí tohoto procesu překladu jsou diagnostické zprávy, kterými překladač informuje uživatele např. o přítomnosti chyb ve zdrojovém programu.

Musí provádět dvě základní činnosti: analyzovat zdrojový program a vytvářet k němu odpovídající cílový program. Analýza spočívá v rozkladu zdrojového programu na jeho základní součásti, na základě kterých se během analýzy vybudují moduly cílového programu.

Analýza zdrojového programu při překladu probíhá na následujících třech úrovních:

- Lexikální analýza: Zdrojový program vstupuje do procesu překladu jako posloupnost znaků. Tato posloupnost se čte lineárně zleva doprava a sestavují se z ní lexikální symboly jako konstanty, identifikátory, klíčová slova nebo operátory.
- Syntaktická analýza: Z posloupnosti lexikálních symbolů se vytvářejí hierarchicky zanořené struktury, které mají jako celek svůj vlastní význam. Např. výrazy, příkazy, deklarace nebo program.

- Sémantická analýza: Během sémantické analýzy se provádějí některé kontroly, zajišťující správnost programu z hlediska vazeb, které nelze provádět v rámci syntaktické analýzy. Např. kontrola deklarací, typová kontrola.

Při implementaci překladače se obvykle používá jednoho ze dvou základních přístupů. Překlad shora dolů nebo zdola nahoru. Tyto názvy odpovídají postupu při vytváření derivačního stromu.

Při překladu shora dolů vycházíme ze startovního symbolu gramatiky a snažíme se postupnou expanzí nonterminálních symbolů dospět až k terminálním symbolům odpovídajícím posloupnosti lexikálních symbolů ze vstupu. Tomuto postupu odpovídají gramatiky typu LL.

Při překladu zdola nahoru se naopak snažíme posloupnost terminálních symbolů ze vstupu redukovat až na startovní nonterminál. Tomuto postupu odpovídají gramatiky typu LR.

Nedílnou součásti překladače je tabulka symbolů, která zaznamenává identifikátory a jejich atributy použité ve zdrojových programech.

Předchozí kapitoly jsou citovány v [1].

2.2 Procesory a jejich architektury

Abychom mohli správně jazyk ISAC využít popř. rozšířit, je nutné nastudovat historické, ale i současné a nově vznikající architektury procesorů. V této podkapitole si popíšeme jednotlivé základní typy.

2.2.1 Procesor

Procesor je základní část jakéhokoli systému, která provádí výpočty a řídí překlad i provádění instrukcí uložených v paměti systému. Jde o elektronický obvod s vysokou hustotou integrace. Někdy se setkáváme s pojmem mikroprocesor, což je synonymum.

Z matematického hlediska realizuje funkci mnoha vstupních proměnných, jejímž výsledkem jsou další proměnné výstupní. Tato transformace je definována programem, což je posloupnost řídících pokynů pro vykonání konkrétních činností, které je procesor schopen realizovat. Tyto pokyny se nazývají instrukcemi a jim asociované činnosti/operace jsou přesně definovány. Vstupem do procesoru chápeme data uložená v pamětech, ať jsou to paměti externí, interní, vyrovnávací (cache) nebo vstupní/výstupní porty apod. Výstup transformační funkce se taktéž zapisují do pamětí nebo se přenášejí se přes vstupní/výstupní porty

Dle typu instrukce rozdělujeme procesory na dva základní typy:

- Complex Instriction Set Computer (CISC) ,
- Reduced Instruction Set Computer (RISC).

CISC procesory jsou charakteristické velkým množstvím složitých instrukcí, které usnadňují programovaní a překlad z vyššího programovacího jazyka do jazyka symbolických instrukcí. Daní za to je doba provádění instrukce a velkou složitost procesoru. Instrukce jsou charakteristické tím, že jsou tvořeny tzv. mikroinstrukcemi. Vykonání jedné instrukce je tedy posloupnost provedení mikroinstrukcí z čehož plyne, že se operace zpravidla provádí více taktů procesoru. Další vlastností je to, že instrukce umí přímo přistoupit do paměti a tam adresují operandy se kterými pracují.

RISC procesory vnikly jako reakce na velni se zvyšující složitosti procesoru typu CISC. Navíc prováděné analýzy zjistily, že v nemalém množství aplikací se z velké množiny instrukcí využívá sotva 25%. Instrukce které zpracovává RISC procesor jsou charakteristické jednoduchostí, defakto se jedná o vytažení mikroinstrukcí do jazyka symbolických instrukcí a zrušení operací skládající se z těchto mikroinstrukcí. Dále nepřistupují přímo do paměti (na přístup do paměti jsou rezervovány pouze dvě instrukce, *LOAD* a *STORE*). Díky jednoduchosti jsou instrukce prováděny za jeden takt procesoru a architektura tedy vybízí k řetězovému zpracování instrukcí (viz. následující kapitola). Nevýhodou je pak větší objem strojového kódu, protože na program o stejné funkci je potřeba více instrukcí.

2.2.2 Sériové a zřetězené zpracování instrukcí

Procesor může přistupovat ke zpracování instrukcí několika způsoby. Při klasickém sériovém zpracování instrukcí je dovoleno začít zpracovávat další instrukci až po kompletním zpracování předchozí instrukce (viz. tab. 1). Při snaze zvýšit výkonnost procesorů se používá metoda tzv. zřetězení instrukcí. Jde o rozdělení zpracování instrukce na separátní části. Podmínkou nutnou k této technologii je to, že každá část musí pracovat stejně rychle (nesmí se stát, aby načítání instrukcí bylo dvakrát rychlejší než provedení této instrukce). Zřetězené zpracování je znázorněno v tab. 2.

instrukce/takt	1	2	3	4	5	6	7	8	9	10	11	12
*i*1	*F*1	*F*2	*F*3	*F*4								
*i*2					*F*1	*F*2	*F*3	*F*4				
*i*3									*F*1	*F*2	*F*3	*F*4

Tab. 1: Naznačení sériového zpracování instrukcí

instrukce/takt	1	2	3	4	5	6	7	8
*i*1	*F*1	*F*2	*F*3	*F*4				
*i*2		*F*1	*F*2	*F*3	*F*4			
*i*3			*F*1	*F*2	*F*3	*F*4		
*i*4				*F*1	*F*2	*F*3	*F*4	
*i*5					*F*1	*F*2	*F*3	*F*4

Tab. 2: Naznačení zřetězeného zpracování instrukcí

Velkou nevýhodou zřetězeného zpracování jsou datové konflikty, které se u zpracování sériového nemůžou vyskytnout. Rozeznáváme několik typů konfliktů:

- Read After Write (RAW) – instrukce *i*2 je závislá na výsledku operace *i*1 a pokud by nepočkala, pracovala by s neaktuálními daty,

```
i1:  mul r3, r2, r1
i2:  add r5, r3, r2
```

- Write After Read (RAW) – instrukce *i*1 je složitá operace a trvá např. 5 taktů, *i*2 čeká na výsledek *i*1, ale operace *i*3 je rychlejší jež *i*1 a do registru zapíše dříve,

```
i1:  div r3, r2, r1
i2:  add r5, r3, r2
i3:  add r3, r4, r2
```

- Write After Write (WAW) – instrukce *i*1 zapíše až po instrukci *i*2, čímž přepíše výsledek následující instrukce.

```
i1:  div r3, r2, r1
i2:  add r3, r4, r5
```

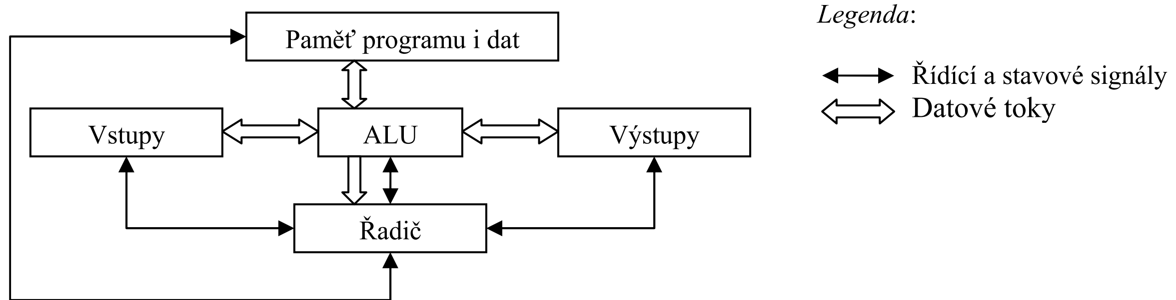
Základním řešením RAW je pozastavení linky do té doby, než jsou připravena další data. Existují i jiné metody jak se vypořádat s datovými konflikty, ale jejich popis přesahuje rozsah této práce. WAR a WAW se odstraňují pomocí přejmenování registru, se kterými instrukce pracují. Přejmenovávání zajišťuje buď překladač nebo přímo hardware v zřetězené lince.

2.2.3 Architektury

Před vlastním navrhováním procesoru je nutné si rozmyslet, jakými směry se bude návrh ubírat, jak budou rozděleny paměti, nebo jak budou uspořádány výpočetní jednotky. Podle FALTÝNKA [3] lze rozdělit architektury do následujících koncepcí a typů.

2.2.3.1 Von Neumanova koncepce

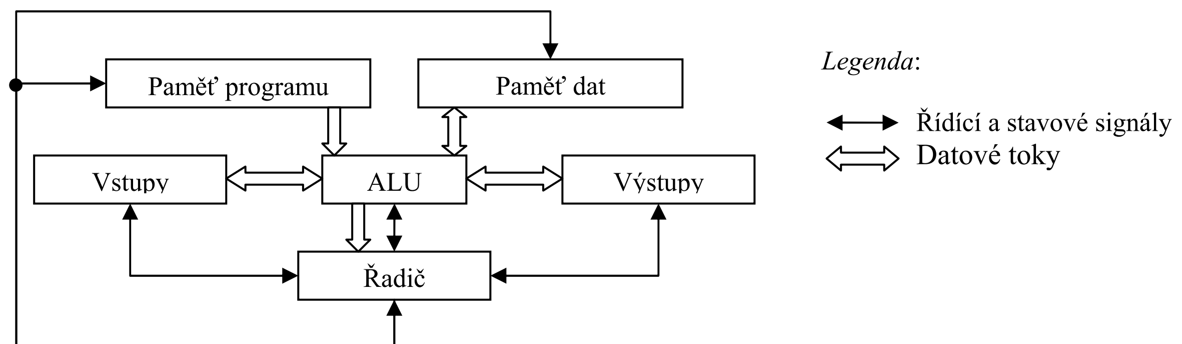
Při tomto uspořádání má procesor k dispozici jednu hlavní paměť, která obsahuje jak program (instrukce), tak data programu. Prolínání těchto dvou druhů dat v jedné paměti může být bráno jako výhoda i jako nevýhoda. Principiálně je totiž možné za běhu programu provádět přímé změny v kódu programu. Pokud jsou tyto změny záměrné a cílené, může to být jedna z technik, kterou lze obohatit možnosti programování daného procesoru. V horším případě může k těmto změnám docházet nezáměrně špatně navrženým programem.



Obr. 2: Von Neumanova koncepce

2.2.3.2 Harvardská koncepce

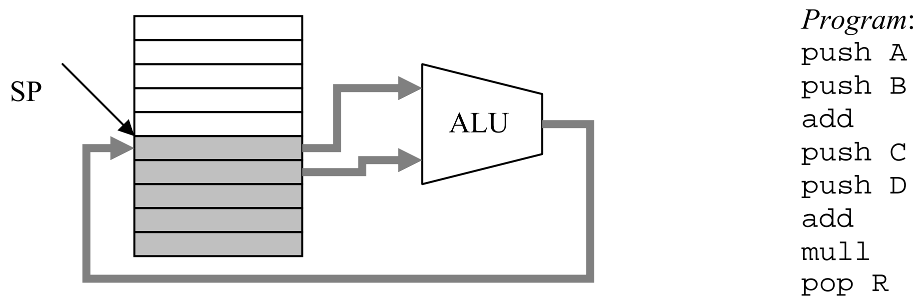
Při tomto uspořádání má procesor k dispozici dvě paměti (resp. dvě oddělené paměťové části). Paměť programu je určena pouze ke čtení a paměť dat je pak dostupná ke čtení i k zápis. Při použití Harvardské koncepce tedy odpadají možnosti modifikace programové paměti při běhu programu. Striktní oddělení programu a dat je chápáno jako výhoda, protože jsou přesně definovány a navzájem odděleny datové cesty, kterými jsou vedena instrukční nebo datová slova. Toto členění napomáhá jednoduššímu návrhu cílového procesoru.



Obr. 3: Harvardská koncepce

2.2.3.3 Zásobníková architektura

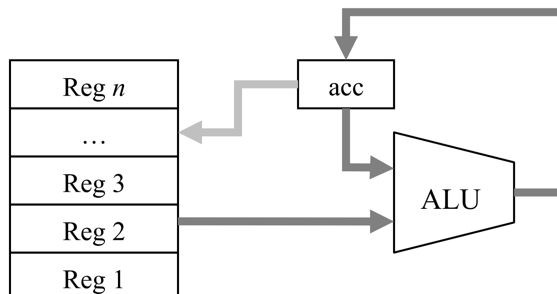
Co do využití hardwarových zdrojů je to nejefektivnější výpočetní architektura. Pro operandy a výsledek využívá zásobníku. Dva záznamy na vrcholu zásobníku jsou výpočetní jednotkou vyjmuty (`POP`), operace se provede a výsledek je uložen do zásobníku (`PUSH`). Tento způsob je realizací tzv. postfixové notace, kdy operační znaménko následuje až za uvedením zdrojových operandů.



Obr. 4: Zásobníková architektura

2.2.3.4 Akumulátorová architektura

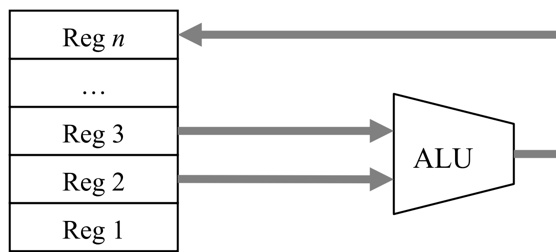
K dispozici máme konečný počet paměťových buněk, registrů. Tato architektura se snaží zmírnit potřebu rozsáhlé části instrukčního slova k adresování operandů. Jeden ze zdrojových operandů je vždy určen implicitně a je umístěn v tzv. akumulátoru, což je další registr se speciálním významem a zapojením. Akumulátor slouží zároveň jako registr výsledku. Díky tomu bude v instrukčním kódu potřeba adresovat pouze jeden zdrojový registr. Nevýhodou je naopak složitější programování – program je třeba psát tak, aby byl před každou operací jeden zdrojový operand v akumulátoru a zároveň počítat s akumulátorem jako výsledkem.



Obr. 5: Akumulátorová architektura

2.2.3.5 Registrová architektura

U této architektury máme opět k dispozici konečný počet registrů, ve kterých jsou uloženy zdrojové operandy operace. Výsledek bude uložen opět do jednoho z registrů. Každá instrukce může použít libovolné registry ke své činnosti. Registrová architektura vede k jednoduššímu programování, protože dostatečný počet registrů umožní ukládat proměnné výpočtu lokálně. Šetří se tím také čas, který by byl jinak potřeba pro přístup do paměti. Nevýhodou registrové architektury je nutnost explicitního uvedení identifikace registrů v instrukčním kódu, díky čemuž narůstá potřebná délka kódu instrukční slova.



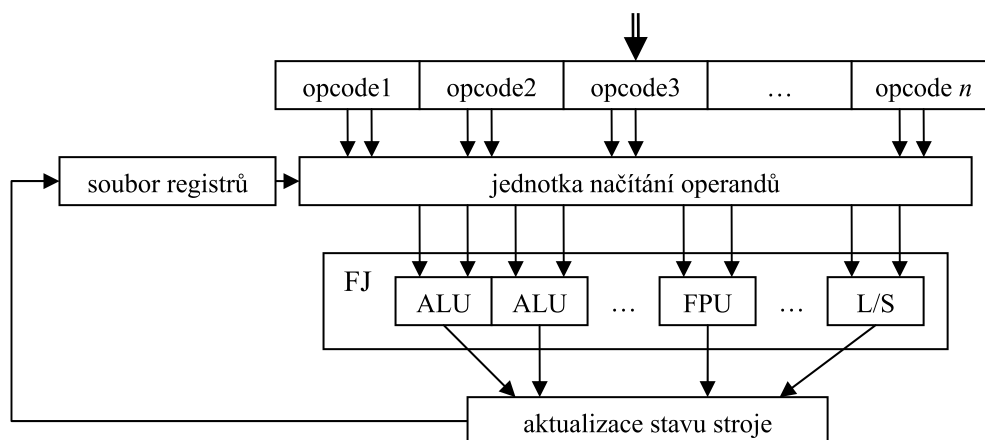
Obr. 6: Registrová architektura

Předchozí podkapitoly vychází z [3].

2.2.3.6 VLIW architektura

Ve všech předchozích architekturách se nacházela pouze jedna aritmetickologická jednotka, stejně tak je pouze jedenkrát zastoupena jednotka pro práci s paměti (L/S). Jestliže architektura obsahuje i jednotku pro operace o pohyblivé čárce (FPU), tak ji obsahuje také jen jednou. Pokud se v programu objeví více po sobě jdoucích instrukcí patřících jedné jednotce, musí čekat na sekvenční dokončení. Jistým zrychlením je zřetězení prováděných operací. Výpočetní jednotka však může v jednom taktu provést jen jednu instrukci.

Architektura VLIW se snaží zvýšit počet operací provedených v jednom taktu tím, že duplikuje výpočetní jednotky – takový procesor může mít tedy 4xALU, 2xFPU atd. Při kompilaci programu se naplánují operace pro příslušné jednotky s ohledem na datové závislosti a vytvoří se dlouhé instrukční slovo. To se rozčlení na jednotlivé části, které se následně předávají jednotkám ke zpracování. Plánování tedy provádí kompilátor, na rozdíl od předešlých architektur, kde se plánování provádělo až za běhu programu.



Obr. 7: VLIW architektura

2.3 Základní pojmy ze simulací

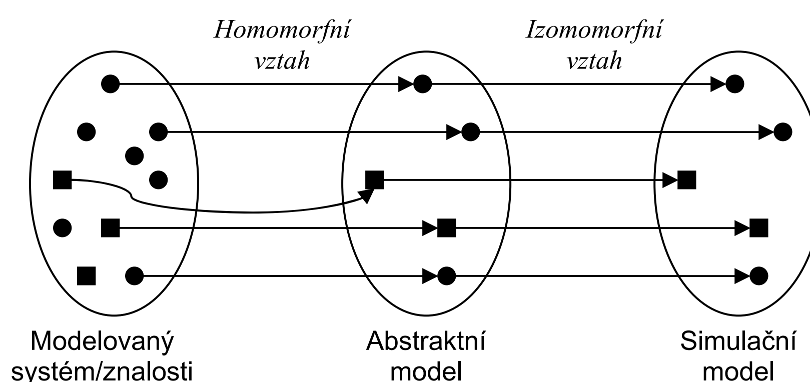
Protože výstup této diplomové práce, tedy disassembler, vytváří jádro simulátorů v projektu Lissom, objasněme si všechny jevy, které se simulací obecně souvisí.

2.3.1 Simulační model

Simulační model je abstraktní model zapsaný v libovolném programovacím jazyce. Při vytváření abstraktního modelu dochází ke ztrátě informací. Není možné, aby byl v izomorfním vztahu k reálnému prostředí. Tedy vybereme pouze takové vlastnosti, které jsou relevantní k míře abstrakce vytvářeného modelu. Naopak, simulační model již musí být v izomorfním vztahu k abstraktnímu modelu. Obecný proces vytváření simulačního modelu je znázorněn na následujícím obrázku.

Podle charakteru programovacího jazyka, v němž je napsán simulační model, ho rozdělujeme na:

- diskrétní model (časový krok simulace je o určité, konečné velikosti),
- spojitý model (časový krok simulace je nekonečně krátký okamžik).



Obr. 8: Vztah mezi modelovaným systémem a simulačním modelem

V projektu Lissom se pak homomorfní vztah uplatní tak, že na úrovni instrukčního nebo kompilovaného simulátoru (viz. kap. 8.5) neuvažujeme hardwarové aspekty procesoru, stejně jako teplotu nebo prostředí, ve kterém je nasazen.

Simulační model je pak diskrétního charakteru, neboť krok je dán tikem procesoru nebo je jako jeden krok bráno provedení jedné instrukce.

2.3.2 Simulace

Simulaci chápeme jako proces experimentování se simulačním modelem (tedy transformaci vstupních veličin a modelu na veličiny výstupní). Jejím cílem je analýza chování modelovaného systému v určitých situacích v závislosti na parametrech, které jsou modelu předány.

Simulace se zakládá na opakovaném provádění procesu a vyhodnocování výsledků. Pokud je simulační model validní vzhledem k skutečnému systému, dostáváme ve výsledku popis chování

modelovaného objektu aniž bychom museli daný model fyzicky realizovat. Toto má velmi podstatný význam, protože kolekce experimentů může odhalit chyby, které jsou v návrhu zařízení, pro které se simulační model vytvářel. V důsledku to může vést ke snížení nákladů na výrobu a provoz produktu, protože se do výrobního procesu dostane již odladěný návrh. Díky simulaci také můžeme vyzkoušet chování reálného systému, které se v praxi zkusit nemůže nebo nesmí zkusit (výbuch atomové bomby a dopad na civilní obyvatelstvo atp.).

Není neobvyklé, že se v průběhu experimentů ukáže, že simulační model přesně neodpovídá modelovanému systému. V takovém případě je nutné model přepracovat a opět musíme pomocí série experimentů ověřit jeho validitu.

2.3.3 Simulátor

Simulátor je realizace simulačního modelu. Realizací můžeme chápat buď fyzickou reprezentaci simulačního modelu (různé trenažéry atp.), nebo jako vytvoření spustitelné aplikace. Jak bylo uvedeno výše, simulační model je zapsán v libovolném programovacím jazyce. Pokud k tomuto jazyku existuje překladač, simulátorem je pak aplikace, která vznikne překladem. V této práci budeme realizací rozumět druhou z uvedených možností.

2.4 Zpětné inženýrství

Pod pojmem zpětné inženýrství (z angl. reverse engineering) rozumíme proces, který má dva základní cíle:

- odhalit technologické principy zařízení, jednotlivé komponenty systému a vztahy mezi nimi,
- vytvořit reprezentaci systému v jiné formě nebo vyšší úrovni abstrakce.

Mezi obvyklé postupy patří např. měření a/nebo analýza struktury. Cílem je tedy porozumění systému, který tak můžeme snadněji systém upravovat, vylepšovat nebo celkově přepracovat. Charakter postupu je inverzní ke klasickému schématu softwarového inženýrství. V následujícím textu tuto obecnou definici upřesníme.

Zpětné inženýrství v informatice (někdy označováno angl. výrazem reverse code engineering) budeme chápat jako postup získávání zdrojových kódů aplikace z jejího binárního obrazu, nebo z jejího programu v jazyce symbolických instrukcí. Ke zdrojovým kódům se dostáváme dvěma způsoby:

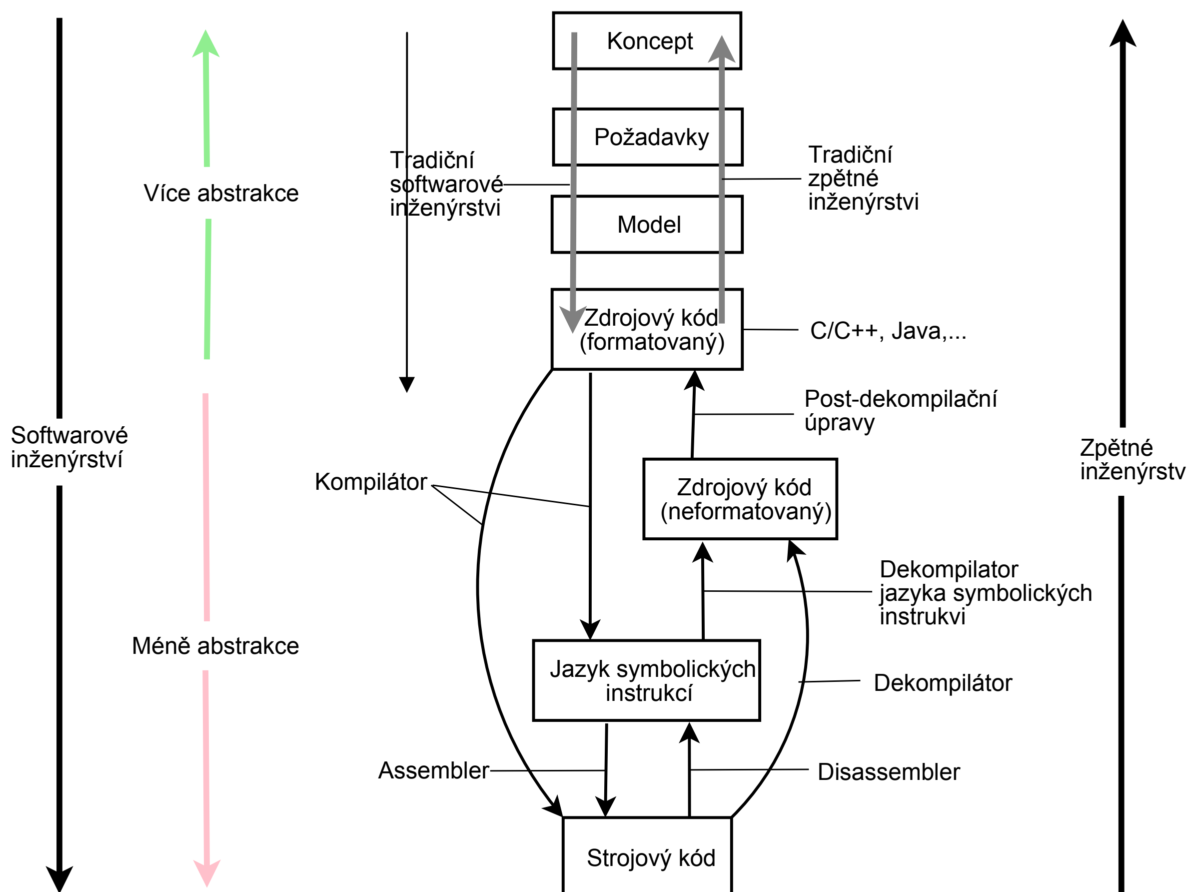
- pomocí disassembleru, který na vstupu přijímá binární obraz aplikace a generuje program v jazyce symbolických instrukcí,
- pomocí dekompilátoru, který na vstupu očekává binární obraz aplikace (např. bytecode jazyka Java), nebo program v jazyce symbolických instrukcí, a generuje program ve vyšším programovacím jazyce.

V prvém případě je velmi obtížné se v získaných informacích orientovat, protože zdrojový kód je ochuzen o komentáře, originální jména identifikátorů jsou ztraceny a nahrazeny identifikátory, které nejsou vhodné (obvykle disassembler přidává jen počitadlo k obecnému názvu „label_“). V případě dekompilace je situace lepší pouze v případě, že vycházíme z programu jazyka symbolických instrukcí. Zde mohou být komentáře uloženy, stejně jako jména identifikátorů. Obecně se dá říct, že výsledek dekompilace je pro člověka čitelnější (konstrukce větvení nebo cyklů jsou hůře čitelné v JSI než ve vyšším programovacím jazyku).

Důvod, proč se chceme dostat ke zdrojovým kódům, může být dvojí:

- zdrojové kódy k binárnímu obrazu aplikace máme, ale jsou nevhodné a chceme vytvořit nové na jiné úrovni abstrakce nebo v jiném jazyce,
- zdrojové kódy k binárnímu obrazu aplikace nemáme, a proto je chceme vytvořit (nečastější případ, pojem reverse code engineering je mnohdy spojován pouze s tímto bodem, nikoliv i s bodem předchozím).

Výše uvedené je znázorněno na obr. 9.



Obr. 9: Obecné schéma softwarového a zpětného inženýrství

2.4.1 Disassembler

Disassembler, neboli zpětný assembler, je program, který převádí strojový kód do jazyka symbolických instrukcí. Používá se pro ladění aplikací, jako součást simulačních nástrojů a k dalším účelům. Je tedy zřejmé, že disassembler je stejně jako assembler silně závislí na typu procesoru a taktéž není přenositelný mezi různými architekturami.

2.4.2 Dekompilátor

Dekompilátor je program, který transformuje program na relativně nízké úrovni abstrakce do vyšší úrovně abstrakce. Jde tedy o inverzní postup ke kompilaci. Takto získaný výstup je funkčně ekvivalentní se vstupem, ale je nutné si uvědomit, že textový výsledek nemusí vždy přesně odpovídat originálnímu zdrojovému kódu. Při kompilaci může totiž docházet k optimalizacím (ve výsledku dostáváme značně odlišný výstup vzhledem ke vstupu) dále ke ztrátě informací ve formě odstranění komentářů, navíc jsou jména identifikátorů nahrazena adresami. Toto se děje především při kompilaci do strojového kódu. Pokud kompilujeme z nějakého důvodu z vyššího programovacího jazyka do jazyka symbolických instrukcí, tyto informace obyčejně zůstávají zachovány ve formě ladicích informací a není tedy problém se k nim dostat. V takovém případě je výsledný zdrojový kód po dekompilaci velice podobný originálu.

2.5 Metody disassemblování

Jak již bylo v předchozí kapitole řečeno, disassembler transformuje program z binárního kódu do jazyka symbolických instrukcí. Co však celý proces disassemblování ztěžuje, je vlastní architektura procesoru, způsob zarovnání v paměti (viz kap. 2.5.1) nebo relokační údaje (viz kap. 2.5.2). Metod, jak se z výše popsanými problémy vyrovnat, se v praxi používá několik. Ty nejčastější si popíšeme v následující části. Před ní se však zmíníme o příčinách, které nejčastěji způsobují problémy ztěžující disassemblování.

2.5.1 Zarovnání paměti

Dnešní procesory vyžadují, aby proměnné a datové objekty byly umístěny v paměti na adresách, které jsou dělitelné celočíselným offsetem. Např. procesor řady x86 vyžaduje, aby proměnná typu *int* (velikost 4 byty) byla uložena na adresách dělitelných čtyřmi. Povolené adresy jsou tedy např. `0xf000`, `0xf004` atp., žádná jiná povolená není.

Pokud ukládáme datový objekt, který je buď menší než *int* (např. *char* pro vyjádření jednoho znaku), nebo větší než *int* (struktura v jazyce C, která obsahuje různě velké datové typy), doplní kompilátor zbylé byty nulovou hodnotou, čímž dojde k zarovnání paměti.

Adresa	Obsah	Symbolický zápis	
…	…	…	
0x1e3df000	61 00 00 00	**char** ch = 'a';	
…	…	…	
0x1e3dff04	ee 43 75 03	**int** ii = 0x37543ee;	**struct** s
0x1e3dff08	62 61 00 00	**char** kk[2] ='ab';	
0x1e3dff0b	45 12 00 10	**int** ll = 0x10001245;	
…	…	…	

Tab. 3: Znázornění zarovnání v paměti

Uvažujeme-li strukturu uvedenou v tabulce, očekávali bychom, že velikost, kterou data v paměti zabírají, je 4+2+4=10 bytu. To však architektura neumožňuje, a kompilátor musí zařídit zarovnání, a tedy celková zabraná paměť je 12 bytu. Obdobně je tomu tak i u jednoduchého typu *char*.

2.5.2 Relokační údaje

Po překladu programu assemblerem neobsahuje výstupní binární soubor u adres symbolů, cílů skoků nebo u adresy celé sekce[2] konečné hodnoty. Namísto toho jsou zde jen informace pro linker, že má adresu dopočítat až by vytvářet spustitelný program na dané platformě. Způsob dopočítávání je dvojí:

- Absolutní – používá se adresa ve tvaru nezáporného celého čísla. Tato adresa se uvažuje vždy v rámci jediné sekce a je počítána od jejího počátku.

Sekce 1 *(absolutní)*	
Adresa	**Operace**
…	…
0x00e310	label1:
…	…
0x00e324	jmp label1<0x00e310>
…	
0x00e329	jmp label2<????>
Sekce 2 *(relativní)*	
…	…
0x000010	label2:
…	…
0x000020	jmp label2<0x000010>
…	…

Sekce 1 + Sekce 2	
Adresa	**Operace**
…	…
0x00e310	label1:
…	…
0x00e324	jmp label1<0x00e310>
…	
0x00e329	jmp label2<0x00e340>
…	…
0x00e340	label2:
…	…
0x00e350	jmp label2<0x00e340>
…	…

Tab. 4: Absolutní relokace symbolů

- Relativní – používá se adresa ve tvaru celého čísla se znaménkem. Adresa symbolu je počítána vždy relativně od aktuální polohy v paměti, na které leží odkaz na tento symbol.

---

[2] Sekce rozdělujeme na tři typy: kódová sekce obsahující program, sekce pro inicializovaná data a dále sekce pro data.

Sekce 1 *(absolutní)*	
Adresa	**Operace**
…	…
0x00e310	label1:
…	…
0x00e324	jmp label1<-14>
…	
0x00e329	jmp label2<????>
Sekce 2 *(relativní)*	
…	…
0x000010	jmp label1: <????>
…	…
0x000020	jmp label2<20>
…	…
0x0000040	label2:
…	…

Sekce 1 + Sekce 2	
Adresa	**Operace**
…	…
0x00e310	label1:
…	…
0x00e324	jmp label1<-14>
…	
0x00e329	jmp label2<40>
…	…
0x00e340	jmp label1: <-30>
…	…
0x00e350	jmp label2<20>
…	…
0x00e370	label2:
…	…

Tab. 5: Relativní relokace symbolů

Obou principů se využívá hojně při sestavování výsledného programu, který se skládá z více modulů, nebo se v jednom programu odkazujeme na symbol, který je obsažen v jiném. Nejvíce se těchto principů využívá u dynamických knihoven.

2.5.3 Základní metody

Mezi základní metody se řadí metoda lineárního průchodu a metoda rekurzivního sestupu. Obě jsou základními prvky, které následně využívají další, v praxi rozšířenější a vylepšené metody.

2.5.3.1 Metoda lineárního průchodu

Je založena na přímém průchodu a disassemblaci všeho, co se vyskytuje v kódových sekcích, tedy v sekcích, které obsahují kód aplikace a statická data. Algoritmus je naznačen na konci této kapitoly a jako parametr bere první byte kódové sekce. Tato metoda je díky své povaze nejrychlejší a nejjednodušší na implementaci. Je ovšem také nejvíce chybová. Problémem jsou uložená data a jejich zarovnání v paměti, které může být vynuceno jejich charakterem. V tomto případě disassembler nemusí zachytit správný začátek nové instrukce a dojde k nesmyslnému dekódování dat, které metoda vnímá jako kód aplikace.

Tuto situaci lze detekovat jedině tím, že se na vstupu objeví strojový kód, který není disassembler schopný přeložit zpět do jazyka symbolických instrukcí. Poté dojde k pokusu o zotavení tím, že se pokusí nalézt nejbližší správný začátek instrukce a disassemblace pokračuje od tohoto místa dále. Je patrné, že nejsme schopni odhalit, jak dlouho a kolik binárních dat jsme špatně přeložili, což je hlavní nevýhoda této metody.

Jako metodu ji využívají programy jako je *objdump*.

```
void linear_sweep(int addr)
{
	while (statr_addr <= addr <= end_addr)
	{
		intr = decode(addr);
		addr += instr.length;
	}
}
```

Alg. 1: Schématický zápis algoritmu lineárního průchodu

2.5.3.2 Metoda rekurzivního sestupu

Metoda rekurzivního sestupu se snaží odstranit chyby v lineárním průchodu tím, že nepřekládá vstupní soubor byte po bytu, ale bere v potaz řídicí instrukce. Tedy disassemblace vždy pokračuje od cíle skoku přeložené větvicí konstrukce. Větvicí konstrukcí rozumíme skokovou instrukci nebo instrukci volání podprogramu. Tím máme zajištěno, že disassembler nikdy nedostane na vstup data popř. zarovnání dat.

Problém zde ale vyvstává s částmi kódu, do kterých se větvícími instrukcemi nedostaneme, protože podmínka skok nikdy nenabude patřičné hodnoty. Dále s těmi částmi kódu, které jsou dosažitelné až po vyhodnocení podmínky, která očekává nějaká vstupní data (pokud by se měly podmínky vyhodnocovat real-time za překladu do jazyka symbolických instrukcí, znamenalo by to implementovat velkou logiku navíc a tedy nepřípustné zpomalení). Nebo cíle skoků ještě nejsou známy, protože neproběhlo linkování. Tyto části zůstávají nepřeloženy a pro algoritmus jsou nedosažitelné.

Algoritmus je naznačen na alg. č. 2, kde disassembler bere jako parametr vstupní bod programu.

```
void recursive_descent(int addr)
{
	while (is_valid(addr))
	{
		if (addr.visited)
			return;
		intr = decode(addr);
		addr.visited = true;
		if (is_branch(instr))
		{
			for (each target t of instr)
				recursive_descent(t);
		}
		else
			addr += instr.length;
	}
}
```

Alg. 2: Schématický zápis algoritmu rekurzivního sestupu

2.5.4 Pokročilé metody

Každá z výše popsaných metod na svoje výhody a slabiny. Buď řesíme problém s daty uloženými v kódové sekci, ale dekódujeme všechno, nebo data úspěšně přeskakujeme, ale necháme potenciálně velkou část kódu nepřeloženou. Nyní si popíšeme metody, které se snaží spojit výhody obou a jejich úskalí co nejvíce potlačit.

2.5.4.1 Hybridní metoda

Hybridní metoda je založena na použití jedné ze základních metod právě v okamžicích, kdy se tyto metody chovají korektně. Jako první je použita metoda rekurzivního sestupu. Dekóduje se co možná nejvíce kódu, poté se metody vymění a na zbylé nedekódované části se použije metoda lineárního průchodu. Pak se řízení předá opět rekurzivnímu sestupu, ten se pokusí najít v nově dekódovaných částech další větvení, a tak se celý postup opakuje, dokud jsou nacházeny jiná nová možná větvení.

Existuje ještě upravená verze, kdy je metoda lineárního průchodu použita na celou kódovou sekci. Poté se výsledky obou metod porovnají, a v případě nesrovnalostí se dekódováni v problematických částech provede opětovně. Pokud ani tento postup nevede ke shodě metod, označí se tato část jako problematická a automaticky se již nedekóduje.

2.5.4.2 Disassemblery založeny na PPG a PKA

Disassemblery založené na párových překladových gramatikách nebo párových konečných automatech se snaží přistupovat k dekódování jako k formálnímu překladu jazyků. Program ve strojovém kódu je chápan jako věta gramaticky popsaného jazyka. Není zde použit mechanizmus deterministického dekodéru, jak tomu je u předchozích metod. Popis jednotlivých překladů je popsán v následujících kapitolách.

2.6 Další pojmy

2.6.1 Strojový kód

Strojový kód (někdy také binární kód nebo binární obraz aplikace) je vyjádření programu v elementární podobě, které rozumí procesor. Jedná se o binárně uloženou informaci jednotlivých instrukcí programu. Pro uživatele je nepřehledný a špatně čitelný.

2.6.2 Jazyk symbolických instrukcí

Jazyk symbolických instrukcí, neboli „assembly language“, je programovací jazyk nejnižší úrovně. Je tvořen zástupnými zkratkovými symboly, které zastupují binární reprezentaci instrukce. Se symboly se daleko lépe pracuje, jsou zapamatovatelnější a jimi napsaný program je čitelnější. Jelikož je assembler i binární reprezentace instrukce jsou vzhledem k procesoru na stejné úrovni abstrakce, je jasné, že assembler je závislý na typu procesoru a není přenositelný mezi různými architekturami.

2.6.3 Assembler

Assembler je překladač jazyka symbolických instrukcí do strojového kódu procesoru.

2.6.4 Linker

Linker vytvoří spustitelný program spojením více přeložených programů, modulů, ve strojovém jazyce a současně vyřeší vzájemné závislosti mezi moduly.

2.6.5 Vestavěný systém

Vestavěný systém je malý počítačový systém, který má v sobě vloženo/vestavěno zpracování informací, a jehož řídícím centrem je procesor. Takový systém může mít za úkol regulovat fyzikální veličiny, měřit fyzikální veličiny, také může řídit zobrazovací jednotky jako jsou displeje apod. Z tohoto důvodu má typicky omezené pole aplikovatelnosti právě na odvětví, pro které byl konstruován. Z pohledu uživatele často nebývá viděn a je součástí většího celku. Systém není jen procesor, ale celá škála dalších komponent, které slouží pro styk s okolím. Jedná se např. o paměti a

vstup/výstupní kanály, přerušení atd. Dále je nutné, aby splňoval vlastnosti dlouhodobé bezporuchovosti, měl by mít nízký příkon a další typické vlastnosti těchto systémů.

2.6.6 XML

XML, neboli eXtensible Markup Language, je obecný textový formát dat/jazyk bez sémantiky. Pracuje pouze s čistými daty. Pro člověka je formát čitelný a modeluje hierarchické struktury. Jeho výhoda je v tom, že není implementačně závislý na softwaru, ani na operačním systému.

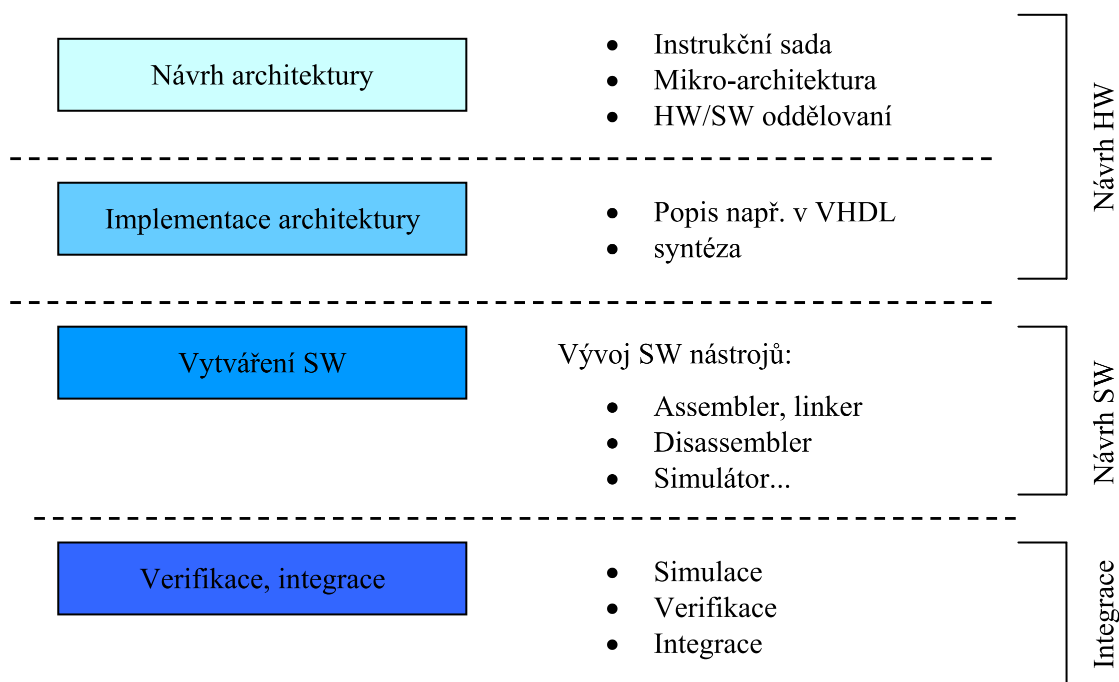
3 Vývoj a návrh procesorů

V současné době se návrh procesoru a jeho součástí provádí s velmi malou automatizací. Proces návrhu je dlouhý a vyžaduje velmi schopné a zkušené návrháře. Většina nově navrhovaných procesů využívá prostředky jak z vestavěných systémů, tak i z integrovaných obvodů. Návrhář vytvoří architekturu procesoru, odsimuluje ji a může vytvářet aplikace, které posléze integruje do systému. Každý krok vyžaduje své návrhářské prostředky a specializovaný tým odborníků.

Celý proces se dá v zásadě rozdělit do 4 fází (obr. 1):

- Navrhuje se architektura s ohledem na budoucí použití procesoru. Taktéž se vytváří instrukční sada a další základní bloky. Tento proces často probíhá několikrát za sebou, dokud se nedosáhne nejvhodnější kombinace jednotlivých bloků.
- Takto navržená architektura se transformuje na popis v jazyce, který umožňuje pozdější syntézu, výrobu procesoru. Typicky jde o jazyk, kterým lze popsat hardware, např. VHDL.
- Poté se vytvářejí softwarové prostředky, které umožní práci s procesorem. Jde především o assembler, disassembler, linker, debugger a kompilátor nějakého vyššího jazyka, např. C.
- V poslední fázi se vše integruje, simuluje a verifikuje.

V každé části procesu jsou využívány jiné modelovací prostředky a jiné jazyky pro popis daných částí. Tyto jazyky se také dají rozdělit do několika kategorii.



Obr. 10: Proces vývoje procesoru

3.1 Jazyky pro popis procesoru

Jazyky typu VHLD jsou pro popis architektury procesoru nevhodné, obsahují příliš mnoho zbytečných informací ohledně hardwarových vztahů jednotlivých částí procesoru. Naopak chybějí informace např. o jazyku symbolických instrukcí.

Proto je vhodnější použít jazyky, které jsou komplexnější a disponují automatizovanými postupy v jistých fázích vývoje procesoru.

3.1.1 Jazyky zaměřené na popis architektury

Zaměřují se na strukturu součástí a jejich zapojení do architektury procesoru. Tyto závislosti jsou často zobrazovány blokovým diagramem. Z tohoto popisu lze tak provést syntézu procesoru. Jsou to např. MIMOLA, COACH.

3.1.2 Jazyky zaměřené na popis instrukční sady

Popisují procesor pomocí instrukční sady. Jinými slovy, je to pohled na procesor z programátorského pohledu bez popisu obvodů, či jiných hardwarových části. Jako zástupce můžeme jmenovat nML, Valen-C.

3.1.3 Jazyky zaměřené na popis instrukční sady a architektury

Přináší výhodný vztah mezi prvně jmenovanými jazyky, které jsou až příliš detailní pro rychlou interpretační simulaci, a pro generovaní assembleru a podobných součástí, a jazyky pro popis instrukční sady, které neumožňují při simulaci využívat cykly procesoru nebo simulovat pipelined systémy. Mezi tyto jazyky patří RADL, LISA.

Kapitola 3 byla citována v [4].

4 Cíl a metodika práce

Cílem této práce je zhodnotit stávající způsob, dle kterého je vytvářen disassembler, najít a identifikovat slabá místa, a vytvořit model nový, který bude stejně obecný, a přitom nebude mít záporné vlastnosti, které se vyskytují u stávajícího.

Je nutné se seznámit s podobnými projekty a jazyky pro popis procesoru, z nichž nejdůležitější je jazyk LISA, ze kterého vychází modelovací jazyk ISAC. Ten je v projektu Lissom používán. Při vývoji disassembleru se tedy budeme podílet i na jeho vývoji a upravování. Při práci je nutno pamatovat na co největší možné využití stávajících utilit, jako jsou vytvořené knihovny pro práci se vstupním formátem nebo knihovny navržené pro správu atributů.

K tomu, abychom mohli co nejlépe navrhnout a implementovat obecný disassembler, musíme prostudovat i jiné oblasti než ty, které byly doposud zmíněny.

Jedná se především o části z teorie formálních jazyků, a to především části týkající se bezkontextových a regulárních jazyků a jejich modelů. Nezbytné je i studium jejich možné transformace a implementace těchto transformací.

Musíme se seznámit nejen s assemblery a architekturami již existujících procesorů, ale také s nově vznikajícími VLIW ASIP procesory, které se stávají stále dominantnějšími na trhu v oblasti vestavěných systémů. V neposlední řadě se také jedná o obecné nové návrhové trendy v paralelních systémech. Ve všech případech hraje nejdůležitější roli instrukční sada, kódování instrukcí, a také způsob zpracování aritmetických operací.

5 Projekt Lissom

Projekt Lissom se zabývá vývojem jazyka pro popis instrukční sady a architektury, generováním softwarových částí procesoru a jejich simulací, vytvořením komplexního prostředí, ve kterém může uživatel modelovat a testovat procesory. Modelovacím jazykem je zde ISAC, který vychází z jazyka LISA.

5.1 Jazyk ISAC

Jazyk ISAC je nadstavbou nad libovolným vyšším programovacím jazykem, např. C/C++. Dalo by se také říci, že jde o preprocesor. Jazyk má dvě oddělené části, které mohou existovat samostatně:

- definici zdrojů a hardwarové části procesoru,
- definici operační části a chování.

V další části textu se budeme zabývat převážně druhou částí popisu procesoru. První část je využita jen pro získání informací o fyzických částech modelovaného procesoru. Jedná se především o paměťový model, tzn. kde je paměť programu, kde jsou data. Dalšími informacemi jsou pak registry, jako je např. program counter.

5.1.1 Popis zdrojů

Z popisu zdrojů jsou zde vybrány ty části, které se přímo týkají zpětného assembleru.

“**PROGRAM COUNTER**” *type id* “**;**”

- *type* vyjadřuje rozsah registru (int, short, bit)
- *id* je jméno registru

Od tohoto registru je odvozena šířka slova používaná pro načítání instrukcí pro dekódování.

“**MEMORY**” *type id* “**{**” *vlastnosti* “**}**”

- *type* vyjadřuje rozsah buněk paměti (int, short, bit)
- *id* je jméno paměti
- *vlastnosti* dodatečné informace, jako jsou flagy, velikosti, atp.

Paměti jsou používány k vytvoření výsledného paměťového modelu.

“**MEMORY MAP**” *id* “**{**”
“**RANGE**” “**(**” *from* “**,**” *to* “**)**” “->” *id_mem banked* “**;**”
{“**RANGE**” “**(**” *from* “**,**” *to* “**)**” “->” *id_mem banked* “**;**”}
“**}**”

- *id* je jméno mapování
- *from* je počáteční adresa v paměťovém modelu, na kterou se namapuje *id_mem*
- *to* je koncová adresa v paměťovém modelu, na kterou se namapuje *id_mem*
- *id_mem* je jméno paměti
- *banked* jsou další informace použité pro mapování (pro disassembler nepodstatné)

Paměťový model je použit při načítání z vstupního objektového souboru (viz kap. 6) pro kontrolu, jestli se nepokoušíme spouštět data.

5.1.2 Popis operační části jazyka ISAC

Na nejvyšší úrovni popisu operační části jsou dvě základní deklarace, a to deklarace skupiny a deklarace operace. Opět jsou zde ukázány konstrukce, které se přímo týkají zpětného assembleru.

Deklarace skupiny sdružuje operace s podobným významem. Tvar zápisu je následující:

"**GROUP**" *group_id* "=" *operation_id* { "," *operation_id* } ";"

- *group_id* je identifikátor skupiny
- *operation_id* je identifikátor operace nebo skupiny

Deklarace operace popisuje operaci, její formu v textové a binární podobě a její chování. Další vlastnosti, které zde nejsou zmíněny a přitom se v operaci dají popsat, slouží jen pro simulátory.

"**OPERATION**" *operation_id* "{" `[instances` ";"]
`assembler` ";"
`coding` ";"
`[behavior` ";"]
`[expression` ";"] "}"

- *operation_id* je identifikátor operace.

Sekce `instances` má tvar:

"**INSTANCE**" *instance_id* { "**ALIAS**" "{" *alias_id* { "," *alias_id* } "}" } ";"

- *instance_id* je identifikátor jiné operace nebo skupiny
- *alias_id* je identifikátor aliasu *instance_id*

Sekce `assembler` má tvar:

"**ASSEMBLER**" "{" {*instance_id* | *text* | `attribute`} ["{" *semantic_action* "}"] "}"

- *instance_id* je identifikátor operace, skupiny nebo aliasu
- *text* je textová konstanta

- *semantic_action* je sémantická akce spojená se sekcí `attribute`

Sekce `attribute` má tvar:

attr_left_id "=" *attr_type* ["=" *attr_right_id*]

- *attr_left_id* je levý identifikátor atributu
- *attr_type* určuje typ atributu. Může nabývat hodnot dle následující tabulky.

Typ	**Význam**
"#U"	desítkové číslo, bezznaménkové
"#S"	Desítkové číslo, znaménkové
"#SYMBOL"	Symbolický odkaz (návěští)

Tab. 6: Typy atributů v sekci `assember`

- *attr_rigth_id* je pravý identifikátor atributu

Sekce `coding` má tvar:

"**CODING**" "{" {*instances_id* | *binary* | `attribute`} ["{"se*mantic_action* "}"] "}"

- *instances_id* je identifikátor operace, skupiny nebo aliasu
- *binary* je binární konstanta
- *semantic_action* je sémantická akce spojená se sekcí `attribute`

Sekce `attribute` má tvar:

attr_left_id "=" *attr_type* ["=" *attr_right_id*]

- *attr_left_id* je levý identifikátor atributu
- *attr_type* určuje typ atributu. Může nabývat hodnot dle následující tabulky.

Typ	**Význam**
"0bx[*k*]"	číslo bez znaménka
"0bsx[*k*]"	číslo se znaménkem
kde *k* označuje bitové pole o *k* bitech	

Tab. 7: Typy atributů v sekci `coding`

- *attr_rigth_id* je pravý identifikátor atributu

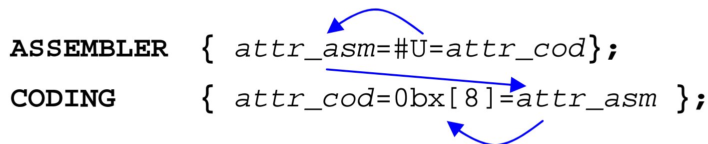
Sekce `behavior` je sekvence akcí popsaná v jazyku C, které vyjadřují chování operace. Tato sekce je využívána pro simulaci chování procesoru.

Sekce `expression` má podobný význam jako sekce `behavior`, avšak obsahuje pouze návratovou hodnotu. Hodnota může být konstanta, nebo hodnota atributu.

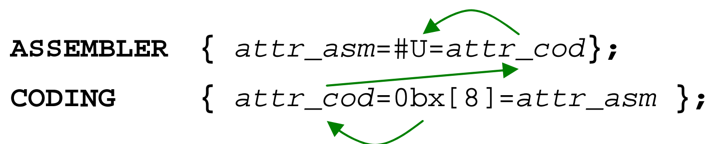
Poznámka:

Význam levých a pravých jmen atributů spočívá v obousměrné komunikaci. Můžeme chtít, aby se v sémantických akcích s atributy prováděly modifikace typu šiftování, připočtení offsetu. Tyto operace mohou být pro překlad z jazyka symbolických instrukcí do strojového kódu jiné, než pro překlad strojového kódu do jazyka symbolických instrukcí (typicky jsou k sobě inverzní). V každém směru je tedy nutné mít separátní komunikaci přes jedinečné jméno atributu. Naznačení komunikace je v dalším textu.

Ve směru generování strojového kódu:



Ve směru generování jazyka symbolických instrukcí kódu:



Pokud v žádném překladu neprovádíme s atributy dodatečné operace, stačí atribut pojmenovat jen levým jménem. Komunikace je pak mezi překlady shodná.

5.1.3 Překladač pro operační část jazyka ISAC

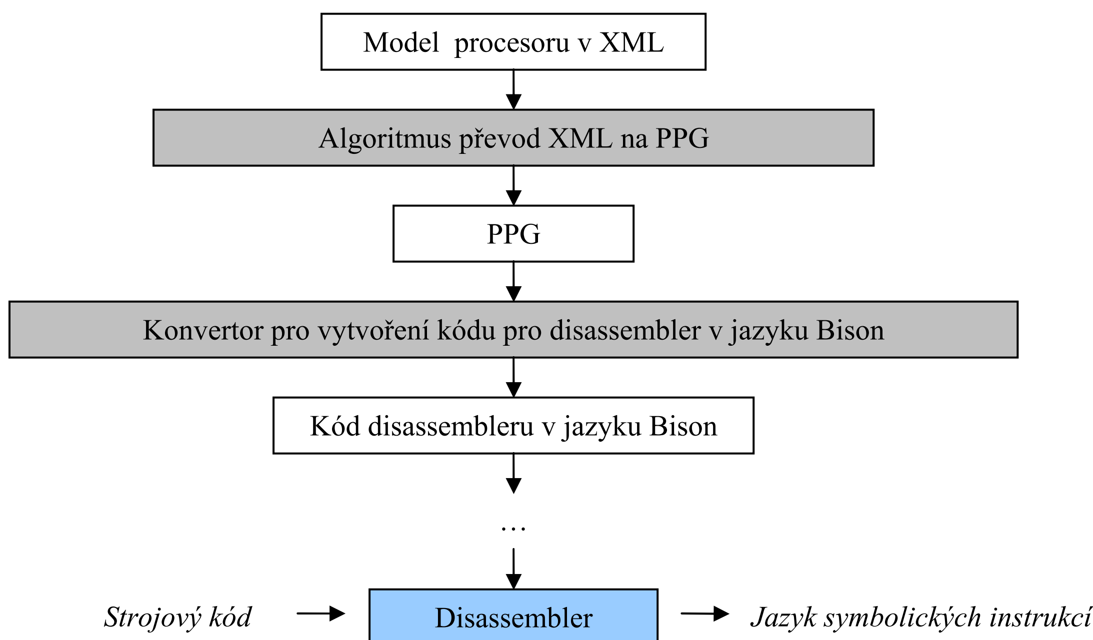
Pro operační část existuje překladač, jehož výstupem je model procesoru v XML. Výsledný model je brán jako vstup do dalších generátorů nástrojů, jako je např. generátor pro (dis)assembler nebo simulátor. Na vstupu těchto generátorů se již neprovádí žádná kontrola a předpokládá se, že vstup je korektní.

6 Schéma překladu popisu procesoru v XML na disassembler

V této kapitole zhodnotíme stávající metodu překladu, která využívá párové překladové gramatiky (PPG) a navrhneme metodu novou, využívající párové konečné automaty (PKA).

6.1 Stávající vnitřní model a metodologie vytváření disassembleru

Sávající metodologii znázorňuje obr. 11. Vnitřní model je založen na překladových párových gramatikách. V dalších podkapitolách této sekce ve stručnosti představíme tuto metodologii. Více informaci najdeme v [12].



Obr. 11:Obecné schéma převodu modelu procesoru v XML na disassembler pomocí PPG

6.1.1 Formální popis algoritmus pro převod XML na PPG

Vstupem pro algoritmus bude přeložený model procesoru v XML, výstupem bude párová překladová gramatika $G = (N, \Sigma, \Delta, P, S)$.

Nechť je gramatika *G,* inicializovaná takto:

$N = \emptyset$,

Σ = {"*0*", "*1*" },

Δ= {"*num*"},

$P = \emptyset$,

S = identifikátor operace nebo skupiny, která zastupuje všechny ostatní.

Pro každou sekci ve tvaru:

"**GROUP** X **=** M_1**,** M_2**,** …**,** M_n**;** ", kde $n > 0$

platí:

- Přidáme X, M_1, M_2, …, M_n do množiny N.
- Přidáme pravidla $X \rightarrow M_1 \mid M_1$, $X \rightarrow M_2 \mid M_2$ …, $X \rightarrow M_n \mid M_n$ do množiny P.

Pro každou sekci ve tvaru:

"**OPERATION** *X* **{** [**INSTANCE** *I*;]
ASSEMBLER *A*;
CODING *C*;
… **}**"

platí pro I:

- Pokud je I ve tvaru "*ID*"
  - Přidáme ID_X do množiny N.
  - Přidáme $ID_X \rightarrow ID \mid ID$ do množiny P.
- Pokud je I ve tvaru "*ID* **ALIAS {** A_1**,** A_2**,** …**,** A_m **}**", kde $m > 0$
  - Přidáme A_1_X, A_2_X, …, A_m_X do množiny N.
  - Přidáme $A_1_X \rightarrow ID \mid ID$, $A_2_X \rightarrow ID \mid ID$, …, $A_m_X \rightarrow ID \mid ID$ do množiny P.

platí pro A a C:

- Nechť je A ve tvaru "**{** A_1, A_2, …, A_v **}**", kde $v > 0$
  - Přidáme všechny $a \in A$, které jsou konstantním textem, do množiny Δ.
  - Nechť h_A je homomorfní zobrazení z A na $N \cup \Delta$ definované takto:

    $h_A(a) = a$ pro všechny $a \in A$, které jsou konstantním textem,

    $h_A(a) = a_X$ pro všechny $a \in A$, které jsou identifikátory operace nebo skupiny,

    $h_A(a)$ = "*num*" pro všechny $a \in A$, které jsou atributy.
- Nechť je C ve tvaru "**{** C_1, C_2, …, C_w **}**", kde $w > 0$
  - Přidáme všechny $c \in C$, které jsou binárními konstantami, do množiny Σ.
  - Všechny $c \in C$, které jsou bitovým polem, reprezentujeme terminálním symbolem "b_*k*" tak tuto reprezentaci přidáme do množiny Σ.
  - Nechť h_C je homomorfní zobrazení z C na $N \cup \Sigma$ definované takto:

    $h_C(c) = c$ pro všechny $c \in C$, které jsou binárními konstantami,

$h_C(c) = c_X$ pro všechny $c \in C$, které jsou identifikátory operace nebo skupiny,

$h_C(c) =$ "b_k" pro všechny $c \in C$, které jsou bitovým polem o délce k, kde $k > 0$.

- Přidáme pravidla $X \rightarrow h_C(C_1\ C_2 \ldots C_w) \mid h_A(A_1\ A_2 \ldots A_v)$ do množiny P.

6.1.2 Překlad pomocí PPG

První gramatiku používáme jako vstupní a budeme provádět LR syntaktickou analýzu vstupního řetězce. Druhou gramatiku budeme využívat jako výstupní, upravenou tak, aby prováděla sémantické akce spojené s naplněním a zpracováním atributů a generováním výstupních řetězců.

Překlad budeme provádět zdola nahoru, a je tedy nutné, aby obě gramatiky byly LR. Pokud by to tak nebylo, existují transformace, které tuto gramatiku převedou na ekvivalentní LR gramatiku. Důvodem je to, že jimi popisujeme konečný jazyk.

Další nutnou podmínkou je, že při nejpravější derivaci ve vstupní gramatice se generuje výstupní větná forma, ve které se každý nonterminál může vyskytovat maximálně jednou. Jelikož jsou gramatiky vytvořeny z jazyka ISAC, máme tuto vlastnost zaručenu.

Poslední podmínkou je vzájemná bijekce mezi pravou a levou stranou pravidla. Tuto vlastnost máme také zaručenou tím, že gramatiky vycházejí z jazyka ISAC.

Výhodou takového přístupu je, že vždy můžeme provést transformaci gramatik tak, aby atributy, v našem případě bitová pole, byly šířeny v rámci páru pravidel, nikoli celým derivovaným stromem nebo jeho částmi.

Aktuální hodnota překládaného vstupního řetězce je uložena ve zvláštní proměnné reprezentující některý z nonterminálů. Výsledný řetězec se vždy z pravé strany „navěsí" na proměnnou reprezentující levý nonterminál. Pokud se na pravé straně vyskytuje terminální symbol, vezme se jeho hodnota. Pokud je na pravé straně nonterminál, vezme se hodnota, která je „zavěšená" na příslušné proměnné a „navěsí" se na proměnnou reprezentující levý nonterminál. Přitom se hodnota v původní proměnné smaže. V případě, že na pravé straně je řetězec složený z terminálních a nonterminálních symbolů, jejich hodnoty se konkatenují.

6.1.3 Zhodnocení

Je možné, že jedna nebo obě gramatiky nebudou typu LR. Stát se to může například tehdy, když jednoznačná identifikace operace v podobě binární konstanty bude následovat až po bitovém poli. Pokud taková situace nastane u dvou instrukcí, není možno v daný okamžik říci, jak dál derivovat. V tomto případě by také někdy nemuselo fungovalo předávání atributu v rámci jednoho pravidla.

Jazyk ISAC disponuje ještě větvícími konstrukcemi, které umožňují v rámci jedné operace více sekcí `CODING` nebo `ASSEMBLER`. Větvicí sekce by tedy zavedly do gramatiky kontext, což by znemožnilo použít překladové párové gramatiky. Je tedy nutné opět vytvořit transformaci. Ta by

z podmínky větvení a jednotlivých sekcí vytvořila takové nové páry pravidel, které by reflektovaly všechny možné hodnoty, kterých může výraz v podmínce nabýt.

V obou popsaných případech by výrazně vzrostla výška derivačního stromu a tím i délka jeho sestavení. Protože nám u zpětného assembleru jde především o rychlost, je toto řešení nevyhovující.

6.2 Výběr nového vnitřního modelu

Pokud bychom chtěli rozšířit předchozí schéma překladu, zejména o možnost větvení v operacích, znamenalo by to velký nárůst pravidel, tím pádem při přijímání vstupního řetězce i velikost derivačního stromu. Díky tomuto by docházelo ke zpožděním, které nejsou přípustná.

Proto bylo potřeba najít jiné řešení. Jelikož jsou assemblery regulárními jazyky, bylo nasnadě použít aparát konečných automatů místo zásobníkových, užitých při použití párových překladových gramatik. Po prozkoumání problematiky regulárních jazyků a jejich modelů se jeví jako použitelné použít párových konečných automatů. To, že jimi můžeme popsat všechny programy v jazyku ISAC, je jasné z následujících teorémů a důkazů.

6.2.1 Teorém 1

Pro každou deklaraci operace ve tvaru:

"OPERATION *X* **{**

ASSEMBLER *Asm***;**

CODING *Cod***;**

};"

existuje PKA Γ takový, který překládá zdrojový kód assembleru této operace do příslušného strojového kódu a naopak.

Důkaz:

Zkonstruujeme PKA Γ, $\Gamma = (M_1, M_2, h)$, kde:

- $M_1 = (\{s_1, f_1\}, alph\{Asm\}, \{s_1Asm \rightarrow f_1\}, s_1, \{f_1\})$,
- $M_2 = (\{s_2, f_2\}, alph\{Cod\}, \{s_2Cod \rightarrow f_2\}, s_2, \{f_2\})$,
- $h = \{(s_1Asm \rightarrow f_1, s_2Cod \rightarrow f_2)\}$.

Takto vytvořený PKA přeloží zdrojový kód assembleru této operace do příslušného strojového kódu a naopak.

6.2.2 Teorém 2

Nechť Id_i je jméno deklarované operace nebo skupiny a Γ_i je PKA, který přeloží zdrojový kód v assembleru této operace do příslušného strojového kódu a naopak pro všechny $i = 1,..,n$. Pak, pro každou deklaraci operace tvaru:

"OPERATION X **{**

INSTANCES Id_1 **ALIAS** A_1,

Id_2 **ALIAS** A_2,

…

Id_n **ALIAS** A_n

ASSEMBLER $Asm_0\ A_{i1}\ Asm_1\ A_{i2}\ \dots\ A_{in}\ Asm_n$ **;**

CODING $Cod_0\ A_{j1}\ Cod_1\ A_{j2}\ \dots\ A_{jn}\ Cod_n$ **;**

};"

existuje PKA Γ takový, který překládá zdrojový kód assembleru této operace do příslušného strojového kódu a naopak.

Důkaz:

Mějme $\Gamma_i = (M_i^1, M_i^2, h_i)$, kde:

- $M_i^1 = (Q_i^1, \Sigma_i^1, R_i^1, s_i^1, \{f_i^1\})$,
- $M_i^2 = (Q_i^2, \Sigma_i^2, R_i^2, s_i^2, \{f_i^2\})$,

pro všechny $i = 1,\dots,n$.

Bez ztráty obecnosti můžeme předpokládat, že pro jakékoli $i = 1,\dots,n$; $j = 1,\dots,n$; $i \neq j$; $k = 1,2$; platí $Q_i^k \cap Q_j^k = \{\}$ a $Q_i^k \cap \{s_k, f_k\} = \{\}$.

Pak zkonstruujeme PKA Γ, $\Gamma = (M_1, M_2, h)$, kde:

- $M_1 = (Q_1, \Sigma_1, R_1, s_1, \{f_1\})$, kde:
  - $Q_1 = \{s_1, f_1\} \cup \left(\bigcup_{i=1}^{n} Q_i^1\right)$,
  - $\Sigma_1 = \bigcup_{i=0}^{n} alph(Asm_i) \cup \left(\bigcup_{i=1}^{n} \Sigma_i^1\right)$,
  - $R_1 = \bigcup_{i=1}^{n} R_i^1 \cup \{f_i^1 Asm_i \rightarrow s_{i+1}^1 : 1 \leq i \leq n-1\} \cup \{s_1 Asm_0 \rightarrow s_1^1, f_n^1 Asm_n \rightarrow f_1\}$,
- $M_2 = (Q_2, \Sigma_2, R_2, s_2, \{f_2\})$, kde:
  - $Q_2 = \{s_2, f_2\} \cup \left(\bigcup_{i=1}^{n} Q_i^2\right)$,
  - $\Sigma_2 = \bigcup_{i=}^{n} alph\ (Cod_i) \cup \left(\bigcup_{i=1}^{n} \Sigma_i^2\right)$,
  - $R_2 = \bigcup_{i=1}^{n} R_i^2 \cup \{f_i^2 Cod_i \rightarrow s_{i+1}^2 : 1 \leq i \leq n-1\} \cup \{s_2 Cod_0 \rightarrow s_1^2, f_n^2 Cod_n \rightarrow f_2\}$,
- $h = \bigcup_{i=1}^{n} h_i \cup \{(f_i^1 Asm_i \rightarrow s_{i+1}^1, f_i^2 Cod_i \rightarrow s_{i+1}^2) : 1 \leq i \leq n-1\} \cup \{(s_1 Asm_0 \rightarrow s_1^1, s_2 Cod_0 \rightarrow s_1^2), (f_n^1 Asm_0 \rightarrow f_1, f_n^2 Cod_n \rightarrow f_3)\}$.

Takto vytvořený PAK přeloží zdrojový kód assembleru této operace do příslušného strojového kódu a naopak.

6.2.3 Teorém 3

Nechť Id_i je jméno deklarované operace nebo skupiny a Γ_i je PKA, který přeloží zdrojový kód v assembleru této operace do příslušného strojového kódu a naopak. Pak, pro každou deklaraci skupiny tvaru:

GROUP G **=** $Id_1, Id_2, \ldots, Id_n$**;**

existuje PKA Γ takový, který překládá zdrojový kód assembleru této skupiny do příslušného strojového kódu a naopak.

Důkaz:

Mějme $\Gamma_i = (M_i^1, M_i^2, h_i)$, kde:

- $M_i^1 = (Q_i^1, \Sigma_i^1, R_i^1, s_i^1, \{f_i^1\})$,
- $M_i^2 = (Q_i^2, \Sigma_i^2, R_i^2, s_i^2, \{f_i^2\})$,

pro všechny $i = 1,\ldots,n$.

Bez ztráty obecnosti můžeme předpokládat, že pro jakékoli $i = 1,\ldots,n$; $j = 1,\ldots,n$; $i \neq j$; $k = 1,2$; platí $Q_i^k \cap Q_j^k = \{\}$ a $Q_i^k \cap \{s_k, f_k\} = \{\}$.

Pak zkonstruujeme PKA Γ, $\Gamma = (M_1, M_2, h)$, kde:

- $M_1 = (Q_1, \Sigma_1, R_1, s_1, \{f_1\})$, kde:
  - $Q_1 = \{s_1, f_1\} \cup \left(\bigcup_{i=1}^n Q_i^1\right)$,
  - $\Sigma_1 = \bigcup_{i=1}^n \Sigma_i^1$,
  - $R_1 = \bigcup_{i=1}^n R_i^1 \cup \left(\bigcup_{i=1}^n \{s_1 \to s_i^1, f_i^1 \to f_1\}\right)$,
- $M_2 = (Q_2, \Sigma_2, R_2, s_2, \{f_2\})$, kde:
  - $Q_2 = \{s_2, f_2\} \cup \left(\bigcup_{i=1}^n Q_i^2\right)$,
  - $\Sigma_2 = \bigcup_{i=1}^n \Sigma_i^2$,
  - $R_2 = \bigcup_{i=1}^n R_i^2 \cup \left(\bigcup_{i=1}^n \{s_2 \to s_i^2, f_i^2 \to f_2\}\right)$,
- $h = \bigcup_{i=1}^n h_i \cup \left(\bigcup_{i=1}^n \{(s_1 \to s_i^1, s_2 \to s_i^2), (f_i^1 \to f_1, f_i^2 \to f_2)\}\right)$.

Takto vytvořený PKA přeloží zdrojový kód assembleru této skupiny do příslušného strojového kódu a naopak.

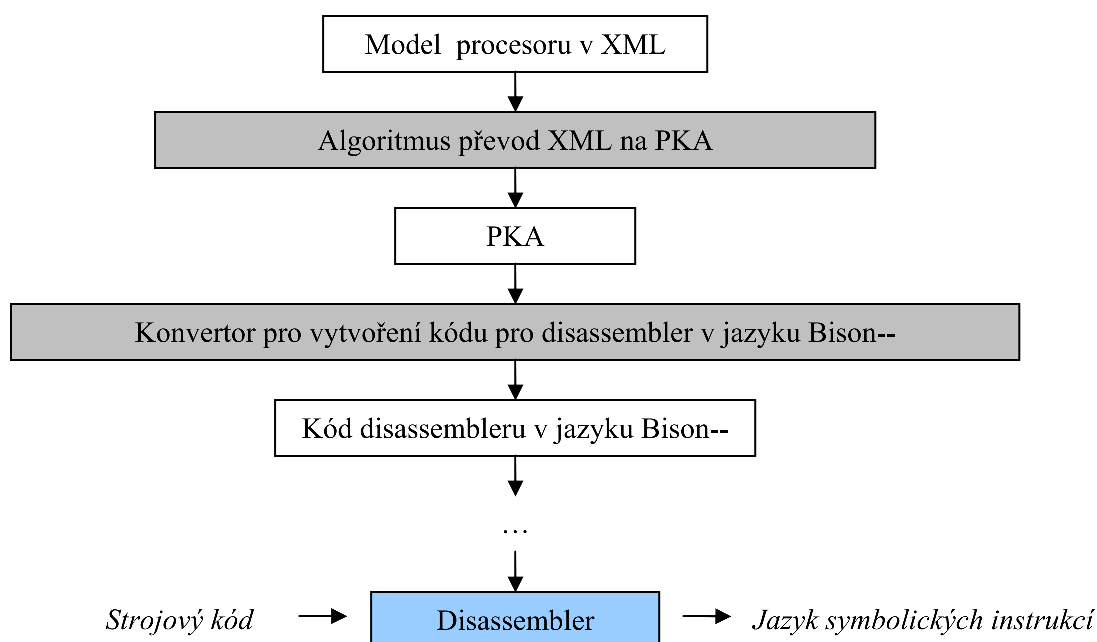
Předchozí teorémy vychází z [9] a [10].

6.2.4 Závěr

Pro každý popsaný procesor v jazyku ISAC existuje párový konečný automat Γ taký, který přeloží zdrojový program v jazyku assembler do příslušného strojového kódu a naopak.

6.3 Nová metodologie vyváření disassembleru

Zde si popíšeme implementační detaily a algoritmy, které jsou použity při překladu popisu procesoru v XML na disassembler pomocí párových konečných automatů. Všechny implementační postupy počítají s tím, že je XML korektní, tedy neobsahuje syntaktické ani sémantické chyby (např. pojmenování atributů musí korespondovat mezi sekcemi `assembler` a `coding`). V opačném případě by docházelo k nedeterministickému chováni. Návrh i implementace algoritmů počítá s pozdější rozšiřitelností. Ta je zajištěna velkou modularitou systému. Schéma postupu je naznačeno na následujícím obrázku.



Obr. 12:Obecné schéma převodu modelu procesoru v XML na disassembler pomocí PKA

6.3.1 Převod XML na PKA

Na XML je kladen požadavek, aby každá operace, než je použita v sekci `assembler`, `coding` nebo `group`, byla deklarována. To můžeme vynutit buď tím, že v opačném případě překladač jazyka ISAC skončí s chybou, a tedy uživatel bude nucen nejprve nadeklarovat jednodušší operace, z nichž se skládají složitější. Nebo necháme uživateli volnou ruku a přeuspořádání provede překladač v rámci svého vnitřního modelu při překladu. V projektu je použita první varianta, jelikož je snadnější na implementaci a překlad je rychlejší, avšak do budoucna se plánuje užití spíše varianty druhé, při níž bude překlad sice složitější, ale pro uživatele bude modelování procesorů pohodlnější.

Při převodu nevzniká pouze jeden párový konečný automat, nýbrž pro každou operaci nebo skupinu právě jeden. Po zpracování celého vstupního XML se generuje pouze poslední. Tím, že je

uživatel donucen nejprve deklarovat, a pak používat, máme zaručeno, že poslední operace je kořenem pro dekódování.

6.3.2 Poloformální popis algoritmu pro převod XML na PKA

Převod pak probíhá podle následujícího předpisu.

Mějme množinu párových konečných automatů Ψ. Na začátku algoritmu je tato množina prázdná.

Pro každou sekci ve tvaru:

“**GROUP** X **=** O_1**,** O_2**,** …**,** O_n**;** ”, kde $n > 0$,

inicializuj nový párový automat Γ takto:

$M_1 = (\{X_s, X_f\}, \emptyset, \emptyset, X_s, \{X_f\})$,

$M_2 = (\{X_s, X_f\}, \emptyset, \emptyset, X_s, \{X_f\})$,

$h = \{\}$

a pro všechny i, kde $1 \leq i \leq n$ proveď:

- Vytvoříme nový párový automat O_i^c, který je kopii párového automatu v množině Ψ reprezentující operaci nebo skupinu O_i, a vhodně u něj změníme názvy stavů v $M_1(M_2)$ tak, aby se nevyskytovaly v žádné množině stavů $M_1(M_2)$ párových konečných automatů obsažených v množině Ψ a vložíme ho do Ψ.
- Přidáme pravidlo $X_s\varepsilon \rightarrow O_i^c_m_1_s$ do množiny R_1 a pravidlo $X_s\varepsilon \rightarrow O_i^c_m_2_s$ do množiny R_2, kde $O_i^c_m_1_s$ je stravovací stav automatu M_1 v O_i^c a $O_i^c_m_2_s$ je startovací stav M_2 v O_i^c.
- Přidáme symboly z množiny Σ_k automatu M_k v O_i^c do množiny Σ_k, kde $k \in \{1,2\}$.
- Přidáme stavy z množiny Q_k automatu M_k v O_i^c do množiny Q_k, kde $k \in \{1,2\}$.
- Přidáme pravidla z množiny R_k automatu M_k v O_i^c do množiny R_k, kde $k \in \{1,2\}$.
- Přidáme prvky ze zobrazení h v O_i^c do zobrazení h.
- Přidáme pravidlo $O_i^c_m_1_f\varepsilon \rightarrow X_f$ do množiny R_1 a pravidlo $O_i^c_m_2_f\varepsilon \rightarrow X_f$ do množiny R_2, kde $O_i^c_m_1_f$ je stav z množiny F_1 v O_i^c a $O_i^c_m_2_f$ je stav z množiny F_2 v O_i^c.
- Přidáme prvky $(X_s\varepsilon \rightarrow O_i^c_m_1_s, X_s\varepsilon \rightarrow O_i^c_m_2_s)$ a $(O_i^c_m_1_f\varepsilon \rightarrow X_f, O_i^c_m_2_f\varepsilon \rightarrow X_f)$ do h.

Takto vytvořený automat přidáme do množiny Ψ.

Pro každou sekci ve tvaru:

“**OPERATION** *X* **{** [**INSTANCE** *I*;]
ASSEMBLER *A*;
CODING *C*;
… }”

inicializuj nový párový automat Γ takto:

$M_1 = (\{X_s, X_f\}, \varnothing, \varnothing, X_s , \{ X_f\})$,

$M_2 = (\{X_s, X_f\}, \varnothing,\varnothing, X_s, \{ X_f\})$,

$h = \{\}$,

a dále platí pro I:

- Pokud je I ve tvaru "*ID*", tak vytvoříme nový párový automat Al_I^c, který je kopii párového automatu v množině Ψ reprezentující operaci nebo skupinu I, a vhodně u něj změníme názvy stavů v $M_1(M_2)$ tak, aby se nevyskytovaly v žádné množině stavů $M_1(M_2)$ párových konečných automatů obsažených v množině Ψ a vložíme ho do Ψ.
- Pokud je I ve tvaru "*ID* **ALIAS {** Al_1**,** Al_2**, …,** Al_m **}**", kde $m > 0$, tak pro všechny i, kde $1 \leq i \leq m$ vytvoříme nový párový automat Al_i^c, který je kopii párového automatu v množině Ψ reprezentující operaci nebo skupinu I, a vhodně u něj změníme názvy stavů v $M_1(M_2)$tak, aby se nevyskytovaly v žádné množině stavů $M_1(M_2)$ párových konečných automatů obsažených v množině Ψ a vložíme ho do Ψ.

Nechť je A ve tvaru "**{** $A_1, A_2, \ldots, A_v$ **}**", kde $v > 0$ a C ve tvaru "**{** $C_1, C_2, \ldots, C_w$ **}**", kde $w > 0$, pak:

i. Všechny atributy v sekci A se nahradí terminálním symbolem '#U' nebo '#S' pro bezznaménkový nebo znaménkový atribut a atributy v C sekci se nahradí terminálním symbolem '0xb[*k*]' nebo '0xbs[*k*]', kde k označuje délku bitového pole, který atribut zabírá, pro bezznaménkový nebo znaménkový atribut.

ii.

- Pokud jsou $A_{1, \ldots,} A_v$ pouze terminálními symboly nebo atributy, vytvoříme pravidlo $X_sa \rightarrow X_f$, kde a jsou zkonkatenované hodnoty $A_{1, \ldots,} A_v$, a vložíme ho do R_1, dále $alph(a)$ vlož do Σ_1.
- Zároveň musí platit, že i $C_1, \ldots, C_w$ se skládá pouze z terminálních symbolů nebo atributů. Pak se vytvoříme pravidlo $X_sb \rightarrow X_f$, kde b jsou zkonkatenované hodnoty $C_1, \ldots, C_w$, a vložíme ho do R_2, dále $alph(b)$ vložíme do Σ_2.
- Přidáme prvek ($X_sa \rightarrow X_f, X_sb \rightarrow X_f$) do zobrazení h.

iii.

a) Pro všechna s, kde s jsou indexy prvků v sekci A, které jsou instancí dělej:

- Pokud je A_s první instancí v sekci A, přidej pravidlo $X_sa \rightarrow Al_s^c_m_1_s$ do R_1, kde $Al_s^c_m_1_s$ je startovací stav automatu M_1 v Al_s^c a a je tvořeno konkatenací terminálních symbolů nebo atributů od prvku A_1 do prvku předcházejícímu A_s.
- Jinak přidej pravidlo $Al_{s-1}^c_m_1_fa \rightarrow Al_s^c_m_1_s$ do R_I, kde $Al_{s-1}^c_m_1_f$ je stav z množiny koncových stavů F_1 automatu M_1 v Al_{s-1}^c, $Al_s^c_m_1_s$ je startovací stav automatu M_1 v Al_s^c a a je tvořeno konkatenací terminálních symbolů nebo atributů mezi prvky A_{s-1} a A_s.
- Přidáme symboly z množiny Σ_1 a $aplh(a)$ automatu M_1 v Al_s^c do množiny Σ_1.
- Přidáme stavy z množiny Q_1 automatu M_1 v Al_s^c do množiny Q_1.

- Přidáme pravidla z množiny R_1 automatu M_1 v Al_s^c do množiny R_1.
- Pokud je A_s poslední instancí v sekci A, přidej pravidlo $Al_s^c_m_1_fa \rightarrow X_f$ do R_1, kde $Al_s^c_m_1_f$ je stav z množiny koncových stavů F_1 automatu M_1 v Al_s^c a a je tvořeno konkatenací terminálních symbolů nebo atributů od prvku následujícímu po A_s do prvku A_v.

b) Pro všechny d, kde d jsou indexy prvků v sekci C, které jsou instancí dělej:

- Pokud je C_d první instancí v sekci C, přidej pravidlo $X_sb \rightarrow Al_d^c_m_2_s$ do R_2, kde $Al_d^c_m_2_s$ je startovací stav automatu M_2 v Al_d^c a b je tvořeno konkatenací terminálních symbolů nebo atributů od prvku C_1 do prvku předcházejícímu C_d.
- Jinak přidej pravidlo $Al_{d-1}^c_m_2_fb \rightarrow Al_d^c_m_2_s$ do R_2, kde $Al_{d-1}^c_m_2_f$ je stav z množiny koncových stavů F_2 automatu M_2 v Al_{d-1}^c, $Al_d^c_m_2_s$ je startovací stav automatu M_2 v Al_d^c a b je tvořeno konkatenací terminálních symbolů nebo atributů mezi prvky C_{d-1} a C_d.
- Přidáme symboly z množiny Σ_2 a $alph(b)$ automatu M_2 v Al_d^c do množiny Σ_2.
- Přidáme stavy z množiny Q_2 automatu M_2 v Al_d^c do množiny Q_2.
- Přidáme pravidla z množiny R_2 automatu M_2 v Al_d^c do množiny R_2.
- Pokud je C_d poslední instancí v sekci C, přidej pravidlo $Al_d^c_m_2_fb \rightarrow X_f$ do R_2, kde $Al_d^c_m_2_f$ je stav z množiny koncových stavů F_2 automatu M_2 v Al_d^c a b je tvořeno konkatenací terminálních symbolů nebo atributů od prvku následujícímu po C_d do prvku C_w.

c) Přidáme prvky zobrazení h ze všech Al_i^c kde $1 \leq i \leq m$ do h.

d) Prvky $(Y_1t_1 \rightarrow Z_1, Y_2t_2 \rightarrow Z_2)$, kde Y_1 a Y_2 jsou koncové stavy stejných instance nebo je to startující stav automatu M_1 a M_2, $t_1 \in \Sigma_1^*$ a $t_2 \in \Sigma_2^*$ jsou konstantní texty, Z_1 a Z_2 jsou startující stavy stejných instancí nebo jsou to stavy z množiny koncových stavů F_1 a F_2 automatu M_1 a M_2 a $Y_1 \neq Z_1$, $Y_2 \neq Z_2$, přidej do h.

6.3.3 Ukázka algoritmu

V ukázce budeme budovat párový konečný automat od základních operací ke složitějším. Základem v našem případě budou registry. Barevně jsou odlišeny hrany, které jsou ve dvojici v zobrazení h.

Operace ax:

```
OPERATION ax {
    ASSEMBLER { "AX" }
    CODING { 0b00 }
}
```

Vstupní data:

$\Psi = \emptyset$,

Γ : M_1 = ({ax_s, ax_f }, ∅, ∅, ax_s , { ax_f }),

M_2 = ({ax_s, ax_f }, ∅, ∅, ax_s, { ax_f }),

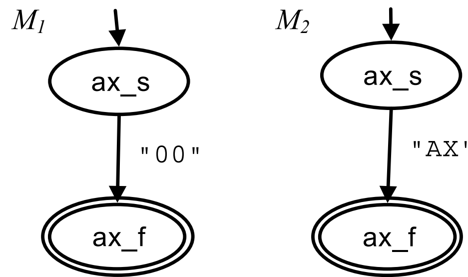
h = {}.

<u>*Vzniklý párový konečný automat:*</u>

Γ : M_1 = ({ax_s, ax_f }, {“0”}, {ax_s “00” → ax_f}, ax_s , { ax_f }),

M_2 = ({ax_s, ax_f }, {“AX”}, { ax_s “AX” → ax_f }, ax_s, { ax_f }),

h = {(ax_s “00” → ax_f , ax_s “AX” → ax_f)}.



Obr. 13: Automaty pro operaci *ax*

<u>*Výstupní data:*</u>

Ψ = {*ax*}

Operace bx:

```
OPERATION bx {
      ASSEMBLER { “BX” }
      CODING { 0b11 }
}
```

Postup vytváření by byl stejný, zaměnily by se jen všechny výskyty *ax* na *bx* a “00” na “11”.

<u>*Výstupní data:*</u>

Ψ = {*ax*, *bx*}

Skupina register:

```
GROUP register = ax, bx;
```

<u>*Vstupní data:*</u>

Ψ = {*ax*, *bx*},

Γ : M_1 = ({register_s, register_f }, ∅, ∅, register_s , { register_f }),

M_2 = ({register_s, register_f }, ∅, ∅, register_s, { register_f }),

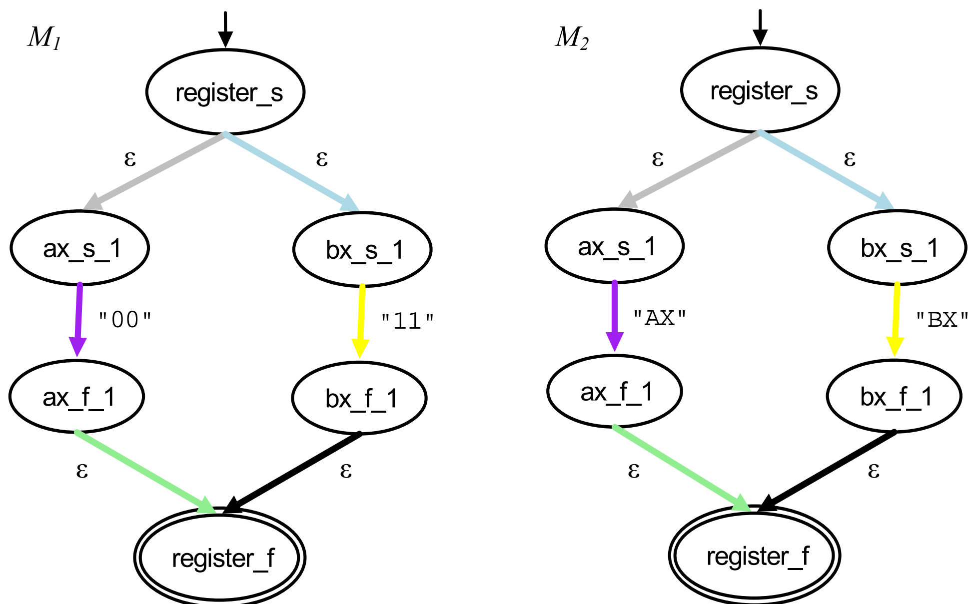
h = {}.

<u>*Vzniklý párový konečný automat:*</u>

Γ : M_1 = ({register_s, register_f }, {“0” , “1”}, {register_sε → ax_s_1, register_sε → bx_s_1, ax_s_1 “00” → ax_f_1, bx_s_1 “11” → bx_f_1, ax_f_1ε → register_f, bx_f_1ε → register_f }, register_s , { register_f }),

M_2 = ({register_s, register_f }, {“A”, “B”, “X”}, {register_sε → ax_s_1, register_sε → bx_s_1, ax_s_1 “AX” → ax_f_1, bx_s_1 “BX” → bx_f_1, ax_f_1ε → register_f, bx_f_1ε → register_f}, register_s, { register_f }),

h = {(register_sε → ax_s_1, register_sε → ax_s_1), (register_sε → bx_s_1, register_sε → bx_s_1), (ax_s_1 “00” → ax_f_1, ax_s_1 “AX” → ax_f_1), (bx_s_1 “11” → bx_f_1, bx_s_1 “BX” → bx_f_1), (ax_f_1ε → register_f, ax_f_1ε → register_f), (bx_f_1ε → register_f, bx_f_1ε → register_f)}.



Obr. 14: Automaty pro operaci *register*

Výstupní data:

Ψ = {*ax*, *bx*, *register*}

Takto by se pokračovalo pro každou další operaci nebo skupinu.

6.3.4 Překlad pomocí PKA

Při překladu používáme první konečný automat M_1 jako přijímací, tedy na vstupu přijímá strojový kód. Druhý automat M_2 nepřijímá žádný vstup a je použit pro generovaní příslušného programu v jazyku symbolických instrukcí. Automaty komunikují mezi sebou přes společné sdílené proměnné.

Automat M_1 funguje standardně, tedy provede přechod ze jednoho stavu do druhého tehdy, je-li na vstupu sekvence terminálních symbolů, kterou vyžaduje dané pravidlo.

Oproti tomu automat M_2 funguje inverzně. Podle pravidel se mezi stavy přechází bez čtení vstupu a sekvence terminálních symbolů příslušných pravidel se generuje na výstup. V obou případech, tedy jak v automatu M_1 , tak v automatu M_2, řadíme mezi terminální symboly i atributy.

Je nutné provést implementační rozšíření oproti definici párového konečného automatu o sémantické akce. Sémantickou akcí myslíme úsek nedělitelného kódu, který je proveden po přechodu z jednoho stavu do druhého. Sémantická akce je tedy spojena s určitým pravidlem v automatu.

Jelikož jsou automaty generovány jako nedeterministické, je nutné provést determinizaci. Přitom tento požadavek je kladen pouze na přijímací automat. Determinizaci zajišťuje modul v překladači jazyka Bison-- (viz. kap. 7.1). U druhého automatu požadavek na determinizmus není. Průchod druhým automatem je totiž řízen proměnnými, resp. jejich hodnotami, které nastavil automat první při příjmu vstupního řetězce.

V obou případech je nutné zajistit konverze formátů a typů atributů, které se vyskytují v terminálních sekvencích v pravidlech automatů. Jde o to, že hodnota atributu je ve strojovém kódu zastoupena binárním bitovým polem. První automat tedy musí obsahovat konverzní funkce, které tato pole převedou do desítkové soustavy a následně uloží do proměnné celočíselného typu, která tento atribut reprezentuje. Tyto převody jsou přidány do sémantických akcí příslušejícímu pravidlu. Spolu s nastavováním `switch` proměnné u skupin se jedná o jediné sémantické akce, které se provádí při příjmu vstupu.

Naopak při generování je nutné nastavené proměnné prvním automatem převést na řetězce, aby se mohl generovat textový výstup, tedy ekvivalentní program v jazyce symbolických instrukcí. Tyto konverze jsou opět přidány pro příslušných sémantických akcí.

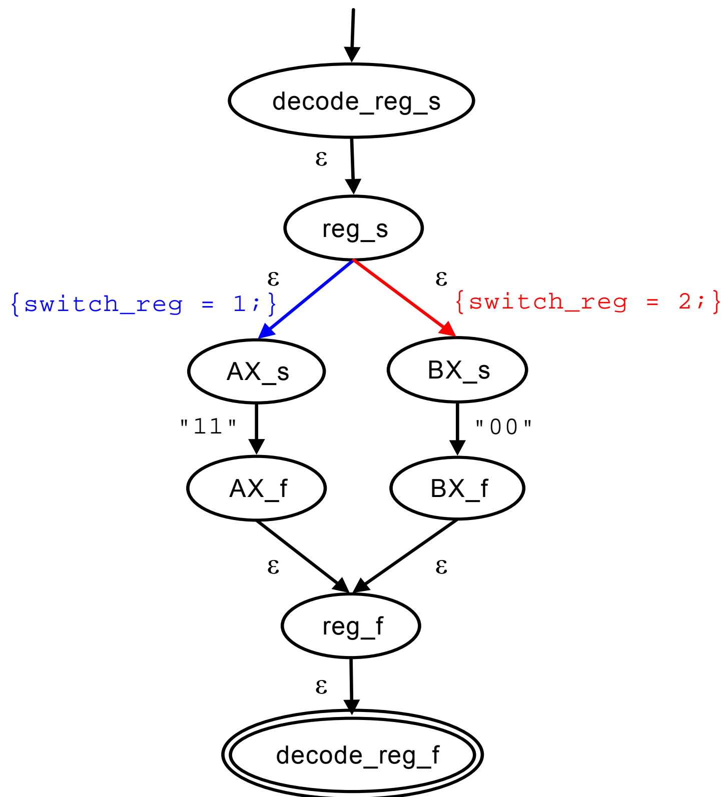
Přitom platí, že uživatelské sémantické akce se provádějí pouze v druhém automatu, tedy až při generování.

Kromě atributů, které definuje uživatel, je přidán do všech úvodních hran, které vznikají při zpracováni skupin, jeden speciální atribut. Jde o rezervovaný atribut `switch`. Při přijímání vstupu první deterministický automat nastaví všechny proměnné tohoto typu na hodnoty, které reprezentují větve, kterými se překlad ubíral. Tedy pokud máme skupinu, která obsahuje 3 operace/skupiny, vznikne stav, ze kterého vedou 3 hrany. Pak v proměnné `switch` hodnota od 1 do 3 reprezentuje, které pravidlo jsme při přijímání řetězce použili. Toto nastavení pak reflektuje druhý automat při generování. Pokud totiž generující automat narazí na větvení pomocí skupiny, podívá se, jaká hodnota je uložena v příslušné `switch` proměnné, a dál provádí generování cestou, kterou určí tato proměnná.

Z výše uvedeného je zřejmé, proč není potřeba mít i druhý automat deterministický. Kdykoli se totiž narazí na potenciální nejednoznačnost, je informace o tom, jak dál pokračovat, uložena v proměnné `switch`.

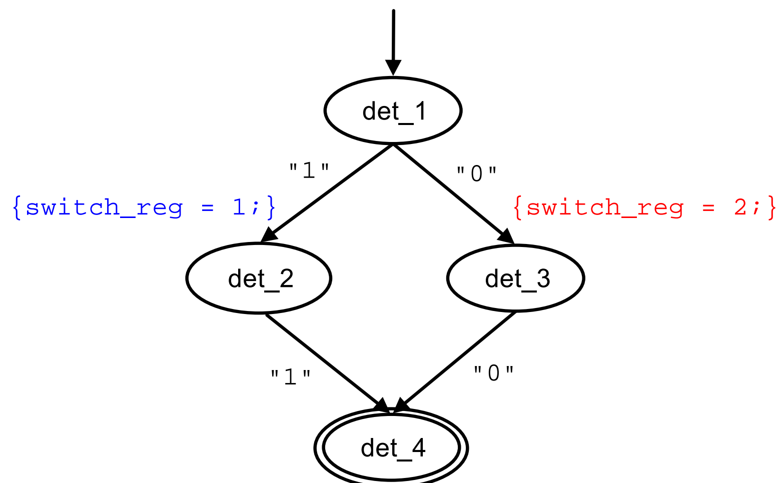
Rozhodování ani generování výstupu není uloženo v žádných sémantických akcích, tyto operace jsou zpracovávány samostatným modulem, který je vytvořen překladačem jazyka Bison--.

Vše je ilustrováno na následujícím příkladu.



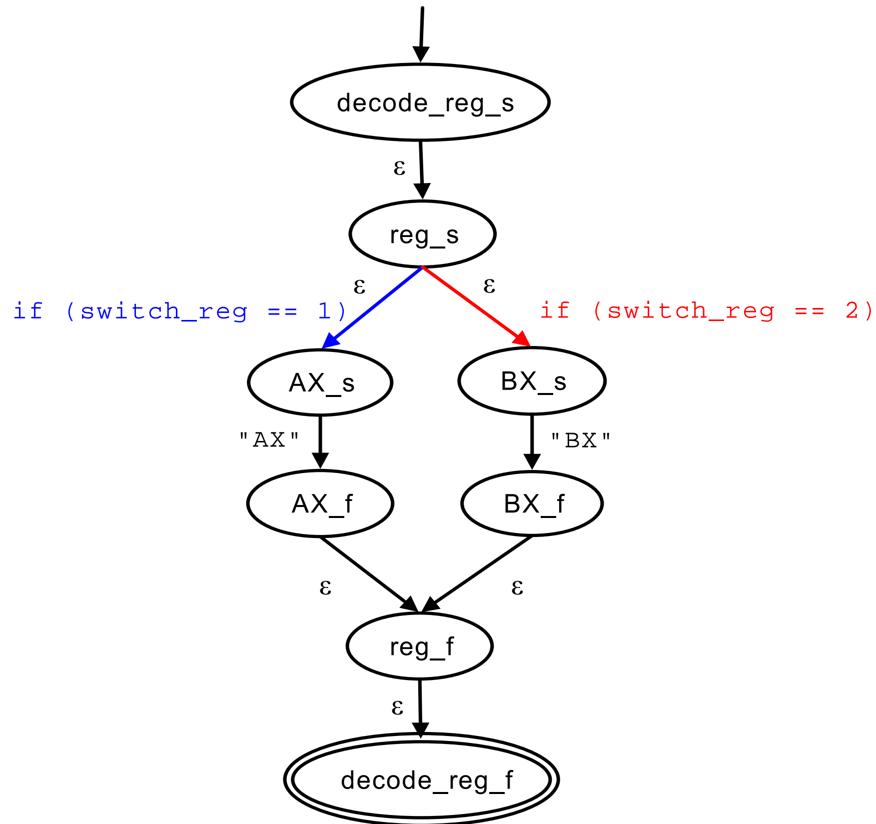
Obr. 15: Přijímací automat M_1

U automatu jsou barevně odlišeny hrany a sémantické akce, které vznikly při zpracování skupiny. Tento automat musí prodělat fázi determinizace. Je však nutné zachovat sémantické akce na správných hranách. Pokud by byla sémantická akce přiřazena k nesprávné hraně, mohlo by dojít v lepším případě k náhodnému chování, v horším pak k pádu aplikace. Výsledný deterministický automat je naznačen na následujícím obrázku.



Obr. 16: Deterministický přijímací automat M_1

Nechť je "11"vstupní věta, pak deterministický vstupní automat M_1 po přijetí znaku "1" nastaví do proměnné `switch_reg` hodnotu 1 a skončí přijímání vstupu po přijetí znaku "1". Nyní přijde na řadu generující automat. Ten již vstup nebude brát žádný. Automat M_2 je naznačen pod textem.



Obr. 17: Generující automat M_2

Generující automat pak generuje terminální symboly a hodnoty atributu, které jsou uložené v proměnných na výstup. Pokud je mezi stavy jen jedna hrana, automat nemá volbu a užije tuto hranu. Pokud lze ze stavu jít více hranami, jde o větvení pomocí skupiny. Automat si pak zjistí hodnotu proměnné `switch` příslušející dané skupině a podle ní pokračuje dále v generování. V našem příkladě se tedy vygeneruje na výstup hodnota "AX".

7 Implementace disassembleru

Pro implementaci využijeme jazyk C a C++. Jazyk C pro výsledný disassembler, protože výsledný kód je rychlý a dá se výrazně optimalizovat. Jazyk C++ použijeme pro generování disassembleru, protože dovoluje objektový přístup, což umožňuje použít mocnější návrhové prostředky. Dalším jazykem je jazyk Bison--, což je jazyk popisující párové nedeterministické konečné automaty. Jeho funkce jsou popsány v následující kapitole. Každý s výše uvedených jazyků má svůj překladač. U jazyků C a C++ vytváří překladač spustitelný kód. U jazyka Bison-- pak reprezentaci párových automatů v jazyku C.

7.1 Jazyk Bison--

Tato podkapitola popisuje jazyk Bison--, který je využit pro generování jak přijímací části disassembleru, tak jeho generující části. Překladač pak transformuje popis do jazyka C.

Jazyk BISON--

Formát:

{Přijímající párový automat}

%%

{Generující konečný automat}

%%

{Deklarace proměnných}

Popis jednotlivých částí:

Přijímající párový automat – obsahuje definici líného konečného automatu, použitého pro příjem vstupního binárního souboru.

Generující konečný automat – obsahuje definici líného konečného automatu, použitého pro generování vstupního souboru v jazyku symbolických instrukcí

Oba automaty jsou zapsány v následující syntaxi:

vstupní_stav –> *terminální_symboly výstupní_stav*

{*sémantická_akce*}

Pro koncový stav je použito této konstrukce:

výstupní_stav –>

Kde - *vstupní_stav* je řetězec.

- *terminální_symboly* je řetězec terminálních symbolů. Pokud chceme povolit mezi dvěma terminálními symboly libovolný počet mezer, použijeme rezervovaný znak ~ (např. `ADD~AX` označuje řetězec `AX BX`, stejně tak jako `AX    BX`). Terminální symboly se neuzavírají do žádných uvozovek. Výjimku tvoří symboly označující atributy. Ty musí být v jednoduchých uvozovkách (např. `00'BU4'` značí `00XXXX`, kde `X` nabývá buď hodnoty `1` nebo `0`).
- *výstupní_stav* je řetězec.
- *sémantická_akce* je posloupnost operací v jazyku C. Kromě akcí, které si definuje uživatel, jsou přidávány specielní akce pro nastaveni `switch` proměnných a pro práce s atributy. Pro identifikaci atributů v terminálních symbolech používáme notaci $<číslo> (např. v terminálním řetězci `00'BU4'01'BU2'11` označuje symbol $1 atribut o délce 4 bity a $2 atribut o délce 2 bity. Z atributu lze buď získat hodnotu, tedy `num = atoi($1);`, nebo lze do něj zapsat `$1 = itoa(num);`.

Deklarace proměnných – obsahuje výčet proměnných použitých pro komunikaci mezi automaty.

Překladač jazyka BISON--

Má několik funkcí. Nejdůležitější částí je fáze determinizace. Vezme automat uvedený v první části a pomocí různých metod a algoritmů vytvoří nový automat, který je ekvivalentní se vstupem a je deterministický. Přitom musí zajistit, že sémantické akce zůstanou na správných hranách. Párovost, která je vyžadována ve vstupním souboru, je rozbita a jediný způsob komunikace, který mezi sebou automaty mají, jsou proměnné uvedené v třetí části vstupního souboru. Po této fázi začne transformace přijímajícího deterministického konečného automatu a generujícího nedeterministického líného konečného automatu do jazyka C.

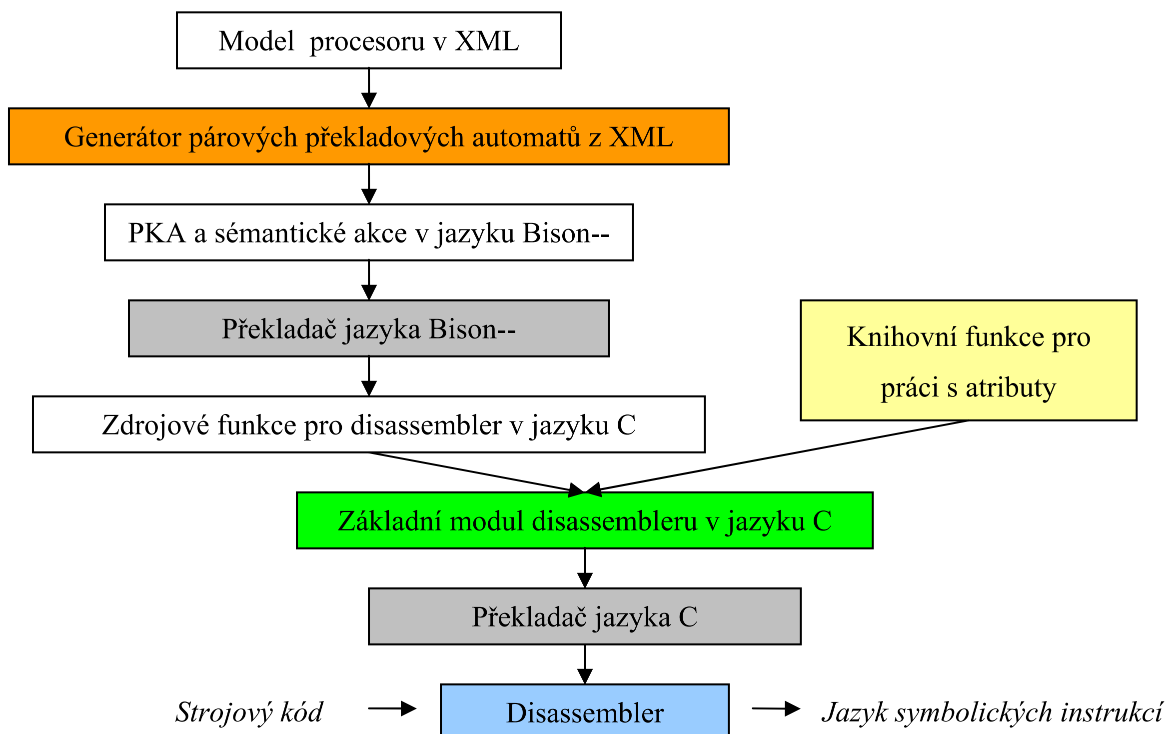
Výsledkem jsou dvě funkce: *parse()*, která převezme jméno souboru, ve kterém je program v binárním kódu, a *generate()*, která vrací textovou reprezentaci přeložené binární instrukce v jazyku symbolických instrukcí.

7.2 Proces vytváření disassembleru

Při vytváření disassembleru vycházíme z namodelovaného procesoru v jazyku XML. V tento okamžik máme zaručeno, že to, co je uvedeno v XML, je správné jak po syntaktické, tak i sémantické stránce. V opačném případě by překladač ISACu XML vůbec nevytvořil. Soubor obsahující model v XML dáme na vstup generátoru párových automatů. Generátor vytvoří program v jazyku Bison-- a uloží ho do souboru. Tento soubor se pak vezme jako vstup pro překladač jazyka Bison--, který zdeterminizuje přijímací automat. Poté provede transformaci tohoto automatu společně s generujícím automatem do jazyka C. Výsledný program se přeloží do spustitelného kódu na cílové platformě. Ten pak na vstupu

přijímá binární soubor a na výstupu je generován ekvivalentní program v jazyku symbolických instrukcí. Během překladu jsou využívány různé knihovní funkce.

Vazby a význam jsou popsány níže.



Obr. 18:Proces vytváření disassembleru

Popis jednotlivých částí:

Model procesoru v XML – přeložený soubor v jazyku ISAC do podoby v XML.

Generátor párových překladových automatů z XML – vezme XML a z něho vytvoří párový konečný automat. Ten je potom uložen do souboru v jazyku Bison--. Také se do výstupního souboru vloží pravidla pro příjem a generování atributů, resp. bitových polí, a akce pro práci se `switch` atributy.

Zdrojové funkce pro disassembler v jazyku C – jak první, tak i druhý automat se transformuje na program v jazyku C, a to konkrétně do dvou funkcí. První z nich, *parse()*, na vstupu očekává strojový kód aplikace. Druhá funkce, *generate()*, podle hodnot vnitřních proměnných nastavených prvním automatem generuje na výstup textovou reprezentaci instrukce v příslušném jazyku symbolických instrukcí.

Knihovní funkce pro práci s atributy – při generování souboru v jazyku Bison-- se používají funkce, které z bitových polí získávají numerické hodnoty a tyto hodnoty pak po provedení sémantických operací převádí zpět na textovou reprezentaci. Další funkce se pak starají o převody čísel mezi různými číselnými soustavami.

Základní modul disassembleru v jazyku C – tento modul spojí vše dohromady, tedy vygenerované funkce reprezentující automaty, funkce pro práci s atributy v nich použitých a základní konstrukci pro všechny disassemblery stejnou. Překladem se vytvoří spustitelný program na dané platformě. Činnost programu končí po přijetí posledního symbolu nebo při chybě.

7.3 Ilustrativní příklad

Ačkoli generátor párového konečného automatu na vstupu očekává XML, v příkladu je pro přehlednost použit zápis v jazyku ISAC.

Operace v ISAC:

```
OPERATION ax {
      ASSEMBLER { "AX" }
      CODING { 0b11 }
}
OPERATION bx {
      ASSEMBLER { "BX" }
      CODING { 0b00 }
}
GROUP register = ax, bx;
OPERATION add {
      INSTANCE register ALIAS { dest, src };
      ASSEMBLER { "ADD" dest "," src ","  attr=#U }
      CODING { 0b0001 src dest attr=0bx[4] }
}
```

Operace vyjádřená v PKA:

Nechť $\Gamma = (M_1, M_2, h)$ je konečný párový automat, kde:

- $M_1 = (Q_1, \Sigma_1, R_1, s_1, F_1)$,
- $M_2 = (Q_2, \Sigma_2, R_2, s_2, F_2)$,
- $h = \{\}$.

A platí:

$Q_1 = Q_2$ = {add_s, register_s_1, ax_s_1_1, bx_s_1_1, ax_f_1_1, bx_f_1_1, register_f_1, register_s_2, ax_s_1_2, bx_s_1_2, ax_f_1_2, bx_f_1_2, register_f_2, add_f },

Σ_1 = { “0”, “1”, ”’BU4’”}, Σ_2 = { “A”, ”D”, “X”, “B”, “’#U’” },

$s_1 = s_2$ = add_s,

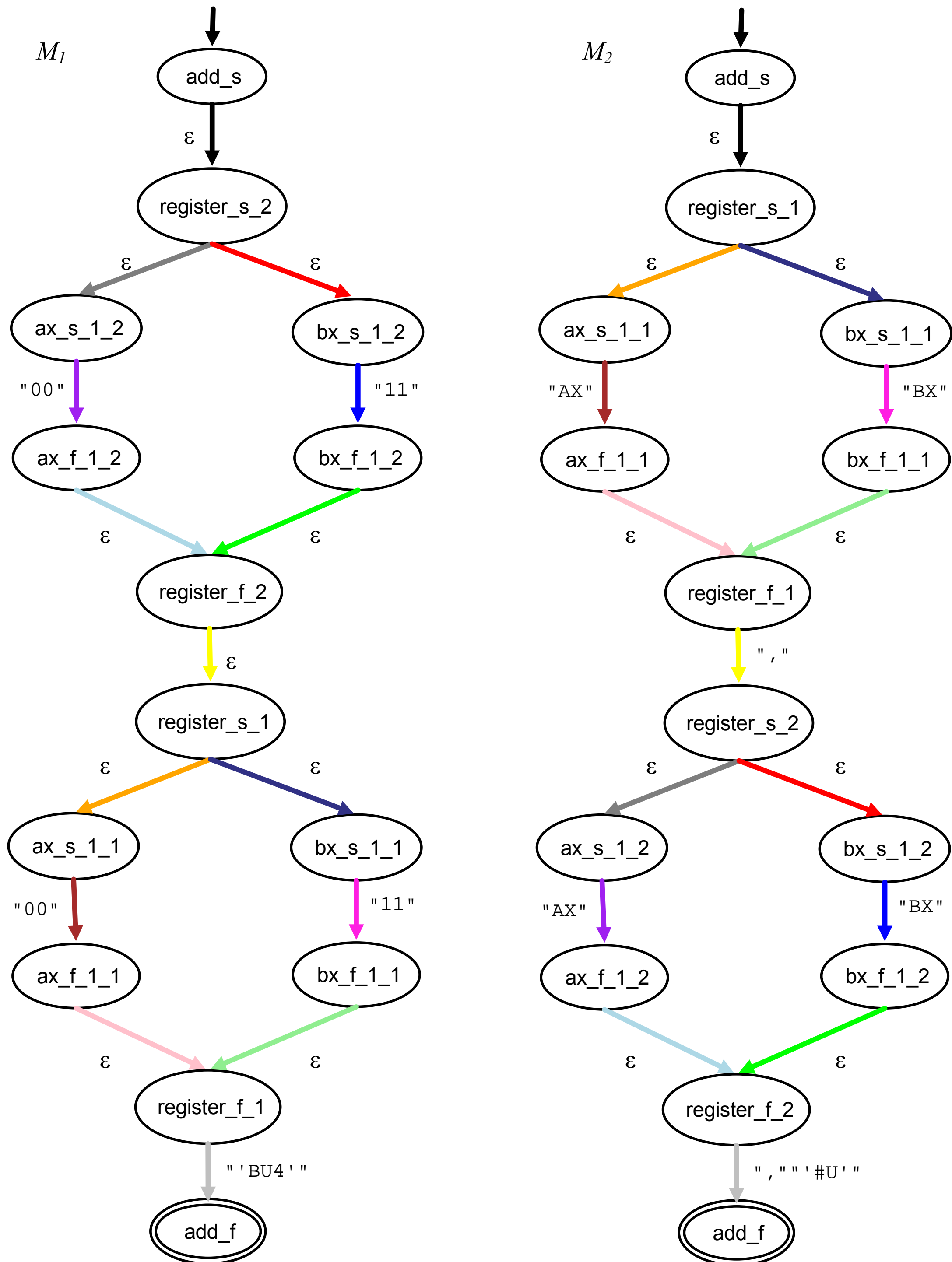
$F_1 = F_2$ = {add_f},

R_1 = {add_s “0001” → register_s_2, register_s_2 ε → ax_s_1_2, ax_s_1_2 “11” → ax_f_1_2, ax_f_1_2 ε → register_f_2, register_s_2 ε → bx_s_1_2, bx_s_1_2 “00” → bx_f_1_2, bx_f_1_2 ε → register_f_2, register_f_2 ε → register_s_1, register_s_1 ε → ax_s_1_1, ax_s_1_1 “11” → ax_f_1_1, ax_f_1_1 ε → register_f_1, register_s_1 ε → bx_s_1_1, bx_s_1_1 “00” → bx_f_1_1, bx_f_1_1 ε → register_f_1, register_f_1 ”’BU4’” → add_f},

R_2 = {add_s “ADD” → register_s_1, register_s_1 ε → ax_s_1_1, ax_s_1_1 “AX” → ax_f_1_1, ax_f_1_1 ε → register_f_1, register_s_1 ε → bx_s_1_1, bx_s_1_1 “BX” → bx_f_1_1, bx_f_1_1 ε → register_f_1, register_f_1 “,” → register_s_2, register_s_2 ε → ax_s_1_2, ax_s_1_2 “AX” → ax_f_1_2, ax_f_1_2 ε → register_f_2, register_s_2 ε → bx_s_1_2, bx_s_1_2 “BX” → bx_f_1_2, bx_f_1_2 ε → register_f_2, register_f_2 ”,””’#U’” → add_f},

h = {(add_s “0001” → register_s_2, add_s “ADD” → register_s_1), (register_s_2 ε → ax_s_1_2, register_s_2 ε → ax_s_1_2), (ax_s_1_2 “11” → ax_f_1_2, ax_s_1_2 “AX” → ax_f_1_2), (ax_f_1_2 ε → register_f_2, ax_f_1_2 ε → register_f_2), (register_s_2 ε → bx_s_1_2, register_s_2 ε → bx_s_1_2), (bx_s_1_2 “00” → bx_f_1_2, bx_s_1_2 “BX” → bx_f_1_2), (bx_f_1_2 ε → register_f_2, bx_f_1_2 ε → register_f_2), (register_f_2 ε → register_s_1, register_f_1 “,” → register_s_2), (register_s_1 ε → ax_s_1_1, register_s_1 ε → ax_s_1_1), (ax_s_1_1 “11” → ax_f_1_1, ax_s_1_1 “AX” → ax_f_1_1), (ax_f_1_1 ε → register_f_1, ax_f_1_1 ε → register_f_1), (register_s_1 ε → bx_s_1_1, register_s_1 ε → bx_s_1_1), (bx_s_1_1 “00” → bx_f_1_1, bx_s_1_1 “BX” → bx_f_1_1), (bx_f_1_1 ε → register_f_1, bx_f_1_1 ε → register_f_1), (register_f_1 ”’BU4’” → add_f, register_f_2 ”,””’#U’” → add_f)}.

Znázornění tohoto párového automatu je na obrázku pod textem. Barevně jsou označeny pravidla, která jsou ve dvojicích v relaci *h*.



Obr. 19: Grafická reprezentace párového konečného automatu

<u>Operace v jazyku Bison--:</u>

Následující část kódu je vygenerovaná, tedy obsahuje už všechny části, včetně sémantických akcí, ať už pro zpracování atributů, nebo pro vzniklé `switch` proměnné.

<u>Transformovaný automat M_1:</u>

```
add_s -> 0001 register_s_2 {};
```

```
register_s_2 ->  ax_s_1_2 { switch_1_2 = ax_s_1_2; };
ax_s_1_2 -> 11 ax_f_1_2 {};
ax_f_1_2 ->  register_f_2 {};
register_s_2 ->  bx_s_1_2 { switch_1_2 = bx_s_1_2; };
bx_s_1_2 -> 00 bx_f_1_2 {};
bx_f_1_2 ->  register_f_2 {};
register_f_2 ->  register_s_1 {};
register_s_1 ->  ax_s_1_1 { switch_1_1 = ax_s_1_1; };
ax_s_1_1 -> 11 ax_f_1_1 {};
ax_f_1_1 ->  register_f_1 {};
register_s_1 ->  bx_s_1_1 { switch_1_1 = bx_s_1_1; };
bx_s_1_1 -> 00 bx_f_1_1 {};
bx_f_1_1 ->  register_f_1 {};
register_f_1 -> '#BU4' add_f { imm = instutilsl_bintounsig($1); };
register_f_1 -> {};
```

Transformovaný automat M_2:

```
add_s -> ADD~ register_s_1 {};
register_s_1 ->  ax_s_1_1 {};
ax_s_1_1 -> AX~ ax_f_1_1 {};
ax_f_1_1 ->  register_f_1 {};
register_s_1 ->  bx_s_1_1 {};
bx_s_1_1 -> BX~ bx_f_1_1 {};
bx_f_1_1 ->  register_f_1 {};
register_f_1 -> ,~ register_s_2 {};
register_s_2 ->  ax_s_1_2 {};
ax_s_1_2 -> AX~ ax_f_1_2 {};
ax_f_1_2 ->  register_f_2 {};
register_s_2 ->  bx_s_1_2 {};
bx_s_1_2 -> BX~ bx_f_1_2 {};
bx_f_1_2 ->  register_f_2 {};
register_f_2 -> ,~'#U' add_f { $1 = instutilsl_unsigtobin(imm, 4); };
register_f_1 -> {};
```

Přijímaný vstup:

```
"000100110011"
```

Generovaný výstup:

```
"ADD AX , BX , 3"
```

8 Problémy k dalšímu řešení a využití disassembleru

8.1 Větvící konstrukce

Jazyk ISAC umožňuje i jiné větvení, než pomocí skupin. Jedná se o větvící konstrukce, které dovolují mít v rámci jedné operace více sekcí `CODING` nebo `ASSEMBLER`. Ve výrazech v podmínkách se můžou vyskytovat atributy, operace, skupiny (operace i skupiny mohou mít sekci `EXPRESSION`, která dovoluje operaci vracet libovolnou celočíselnou hodnotu. Dají se tak např. rozlišovat aritmeticko logické operace apod.) nebo konstanty. Jazyk obsahuje dvě standardní konstrukce, a to `IF` nebo `SWITCH`.

Ilustrativní příklad

Mějme operaci “`add`”.

<u>*Operace v jazyku ISAC:*</u>

```
OPERATION inc_noinc {
    INSTANCE register ALIAS { reg };
    CODING { 0b001 reg attr=0bx[1] };
    IF ( attr == 0 ){
          /*sekce ASSEMBLER  1*/
          ASSEMBLER { "NOINC" reg "," attr=#U };
          …
    }ELSE{
          /*sekce ASSEMBLER 2*/
          ASSEMBLER { "INC" reg "," attr=#U };
          …
    }
}
```

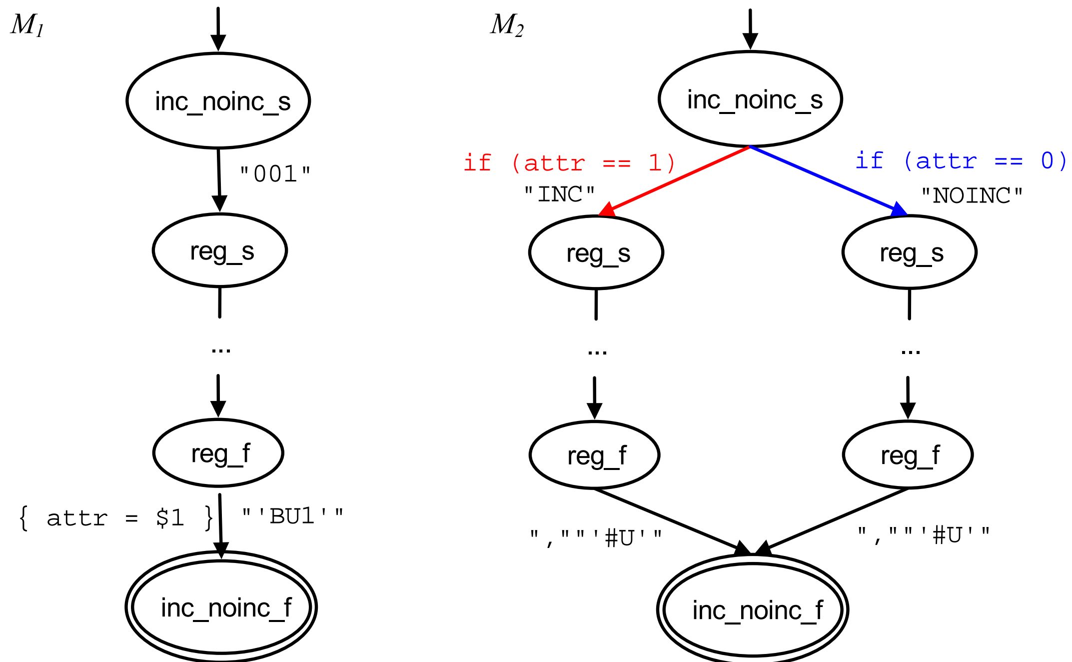
Význam je takový, že podle hodnoty v jednobitovém atributu se rozhoduje, jestli instrukce inkrementuje registr nebo ne.

<u>*Možné řešení:*</u>

Větvící sekce by znamenali poupravení definice párového konečného automatu, protože „párů“ automatů M_1 a M_2 by současně existovalo více, avšak při překladu nebo generování by se uplatil právě jeden pár. Situace by se dala řešit podobně jako u skupin. Při generování automatu M_1 se tedy musejí přidat další sémantické akce, které by při příjmu vstupu naplnily řídící atribut hodnotou. Také

se musejí přidat sémantické akce do automatu M_2, které by dle hodnoty atributu v podmínce rozhodovaly o tom, kterou větví se bude generovat řetězec.

Výše popsané je ilustrováno na následujícím obrázku.



Obr. 20: Znázornění větvení uvnitř operací

8.2 Sekce ACTIVATION a CODINGROOT

Další konstrukce, kterými jazyk ISAC disponuje, jsou sekce:

- `ACTIVATION` – umožňuje plánovat operace vzhledem k současné operaci (např. po operaci *decode* můžeme naplánovat operaci *execute*).
- `CODINGROOT` – umožňuje určit místo, jež obsahuje instrukce, které budou dekódovány (např. operaci *decode* řekneme, aby dekódovala operaci, která je uložena v registru zřetězené linky).

Tyto dvě sekce umožňují namodelovat např. zřetězenou architekturu. Pro disassembler z toho plyne, že se bude muset upravit blok „Základní modul disassembleru v jazyku C“. Musíme do tohoto bloku přidat plánovač operací, který si bude udržovat informace o tom, která instrukce probíhá, a přijímat požadavky na naplánování dalších. Plánovat další operace bude moci jakákoli operace obsahující sekci `ACTIVATION`.

Disassemblace pak bude probíhat ve dvou, neustále se střídajících, fázích. První fáze bude pracovat s informacemi uloženými v plánovači operací. V momentě, kdy plánovač narazí na operaci, která obsahuje sekci `CODINGROOT`, dojde k přepnutí na druhou fázi. Ta začne načítat binární kód.

Odkud se bude načítat je uloženo v této operaci. Jakmile skončí načítání (tedy přijme se právě jedna instrukce), dochází opět k přepnutí zpět. Vše skončí po načtení celého vstupu nebo při chybě.

8.3 Rozšíření překladače jazyka Bison--

Z předchozí podkapitoly plyne, že má cenu generovat párové automaty pouze tehdy, obsahuje-li operace sekci `CODINGROOT`. Obecně však těchto operací může být více. Stávající překladač počítá pouze s jedním automatem, což vyhovuje případům, kdy architektura obsahuje pouze jedno místo pro dekódování operací. Standardně jsou to pětistupňové zřetězené procesory a fáze nazvaná *decode*.

Představme si však příklad sedmistupňové zřetězené linky. Ta s nejvyšší pravděpodobností bude obsahovat dvě fáze, které budou provádět dekódování. A to fáze *predecode*, která zjistí z části operačního kódu instrukce, o jakou instrukci se jedná. Po nějaké době dojde k fázi *decode*, která ze zbytku operačního kódu dekóduje operandy. V tomto případě musíme mít dva párové automaty. Jeden, který bude zpracovávat operaci *predecode*, a druhý, který bude zpracovávat *decode*. Podobných architektur lze vymyslet několik.

Pro překladač toto znamená generovat více funkcí *parse()* a *generate()*. Dobrou volbou by bylo tyto funkce nazývat stejným jménem, které má operace obsahující sekci `CODINGROOT`. Tyto funkce pak budou ve vhodné okamžiky, tedy tehdy, dojde-li k naplánování dané operace, volány plánovačem.

8.4 Podpora vstupního formátu

Přestože disassembler jako vstup bere pouze „jedničky a nuly“, je potřeba tento vstup formátovat. Proto byl vytvořen model, který zjišťuje součinnost mezi tím, co assembler vyprodukuje po předložení vstupního programu v jazyku symbolických instrukcí, a tím, co disassembler, linker a simulátor dostanou jako vstup. I přes snahu pojmou model co nejobecněji je pravděpodobné, že se do něj ještě bude zasahovat.

Formát, který je v projektu Lissom použit, je inspirován v praxi používaným COFF (Common Object File Format) formátem. Disassemblery pracující na bázi překladových párových gramatik jsou již upraveny na přijímání tohoto vstupního formátu.

Disassemblery na bázi párových konečných automatů tato úprava ještě čeká. V zásadě by šlo o to, aby bylo možné zaměňovat načítací funkce ve vygenerované části pomocí překladače jazyka Bison--. V aktuální verzi toto není umožněno. Pouze můžeme předat vestavěné funkci jméno vstupního souboru.

Pokud bychom mohli libovolně zaměňovat tuto funkci, moli bychom mít v budoucnu vstupní formát jakýkoli (současný formát je textový, je nutné pro praxi tento formát předělat do binární podoby).

8.5 Využití disassembleru v simulátorech

V projektu Lissom tvoří disassembler jádro ve všech variantách simulací. Proto jsou na něho kladeny přísné požadavky, a to především omezení podmínky na rychlost. Simulátor, ve kterém by doba simulace výkonově nižšího procesoru na výkonově vyšším byla větší, než délka běhu na reálném procesoru , by byl v praxi nepoužitelný. V projektu jsou použity dva základní typy simulátorů. Jedná se o:

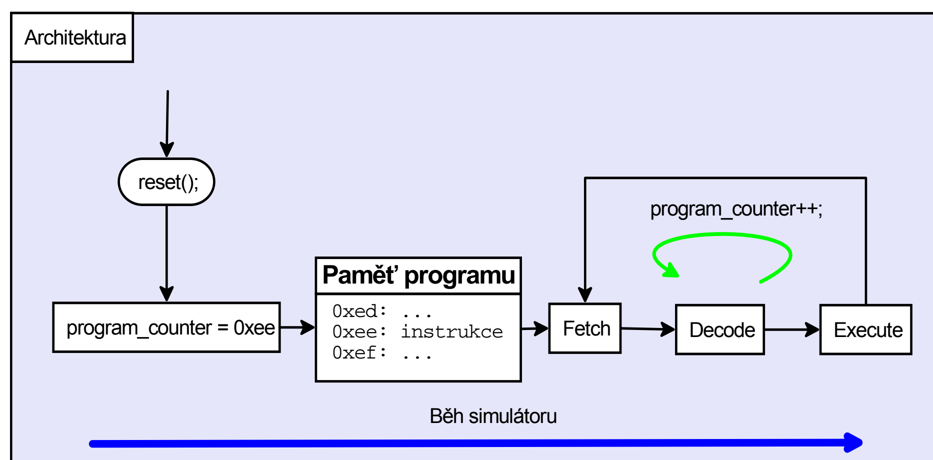
- instrukční simulátor,
- kompilovaný simulátor.

Každý má svoje kladné i záporné vlastnosti. Jednotlivé typy jsou probrány v dalších kapitolách.

8.5.1 Instrukční simulátor

Instrukční simulátor je konstruován na rychlé použití, takže je kladen důraz na krátkou dobu vytváření. Daní za to je ale delší doba simulace. Pro každou instrukci provádí načtení, dekódování a provedení behaviorální sekce jako dále nedělitelný celek. Z tohoto plyne, že instrukčním simulátorem nemůžeme simulovat zřetězené zpracování instrukcí v procesorech. Pokud se v programu objeví smyčka, jsou instrukce v ní opětovně dekódovány, což vede ke značnému zpomalení. Výhodou je naopak vlastnost, že simulátor není závislý na programu, který simuluje. Tedy pomocí instrukčního simulátoru zpracujeme jakýkoli korektní vstupní program v jazyku symbolických instrukcí, pro který se simulátor vytvořil.

Obecně simulace začíná inicializací zdrojů (program counter, …), neboli resetem, a pokračuje načtením instrukce z paměti a dalšími fázemi. Proces ilustruje následující obrázek.



Obr. 21.: Schéma instrukčního simulátoru

8.5.2 Kompilovaný simulátor

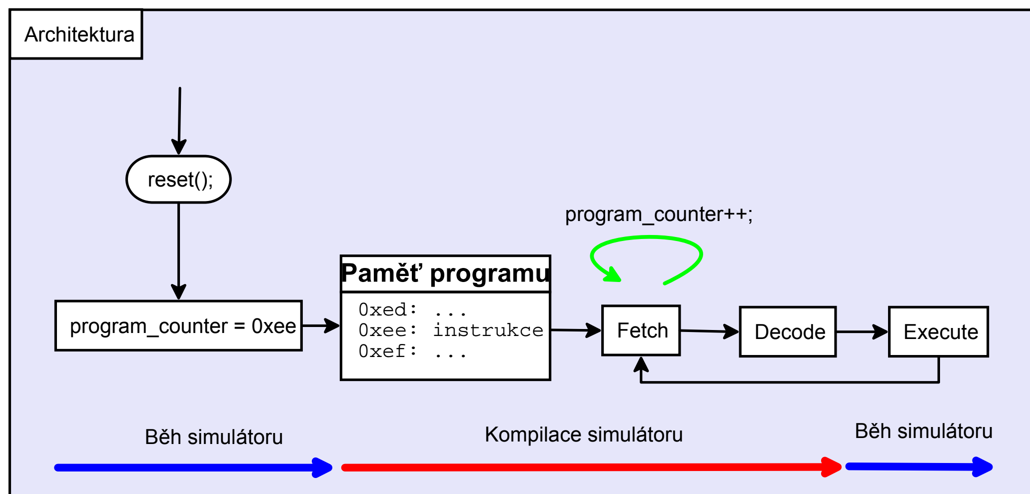
Na rozdíl od instrukčního simulátoru si kompilovaný klade za cíl co nejrychlejší simulaci za cenu delšího vytváření. Fáze načtení a dekódování instrukce probíhá pouze při vytváření simulátoru, z čehož plyne, že potřebujeme mít při generování aplikaci v strojovém kódu. Výhodou je pak velice rychlá simulace, protože za běhu nedochází k dekódovaní každé prováděné instrukce. Tedy kompilovaný simulátor je vždy svázán s jednou aplikací, a jinou na něm neodsimulujeme.

Existuje několik metod, jak simulátor vnitřně pracuje. Každá metoda má jiný poměr délek vytváření a doby běhu simulace. Pro příklad uveďme dvě metody.

- Pokud je nutnost zkrátit dobu vytváření, používá se metoda, která po dekódování jednotlivých instrukcí vygeneruje do zdrojového souboru simulátoru funkce, které obsahují jejich behaviorální sekce. Simulátor si pak lze představit jako posloupnost volání funkcí, které představují jednotlivé dekódované instrukce.
- Nebo požadujeme co možná nejrychlejší simulaci, a proto použijeme trošku jiný přístup. Fáze načtení a dekódovaní je shodná s předchozí metodou, ale poté se negeneruje volání funkce, nýbrž se její behaviorální část vloží jako makro. Výsledkem je pak dlouhý kód, který se překládá. Pak za běhu simulace nedochází ke zpožděním vlivem volání funkcí a neustálých změn kontextu.

V projektu Lissom je využita první metoda, a to proto, že se lépe ladí a je více čitelná. Nicméně v rámci optimalizací bude nutné posléze dodělat i druhou metodu.

Obecný proces kompilovaného simulátoru je na následujícím obrázku.



Obr. 22.: Schéma kompilovaného simulátoru

9 Závěr

V současné době je návrh vestavěných systémů a procesorů nepříliš automatizovaný a je potřeba skupiny odborníků, kteří provádějí návrh. Proto je nutné navrhnout a implementovat prostředky, které budou co nejvíce tento návrh umožňovat. Dalším požadavkem je i rozdělení práce do více paralelních větví, aby bylo vše rychlejší. V práci bylo zmíněno několik projektů/prostředků, které se právě o toto snaží, a v praxi jsou používané.

Jedním z nich je i projekt Lissom, kterého je tato práce součástí. Ten se snaží o komplexní vývojové prostředí, které umožní modelovat procesor, vytvářet základní prostředky procesoru a simulovat ho.

Práce se zabývá částí projektu, která generuje disassembler. Stávající vnitřní formát založený na párových překladových gramatikách je pomalý a nevyhovující. Proto bylo nutné nahradit jej vnitřním modelem novým, který nemá negativní vlastnosti předchozího, s důrazem na optimalizaci na rychlost. Byl navržen vnitřní formát, který využívá párové konečné automaty. Návrh byl prováděn vzhledem ke zkušenostem a problémům, které se vyskytly u předchozího vnitřního modelu. Nejdůležitější podmínkou bylo zrychlení běhu disassembleru. To, vzhledem k tomu, že se nakonec párové překladové automaty transformují do jazyka C ve formě velké větvící konstrukce switch, je zaručeno.

Společně s vývojem disassembleru a jeho generátorů je vyvíjen jazyk Bison-- a jeho překladač. Je nutné vést vývoj obou nástrojů co nejvíce společně, protože pak lze dosáhnout vzájemných optimalizací. Už teď je jasné, že se oba nástroje dočkají změn, které popisuje předchozí kapitola, ale jsou navrženy tak, aby těmito změnami nebyla narušena stabilita.

Současný generátor čeká na sadu testů, které mají za úkol do důsledku ověřit jeho schopnosti. Je důležité vyzkoušet modely reálných procesorů, které jsou otestovány na párových překladových gramatikách, a porovnat výsledky, případně provést modifikace, které upraví nedostatky.

V konečné podobě by vytvořený návrhový systém mohl usnadnit a zautomatizovat návrh procesorů, zlepšit ekonomickou stránku vývoje a efektivitu vývojářského týmu.

Literatura

[1] BENEŠ, M; HRUŠKA, T; ČEŠKA, M. *Překladače.* Brno: VUT, 1994. 176 s., ISBN 176-135/2-93

[2] ČEŠKA, M. *Teoretická informatika, Učební texty.* Brno: Skriptum FIT VUT Brno, 2002

[3] FALTÝNEK, P. *Podpora výuky hardware na bázi FPGA.* Brno: FIT VUT Brno, 2004. 69 s. Diplomová práce.

[4] HOFFMANN, A, MEYR, H, LEUPERS, R. *Architecture exploration for embedded processors with LISA.* Boston: Kluwer Academic Publishers, 2002. 230 s., ISBN 1-4020-7338-0

[5] HRUŠKA, T. *Komentovaný popis LISA 2.0.* Brno: Interní materiál FIT VUT Brno, 2004

[6] HRUŠKA, T. *Návrh jazyka ISAC 0.0.* Brno: Interní materiál FIT VUT Brno, 2004

[7] LUKÁŠ, R. *ISAC 0.0: Vnitřní model operační části.* Brno: Interní materiál, projekt Lissom, 2004

[8] LUKÁŠ, R. *Modely pro obecný asembler a disasembler.* Brno: Interní materiál, projekt Lissom, 2005

[9] LUKÁŠ, R; HRUŠKA, T; KOLÁŘ, D; MASAŘÍK, K. *Two-way Coupled Finite Automata.* Brno: Interní materiál FIT VUT Brno, 2006

[10] LUKÁŠ, R; HRUŠKA, T; KOLÁŘ, D; MASAŘÍK, K. *Two-Way Deterministic Translation and Its Usage in Praktice*, Brno: Interní materiál FIT VUT Brno, 2005

[11] MARTÍNEK, T. Návrh mikroprocesorů v FPGA. Brno: Přednáška č. 6 k předmětu Pokročilé číslicové systémy, 2006.

[12] PŘIKRYL, Z. *Implementace obecného zpětného assembleru.* Brno: FIT VUT Brno, 2005. 38 s. Bakalářská práce.

[13] SCHWARZ, B; DEBRAY, S; ANDREWA, G. *Disassembly of Executable Code Revisioned*. Tucson: University of Arizona, 2002.

[14] SCHWARZ, J. *Vestavěné systémy-úvod.* Brno: Přednáška č. 1 k předmětu Mikroprocesorové a vestavěné systémy, 2004

[15] VAŠÍČEK, L. *Návrh struktury obecného assembleru a zpětného assembleru.* FIT VUT Brno, 2005. 43 s. Bakalářská práce.

[16] *O počítačích a internetu* [online]. [cit. 20.4.2007]. Dostupné na WWW: <http://www.zive.cz/slovnik/>

[17] *Reengineering Wiki* [online]. [cit. 20.4.2007]. Dostupné na WWW: <http://www.program-transformation.org/Transform/ReengineeringWiki>

Seznam příloh

Příloha 1. Komplexní příklad generování párových automatů pro celistvý model procesoru.
Příloha 2. CD, které obsahuje tuto práci v elektronické podobě a zdrojové texty k programovému dílu včetně dokumentace.

Příloha č. 1

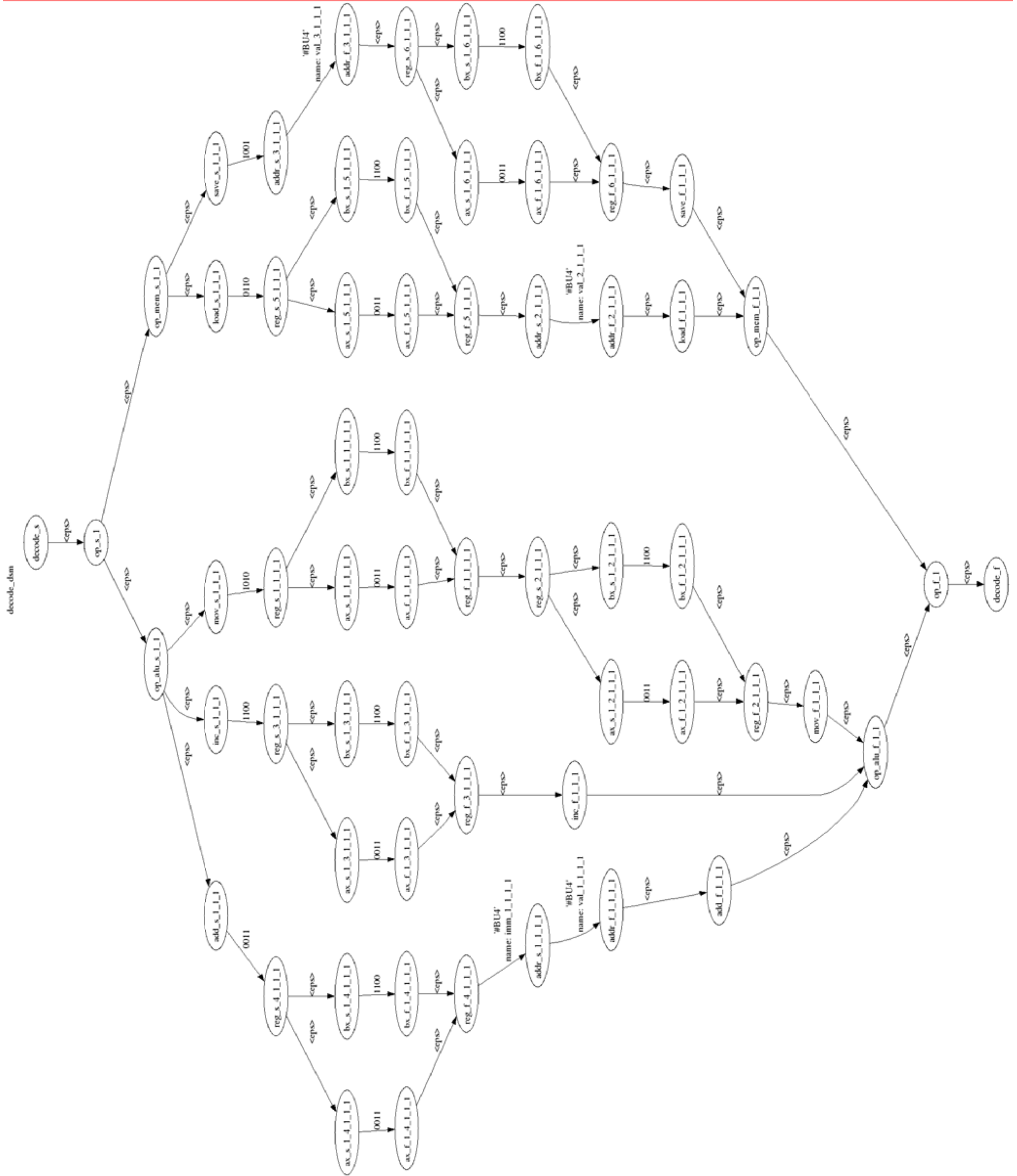
Následuje kompletní popis procesoru v jazyku ISAC a výsledný párový automat, který je výsledkem generování. Obrázek vychází z výstupu programu `gendsm`.

Model:

```
RESOURCE{
	// program counter
	PC REGISTER	bit[16]	pc;
	// program memmory
	RAM bit[8] program {
		ENDIANESS	(BIG);
		BLOCKSIZE	(8, 8);
		SIZE (255);
		FLAGS (X);
	};

	// set the default map
	MEMORY_MAP defaultmap {
		RANGE(0, 254)->program [(7..0)];
	};
}

OPERATION reset
{
	BEHAVIOR{ pc = SIMHALT; };
}

OPERATION addr
{
	ASSEMBLER { val=#U };
	CODING { val=0bx[4] };
	EXPRESSION { val; };
}

OPERATION ax
{
	ASSEMBLER { "AX" };
	CODING { 0b0011};
}

OPERATION bx
{
	ASSEMBLER { "BX" };
	CODING { 0b1100};
}

GROUP reg = ax, bx;

OPERATION mov
{
	INSTANCE reg ALIAS { dst, src };
	ASSEMBLER { "MOV" dst "," src};
	CODING { 0b1010 dst src};
}
```

```
OPERATION inc
{
      INSTANCE reg ALIAS { dst };
      ASSEMBLER {"INC" dst};
      CODING { 0b1100 dst};
}

OPERATION add
{
      INSTANCE reg ALIAS { dst };
      INSTANCE addr ALIAS { src };
      ASSEMBLER {"ADD" dst "," src "," imm=#U};
      CODING { 0b0011 dst imm=0bx[4] src 0bx[4]};
}


GROUP op_alu = add, inc, mov;

OPERATION load
{
      INSTANCE addr ALIAS { addr };
      INSTANCE reg ALIAS { dst };
      ASSEMBLER { "LOAD" dst "," addr };
      CODING { 0b0110 dst addr };
}

OPERATION save
{
      INSTANCE addr ALIAS { addr };
      INSTANCE reg ALIAS { src };
      ASSEMBLER { "SAVE" addr "," src };
      CODING { 0b1001 addr src};
}

GROUP op_mem = load, save;

GROUP op = op_alu, op_mem;

OPERATION decode
{
      INSTANCE op ALIAS { op };
      ASSEMBLER { op };
      CODING { op };
}

OPERATION main
{
      INSTANCE decode ALIAS { decode };

      CODINGROOT { decode(program[pc]); };
}
```

Přijímací automat M_1.



Generující automat M_2.