VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

PROFILER A VÍCEPROCESOROVÁ SIMULACE V PROJEKTU LISSOM

SEMESTRÁLNÍ PRÁCE DO PŘEDMĚTU VPD

Zdeněk Přikryl

BRNO 2008

Abstrakt

Aplikačně specifické procesory se staly běžnou součástí našich životů. Najdeme je v nejrůznějších podobách v nejrůznějších zařízeních. Již při návrhu a simulování aplikací na takovýchto procesorech je nutné sledovat, které části jsou v systému nadbytečné nebo naopak, které jsou přetížené a způsobují úzká místa. K identifikaci těchto míst nám slouží profiler, který při simulaci zaznamenává různorodé informace jako jsou přístupy ke zdrojům v procesoru nebo sleduje vytížení funkčních jednotek. Z těchto informací pak můžeme vhodně upravit instrukční sadu nebo strukturu procesoru. Pokud navrhujeme systém na čipu, tedy víceprocesorový systém, je situace obdobná.

Klíčová slova

Aplikačně specifické procesory, simulace, profiler, víceprocesorová sílulace, systém na čipu.

Obsah

1 Úvod

Aplikačně specifické procesory (ASIP), na rozdíl od procesoru pro obecné použití, mají upravenou vnitřní strukturu a instrukční sadu tak, aby splňovaly kriteria, která jsou na ně kladená. Jako kriterium můžeme chápat plochu, kterou zabere vlastní procesor na čipu, počet funkčních jednotek nebo spotřebu. Stejně, jako procesor pro obecné použití, i ASIP obsahuje základní zdroje. Pod zdroji si můžeme představit speciální i obecné registry, stupně pipeliny nebo paměťové bloky. Čím dál častěji se setkáváme s více procesory na čipu, tzn. systém na čipu (SoC). V tomto případě má každý procesor svoji úlohu v systému. O průběhu výpočtu si jednotlivé procesory dávají vědět například zasíláním zpráv nebo jinou formou signalizace. Takovýto aplikačně specifický SoC nebo ASIP bývá součástí nějakého většího celku, typicky nějakého vestavěného systému, který obsahuje navíc i periferie (např. D/A převodník, LCD displej,…).

Pokud navrhujeme nový ASIP nebo aplikačně specifický SOC, tak v obou případech potřebujeme před vlastním syntézou obvodu ověřit funkčnost. Jednou metodou může být verifikace návrhu a modelu nebo můžeme provést funkční testování v podobě simulace. V tomto dokumentu budeme rozvíjet pouze druhou variantu. Simulace, jak ji chápeme my v tomto dokumentu, je proces, kdy na vstup simulátoru vkládáme program ve strojovém jazyce a výstupem jsou pak hodnoty jednotlivých zdrojů procesoru. V první řadě nás zajímá vlastní proces simulace, tedy jestli se systém chová dle specifikace. Po úspěšném odladění a ověření modelu pak nastává druhá fáze, optimalizace. V druhé fázi využíváme tzv. profiler. Profiler je nástroj, který sleduje co a kde se v systému provádí, sleduje kdo jaká data kde přečetl nebo zapsal. Z těchto statistik je pak schopný vypsat jaké jednotky jsou přetěžovány nebo naopak nevyužívány, jaké je pokrytí instrukční sady atd. Z těchto výsledků jsme schopni upravovat model. Například může dojít k zjištění, že jednotka aritmetickologických operací nestíhá a je vhodné, aby došlo ke zdvojení nebo naopak. U více řezových procesorů (VLIW) můžeme zjistit, že dochází k nerovnoměrnému zatížení atp. Proces optimalizace nemusí být vždy jen na straně hardwaru. Pokud s hardwarem nemůžeme nebo nechceme manipulovat a přesto je z profilingu jasné, že dochází k neoptimálnímu zatěžování částí systému, můžeme pozměnit program, které simulujeme. Vhodnou změnou může dojít k získání lepších výsledků. Profiling může sloužit i k dalším věcem. Například z výsledků můžeme taktéž modifikovat překladač jazyka C. Vhodným příkladem jsou víceřezové procesory, kde o zatížení jednotlivých jednotek v systému rozhoduje právě překladač. Z profilingu můžeme zjistit, že překladač klade spoustu práce na jeden z řezů a ostatní v době, kdy pracuje tento řez, nedělají nic, i když by mohly.

Takto popsaný proces optimalizace modelu se dá uplatnit nejen na jednoprocesorovém systému, ale i na aplikačně specifickém SoC. Problém, který zde vzniká, je proces synchronizace a komunikace. Každý simulátor a profiler si musí během simulace vhodně vyměňovat a synchronizovat data. Jak u ASIP tak u aplikačně specifických SoC je kladen velký důraz na rychlost simulace a profilování.

2 Simulace

V následujících podkapitolách jsou uvedeny jednotlivé druhy simulací jednoprocesorového systému podle jemnosti a přesnosti simulace vzhledem k taktům procesoru. Dále je zde naznačena víceprocesorová simulace, tedy simulace SoC.

2.1 Simulace jednoprocesorového systému

Simulaci jednoprocesorového systému lze rozdělit dle jemnosti a přesnosti simulace vzhledem k taktům procesoru a reálnému prostředí. V následujících bodech jsou představeny jednotlivé typy simulátorů od nejméně přesného až po simulátor, jenž se nejvíce blíží reálnému. Zde také platí, že čím přesnější simulace, tím náročnější je vytvoření simulátoru a taktéž je náročnější proces simulace.

- Instrukční simulátor (Instruction accurate simulator) – tento simulátor bere jako výpočetní krok systému zpracování celé instrukce. Pod zpracováním instrukce si můžeme představit proces, který se skládá z načtení, dekódování, provedení a případný zpětný zápis výsledků. Pomocí tohoto typu simulátoru nelze simulovat zřetězené zpracování instrukce (pipeline architektury).
- Simulátor na úrovni taktů (cycle accurate simulator) – má jako výpočetní krok sytému jeden takt. Tedy zpracovává jednotlivé takty odděleně. Proces zpracování instrukce je rozdělen do separátních částí. Z předchozího příkladu se zpracování instrukce rozdělí do 4 částí, a to načtení, dekódování, provedení a případný zpětný zápis. Je zřejmé, že tento typ simulátoru je vhodný pro zřetězené architektury. Proces vytváření je náročný, protože je nutné zavést mechanismus, jak si bude systém uchovávat informace o historii naplánovaných událostí v systému.
- „Pravý“ simulátor na úrovni taktů (true cycle accurate simulator) – Principielně je stejný, jako simulátor na úrovni taktů, tedy základním krokem výpočtu je jeden takt. Tento princip je doveden ještě dál. Zásadní rozdíl je ve zpracovávání chování operace neboli vykonávání operace.
  V předchozích typech simulátorů bylo vykonávání operace bráno jako nedělitelný celek, který se provede najednou. To v praxi nemusí být reálné. Například operace dělení se obvykle provádí ve více taktech atp. „Pravý“ simulátor na úrovni taktů tuto skutečnost reflektuje, tedy i vlastní provádění instrukce je rozděleno na taky.

V současné době se v projektu Lissom využívá druhý typ simulátoru. Při vytváření simulátoru se využívá část párového překladového automatu [1], který vzniká při zpracování modelu procesoru. Pro modelování je použit jazyk ISAC [2]. Párový automat je také mimo jiné využit pro vytváření assembleru a disassembleru. V simulátoru se využívá jen jeden z dvojice automatů párového automatu, a to část která přijímá strojový jazyk. Párové automaty se vytvářejí na základě konstrukcí OPERATION (operace) a GROUP (grupa). Konstrukce OPERATION má různé sekce, které jsou využity při sestavování automatu. Nejvýznamnější jsou sekce popisující chování operace a způsob jejího zakódování ve strojovém jazyce.

Tuto část nazýváme dekodérem. Další část simulátoru je tvořena automatem událostí v systému [3]. Automat událostí se vytváří z jiných sekcí konstrukce OPERATION. Pod událostí v systému si můžeme představit načtení instrukce z paměti nebo naplánování dekódování na další takt atp. Spojením těchto dvou automatů dostáváme výsledný simulátor. Více informací lze nalézt v [3].

2.2 Simulace víceprocesorového systému

U více procesorové simulace se musí kromě vlastní simulace řešit ještě synchronizace a komunikace mezi simulátory jednotlivých procesorů. V projektu Lissom je použit následující koncept víceprocesorové simulace. Pro každý ASIP na SoC je vytvořen vlastní simulátor. Tyto simulátory mohou být rozdistribuovány na různé výpočetní systémy, čímž můžeme zrychlit simulaci. Po spuštění simulace se rozběhne každý simulátor na svém výpočetním uzlu nezávisle. Při simulaci pak může docházet k potřebě zapsat nebo přečíst hodnotu zdroje jiného procesoru. Aby simulátor obdržel korektní hodnoty, je nutná nějaká forma synchronizace. Pro synchronizaci hodnot zdrojů vznikl synchronizační protokol, který vychází z protokolu MSI pro synchronizaci cache u víceprocesorových systémů se sdílenou pamětí. Komunikace a synchronizace je založena na metodě zasílání zpráv a na třívrstvé architektuře. Třívrstvá architektura je tvořena presentační vrstvou (ta může mít několik forem, např. plugin pro IDE Eclipce nebo se jedná o příkazovou řádku), střední vrstvou (stará se o zpracování požadavků presentační vrstvy, získává data od simulátorů atp.) a simulační vrstvou (simulační vrstva zaštiťuje všechny simulátory). Každá vrstva může běžet na jiném výpočetním uzlu. Zprávu pak může zaslat simulátor simulátoru nebo střední vrstva může zaslat zprávu simulátoru a ten jí odpovědět. Tělo zprávy je tvořeno fragmentem jazyka XML. Podrobné informace lze najít v [4] a v [6].

3 Profiler

Jak již bylo řečeno v úvodu, profiler nám slouží k tomu, abychom získali informace o tom, jak je která část systému nebo programu vytížená. Z výsledků můžeme identifikovat úzká místa v architektuře procesoru, v programu nebo v nástrojích, které používáme pro překlad programů. Způsob podávání informací může být různý. V praxi se používají různé způsoby vizualizace, např. různé formy grafů nebo tabulek. Stěžejní pro profiler je, jaké informace se mají sledovat. Nevhodným výběrem můžeme ztratit informační hodnotu nebo získané informace špatně vyhodnocovat. V projektu Lissom budeme chtít sledovat následující informace:

- Kolikrát se jaká instrukce v instrukční sadě provedla. Tuto informaci můžeme obohatit o TOP statistiky typu: pět nejčastěji/nejméně častých prováděných instrukcí atp.
- Kolikrát se přistupovalo ke zdroji. Přístupy dělíme na čtení, zápis a spuštění.
- Kolikrát a jakým způsobem daná instrukce přistupovala ke zdroji.
- Pokrytí kódu, tzn. kolikrát se daný řádek v programu provedl.
- …

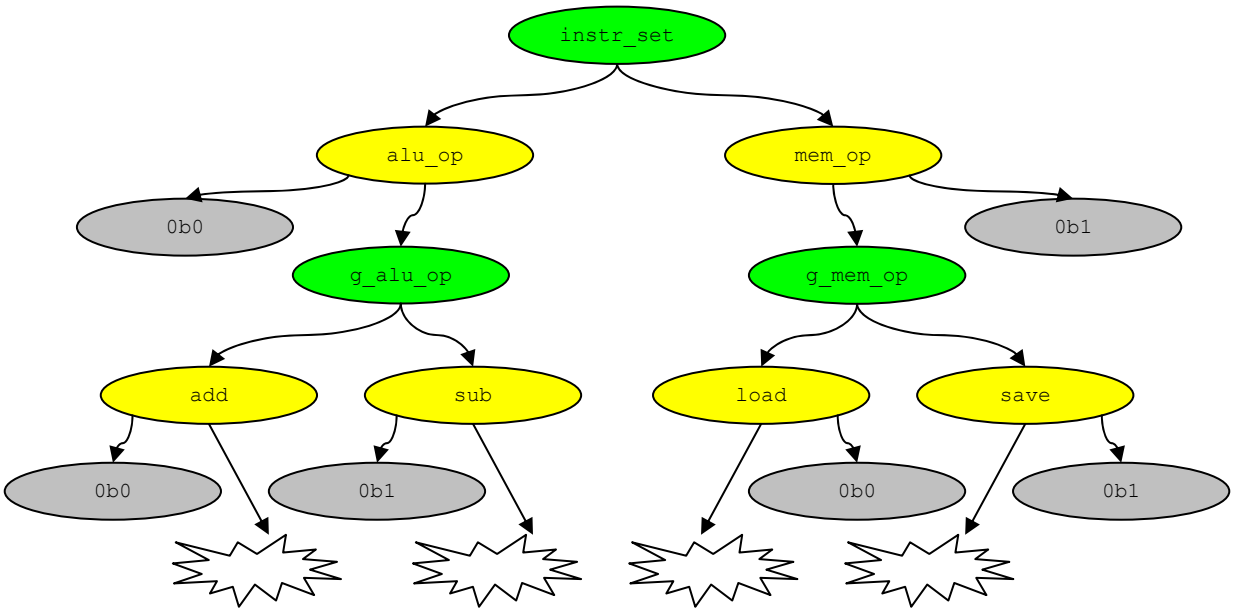
V dalším textu si představíme zvolený koncept profileru pro jednoprocesorové systémy, a pak tento koncept rozšíříme na víceprocesorové systémy.

3.1 Profiler pro jednoprocesorový systém

Základní jednotkou, ke které se většina statistik vztahuje, je instrukce. Je tedy nutné objasnit, co přesně pod tímto pojmem myslíme. Intuitivní přístup je takový, že si pod instrukcí představíme jeden příkaz v jazyku symbolických instrukcí, např. `add ax, #3`, která má předepsané chování sčítání. Tento textový zápis má pak ekvivalentní zápis v strojovém jazyku, např. `0011010`. Jeden přístup tedy je, že v popsané instrukční sadě budeme hledat takové konstrukce OPERATION, které mají BEHAVIOR sekci. Pokud takovouto operaci najdeme, prohlásíme ji za instrukci a statistiky budeme počítat vzhledem k ní. Tento postup ale není dostatečně obecný. Představme si zřetězenou architekturu, kde je chovaní všech instrukcí vloženo do operace execute a v této operaci se může vybrat chování jedné instrukce dle hodnoty nějakého zdroje. Hodnota zdroje je nastavena v předchozího kroku, ve kterém mohlo dojít k dekódování. V tomto případě by jistě docházelo k špatnému sběru a vyhodnocování statistik. Dalším příkladem může být situace, kdy by se zápis instrukce ve strojovém jazyce vytvářel pomocí dodatečných operací. Dodatečnou operací můžeme myslet operaci, která zastřešuje jistou skupinu operací a může mít v BEHAVIOR sekci vložené společné chování. Např. může zastřešovat grupu, která sdružuje operace pro práci s pamětí (viz. příloha A). Abychom se zbavili potřeby stanovovat, co to vlastně instrukce je, a tedy jistým způsobem omezovat návrháře, zavedeme následující koncept.

3.1.1 Vzory dekódovaných operací

Statistiky se budou vztahovat k tzv. vzorům. Vzor je řetězec, který jednoznačně popisuje to, co se dekódovalo. V projektu Lissom je zajištěno, že každá instrukční sada má jen jeden kořen, a to i u víceřezových architektur (je dovolen pouze jeden výskyt sekce CODINGROOT). Tyto vzory se budou vytvářet pokaždé, kdy dojde k dekódování a budou předány profileru na vyhodnocení. Protože vzor bude vždy podstromem dekódovacího stromu (konkrétně pokaždé takový podstrom, který bude mít kořen shodný s kořenem dekódovacího stromu a poslední patro bude obsahovat jeden nebo více listů, které budou shodné s listy v dekódovacím stromu) je už na vývojáři, co v tomto podstromu bude chápat jako instrukci. Statistiky budou přitom uloženy v uzlech dekódovacího stromu. Vyšší patra budou agregovat přístupy z nižších pater a přidávat přístupy z jednotlivých BEHAVOR sekcí v tomto patře. Bude tedy docházet k agregaci nebo rozdělení hodnot statistik podle toho, na jaké úrovni v stromu jsme. Následující obrázek ukazuje část dekódovacího stromu a následují jednotlivé vzory, které mohou vzniknout při dekódování. Pro vzor byla zvolena závorková notace. Otevírací závorka značí nižší úroveň ve stromu, uzavírací závorka konec této úrovně. Pokud jsou závorky prázdné, jedná se o uzel.



Obr. 1: Část dekódovací stromu

Obrázek 1 využívá zelenou barvu k znázornění konstrukce GROUP, žlutá barva je konstrukce OPERATION, šedá barva znázorňuje sekci CODING. Stejná notace bude použita i v dalších obrázcích dekódovacích stromů a podstromů. Obláček znázorňuje další možnou hierarchii. Na obrázku je vidět problematické chápání pojmu instrukce, protože každé aritmetickologické operaci je v binárním zápisu přidáván prefix 0 a paměťovým instrukcím je přidáván prefix 1.

Možné vzory, které mohou vzniknout, jsou:

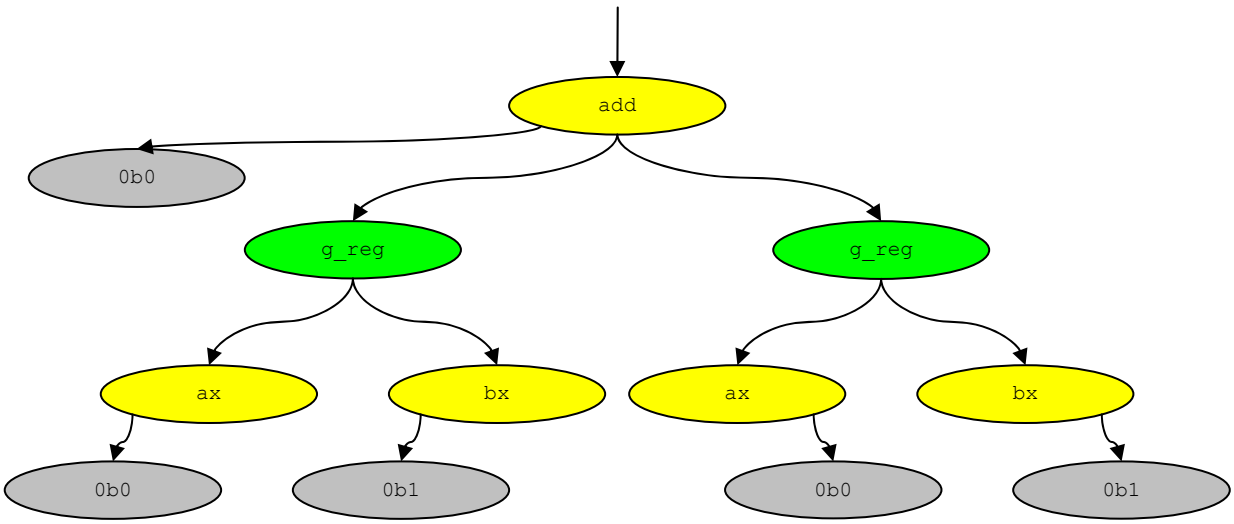
- instr_set(alu_op(g_alu_op(add(…)))) – případ, že je na vstupu 00…

- instr_set(alu_op(g_alu_op(sub(…)))) – případ, že je na vstupu 01…
- instr_set(mem_op(g_mem_op(load(…)))) – případ, že je na vstupu 10…
- instr_set(mem_op(g_mem_op(save(…)))) – případ, že je na vstupu 11…

Po ukončení profilování budou všechny přístupy a počty provedení agregovány v uzlu `instr_set`. Pokud uživatel bude chtít detailnější pohled a bude chtít vědět statistiky například pro aritmetickologické operace, posune se v dekódovacím stromu o patro níž a uvidí nyní již rozdělené statistiky uzlu `instr_set` mezi uzly `alu_op` a `mem_op`. Stejným způsobem se dá dostat na úroveň, ve které budou pro návrháře relevantní informace.

Statistiky, tedy počet přístupů k určitým zdrojům, latence operace atp. jsou určována z BEHAVIOR sekce konstrukce OPERATION. V současné situaci by se dal aplikovat přístup, který analyzuje tuto sekci a zjistí do jakých zdrojů se zapisovalo a z jakých se četlo. Pokud víme, které zdroje se používají při zpracování této instrukce, můžeme stanovit latenci operace. Každý zdroj může mít u sebe informaci o tom, jakou dobu který přístup trvá, tedy např. u pamětí může být informace o tom, že čtení trvá 1ms, ale zápis 10ms. Jednoduchými aritmetickými operacemi můžeme vyčíslit celkovou latenci operace. Jiný přístup je, že sekce BEHAVIOR bude explicitně obsahovat hlavičku se zdroji. Tato část není ještě dořešena. Při dekódování bude docházet k tomu, že k daným uzlům ve vzoru (defakto operaci, které odpovídá tento uzel) se budou ukládat statistiky. Po úspěšném dekódování se tyto statistiky přenesou do dekódovacího stromu.

Nyní rozšíříme dekódovací strom o větev, která doplní operaci `add`.



Obr. 2: Dekódovací strom pro operaci add

V obrázku číslo 2 vidíme, že operace add má dva operandy, a to dva registry. Za zmínku stojí, že v modelu procesoru je pouze jedna konstrukce OPERATION se jménem ax resp. bx. Při počítání statistik je nutné vytvářet jakoby instance těchto operací pro profiler. Konkrétně v tomto případě dvě instance operace ax resp. bx, stejně tak i konstrukce GROUP g_reg. Instance reprezentují cílový a zdrojový operand.

Možné vzory, které mohou vzniknout jsou:

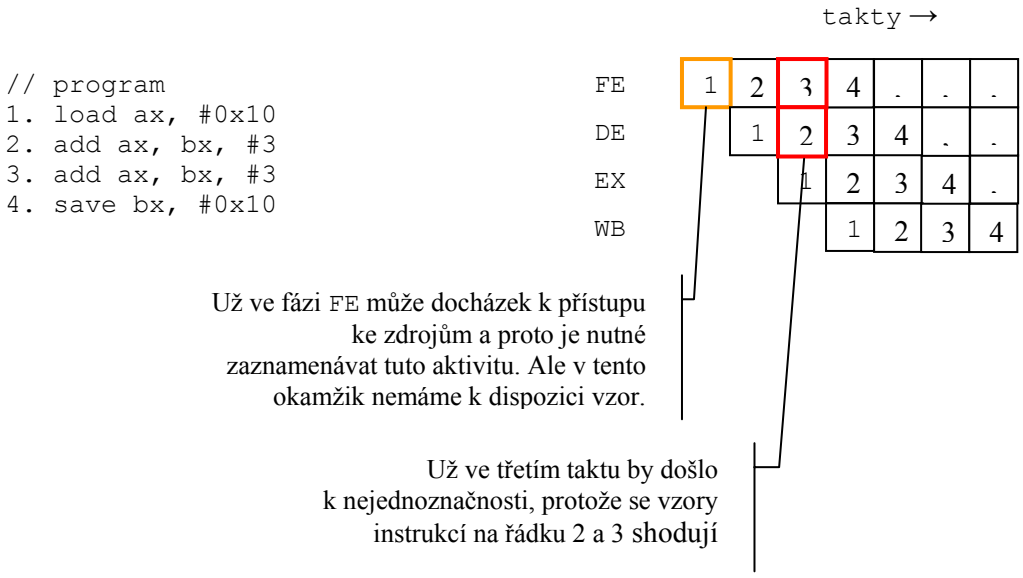
- …add(g_reg(ax())g_reg(ax())) – případ, že je na vstupu …000
- …add(g_reg(ax())g_reg(bx())) – případ, že je na vstupu …001
- …add(g_reg(bx())g_reg(ax())) – případ, že je na vstupu …010
- …add(g_reg(bx())g_reg(bx())) – případ, že je na vstupu …011

Složením vzorů, které byly uvedeny v předchozím textu, dostáváme popisované podstromy dekódovacího stromu (kořeny shodné, a poslední patro obsahuje alespoň jeden list). Oba dva dekódovací stromy vznikly z popisu architektury procesoru, která je uvedena v příloze A.

Zbavili jsme se nutnosti stanovovat návrháři, co má chápat pod pojmem instrukce. Avšak tato technika se dá aplikovat až po dekódování. Je nutné zavést aparát, který bude schopný zpracovat události před dekódováním. Ten je popsán v následující kapitole.

3.1.2 Id a vzory

Jak bylo řečeno, musíme zavést aparát, který bude zachytávat statistiky ještě před tím, než dojde k dekódování a ke zjištění vzoru. Tímto zpracujeme události v systému, jako je například událost fetch ve zřetězené architektuře. K tomu potřebujeme zjistit, jaké události mohou v systému nastat a jaké jsou jejich vzájemné vazby, vzájemné podmínění nebo opoždění Na to využijeme strom událostí [5]. Ten je zkonstruován z modelu architektury procesoru. Specifický význam má událost main. V modelu procesoru musí být právě jedna. Ta bude vždy kořenem stromu událostí a identifikuje nový takt. Více informací, jak se strom událostí vytváří lze nalézt v [5]. Každý nový takt procesoru dojde k vygenerování jednoznačného ID. Než dojde k dekódování, budou se statistiky sdružovat vzhledem k tomuto ID. Dalším důvod, proč zavádět ID je nemožnost v zřetězené architektuře rozlišit dvě shodné instrukce, které v simulovaném programu následují za sebou. Tyto instrukce budou mít stejný vzor. Tedy např. pokud jeden vzor bude ve fázi execute a druhy již ve fázi write-back, pak by docházelo k zmiňovaným nejednoznačnostem. Problémy jsou znázorněny na ilustraci číslo 1.

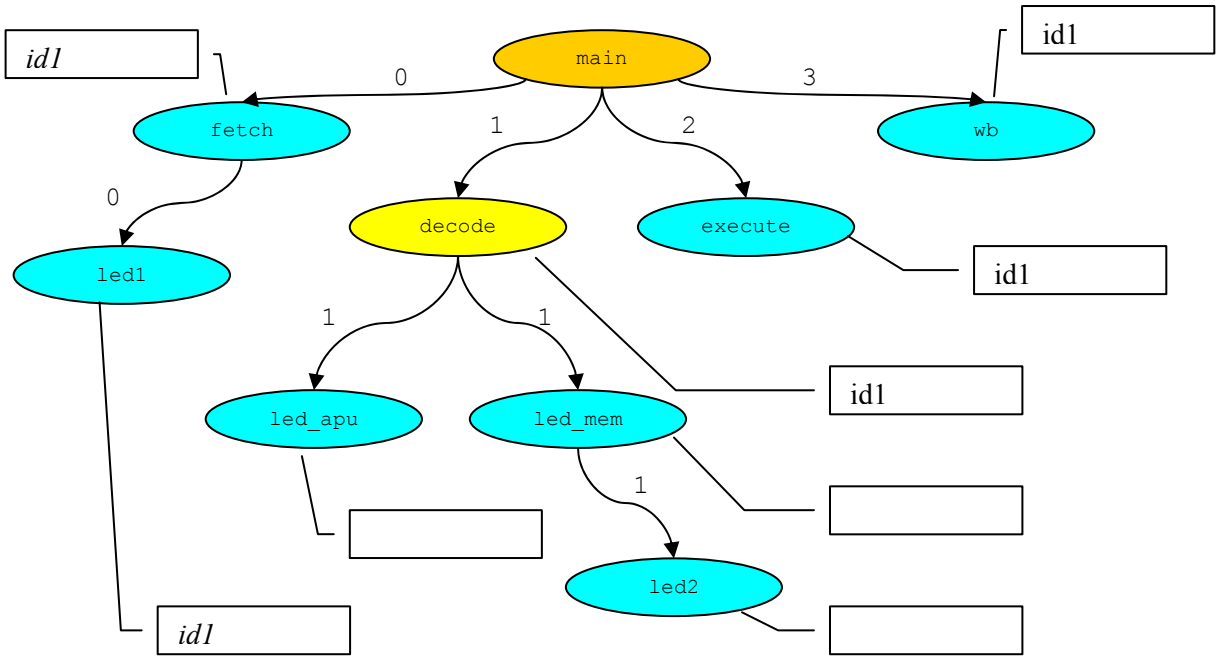


Ilustrace č.1: Ukázka možné nejednoznačnosti identifikace pomocí vzorů

V systému se tedy mohou vyskytovat různé ID, které budou dodatečně identifikovat vzor. Jakmile dojde k dekódování, dojde ke vzniku složeného klíče, kdy první částí bude ID a druhou částí bude vzor. Každý uzel v stromu událostí u sebe bude mít frontu, do které se budou ukládat jednotlivá ID. Pokud dojde k vytvoření nového ID v main, tak se toto ID vloží do front všech událostí, které jsou přímými potomky. Dále dojde k vložení tohoto ID do všech potomků přímých potomků, které mají nulové opoždění. Rekurzivně pak voláme vlkádání ID do všech potomků, které splňují předchozí podmínku. Defakto dojde k uložení ID do front všech událostí, jež nastanou v takt, kdy se provádí main a taktéž do všech front událostí, které jsou naplánovány pomocí těchto událostí. Stejný princip, tedy vkládání ID do potomků, se aplikuje v momentě, naaktivuje-li událost nebo operace v systému nějakou jinou událost v systému. Pokud je toto aktivování podmíněné, pak se ID vkládá pouze v případě, že je podmínka aktivace splněná.

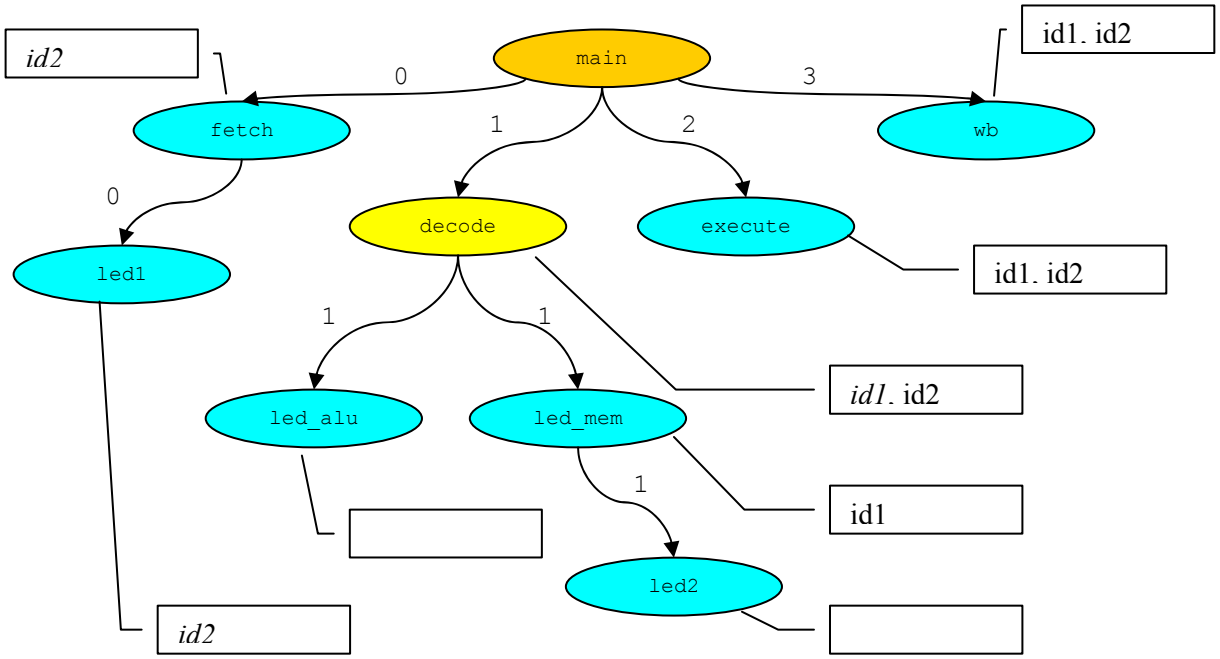
Při provádění naaktivované události dojde k vyjmutí ID z fronty. Tímto vyjmutím se signalizuje zpracování a profiler v tento okamžik ví, k jakému ID a posléze i vzoru připočítávat statistiky. Maximální velikost ID je dána maximálním zpožděním událostí oproti události main. Tato informace se dá určit ze stromu událostí. Může pak docházet k znovu použití již jednou využitého ID.

Předchozí princip je ilustrován na následující sekvenci obrázků spolu s popisy. Program, který budeme profilovat bude ten samý, jako je uveden v ilustraci číslo 1. Pro operace load a save již neuvádíme dekódovací stromy, ale již přímo vzory. Graf událostí je vytvořen z popisu architektury, která je uvedena v příloze A. Modrou barvou jsou označeny standardní události v systému, oranžovou barvou je vyznačena specifická událost main a žlutou barvou je označena událost, která obsahuje konstrukci CODINGROOT. Jedná se o událost, která spouští dekódování. Na hranách grafu je pak znázorněno zpoždění oproti předchozí události. Pokud je ID ve frontě kurzívou, signalizuje to její zpracování.



ID	Vzor	Co se zpracovává
id1	neznámý	fetch, led1

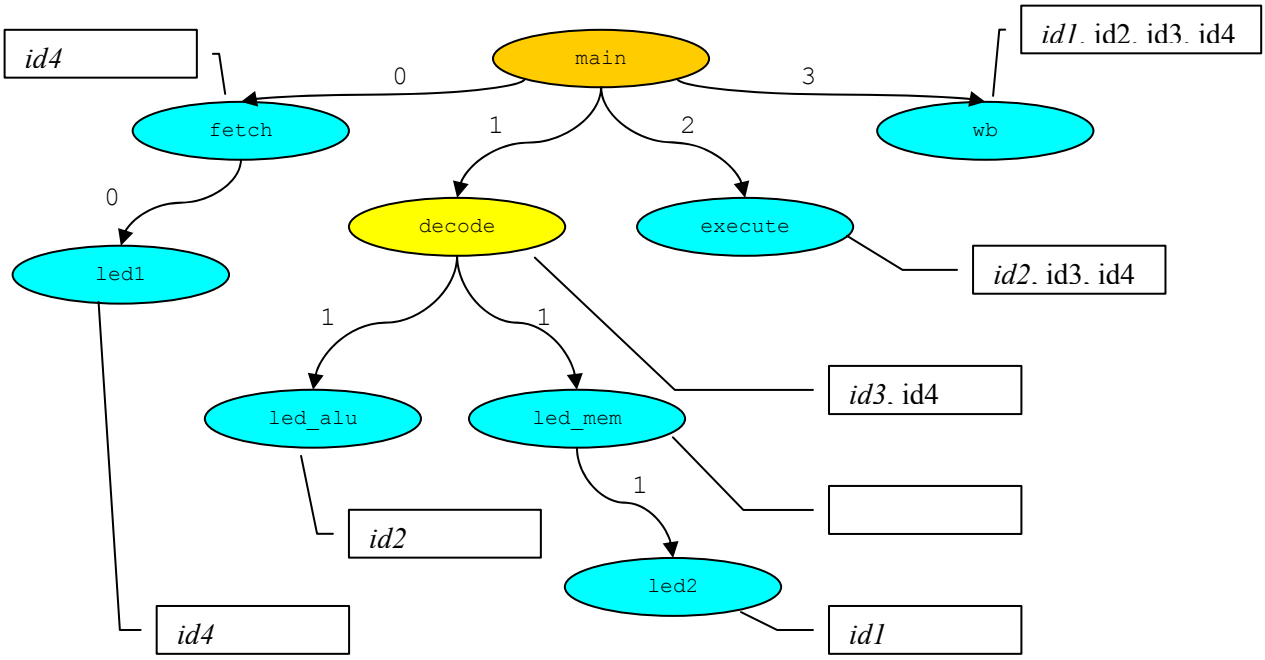
V prvním taktu dojde k vytvoření nového ID a zařazení do front podle výše popsaného algoritmu. Taktéž dojde ke zpracování událostí, které se mají provést v tento takt.



ID	Vzor	Co se zpracovává
id1	instr_set(mem_op(g_mem_op(load(g_reg(ax()) imm_8bit()))))	decode
id2	neznámý	fetch, led1

V druhém taktu dojde opět k vytvoření nového ID a vložení do patřičných front. Z fronty u událostí fetch a led1 bylo odstraněno id1, protože se již tyto události zpracovaly. Došlo také k dekódování. Protože je operace load součástí paměťových operací, došlo při dekódování k naaktivování události led_mem. Ta se ale provede až o takt později.

Přesuneme-li se do čtvrtého taktu, vypadala by situace následovně.



ID	Vzor	Co se zpracovává
id1	instr_set(mem_op(g_mem_op(load(g_reg(ax()) imm_8bit()))))	wb, led2
id2	instr_set(alu_op(g_alu_op(add(g_reg(ax())g_reg(bx())))))	led_alu, execute
id3	instr_set(alu_op(g_alu_op(add(g_reg(ax())g_reg(bx())))))	decode
id4	neznámý	fetch, led1

Vidíme, že id1 propadlo do nejzpožděnějších událostí a v dalším taktu vypadne ze systému. Poté může být toto ID opětovně použito.

3.1.3 Podmíněné dekódování

Na závěr si popíšeme zpracování podmíněného dekódování. Jazyk ISAC disponuje možností mít dekódování zpožděné nebo podmíněné. Abychom nemuseli měnit předchozí princip agregaci statistik, budeme se na podmíněnou sekci CODINGROOT dívat jako na speciální typ operace a vzhledem k této operaci pak budeme stavět dekódovací strom. Budeme vytvářet virtuální operace, které nám budou identifikovat jednotlivé dekodéry v CODINGROOT sekci. Vytváření můžeme rozdělit na následující části:

- Pokud sekce obsahuje více nepodmíněných operací, vytvoříme jednu virtuální, která bude složena z těchto částí. Ilustrace je na následujícím příkladu.

```
OPERATION decode {
       CODING ROOT {
             op1(mem[pc]);
             op2(mem[pc+1]);
             op3(mem[pc+2]);
       }
}
```

  Vzor bude složen z částí, odpovídající operacím v CODINGROOT sekci, decode(op1(…)op2(…)op3(…)).

- Pokud sekce obsahuje podmíněné dekódování, je situace více komplikovaná. Budeme muset vytvářet více virtuálních operací a grup, než jen jednu a to díky tomu, že pro podmiňování je používána konstrukce SWITCH. Tuto konstrukci můžeme chápat jako virtuální operaci, která je složena ze dvou částí. První část je část pro výraz. V této části se může objevit pouze instance grupy. Druhá část je část pro jednotlivé varianty, jedná se tedy o virtuální grupu. Tato grupa pro varianty je pak složená z dalších virtuálních operací, které odpovídají již jednotlivým variantám. Ilustrace je na následujícím příkladu.

```
OPERATION decode {
       CODING ROOT {
             SWITCH(gr(mem[pc])) {
                   CASE mem_op : op1(mem[pc+1]);
                   CASE alu_op : op2(mem[pc+2]);
             }
       }
}
```

  Vzor již nebude jen jeden, ale dva. Důvodem jsou právě varianty, které se chovají jako virtuální operace sdružené do virtuální grupy. Vzory budou následující:
  decode(SWITCH_0(gr(…)CASES(CASE_0(op1(…))))
  decode(SWITCH_0(gr(…)CASES(CASE_1(op2(…))))).

- Pokud varianta obsahuje další konstrukci SWITCH nebo obsahuje více nepodmíněných operací, dochází k aplikování prvních dvou bodů rekurzivně.

3.1.4 Implementace

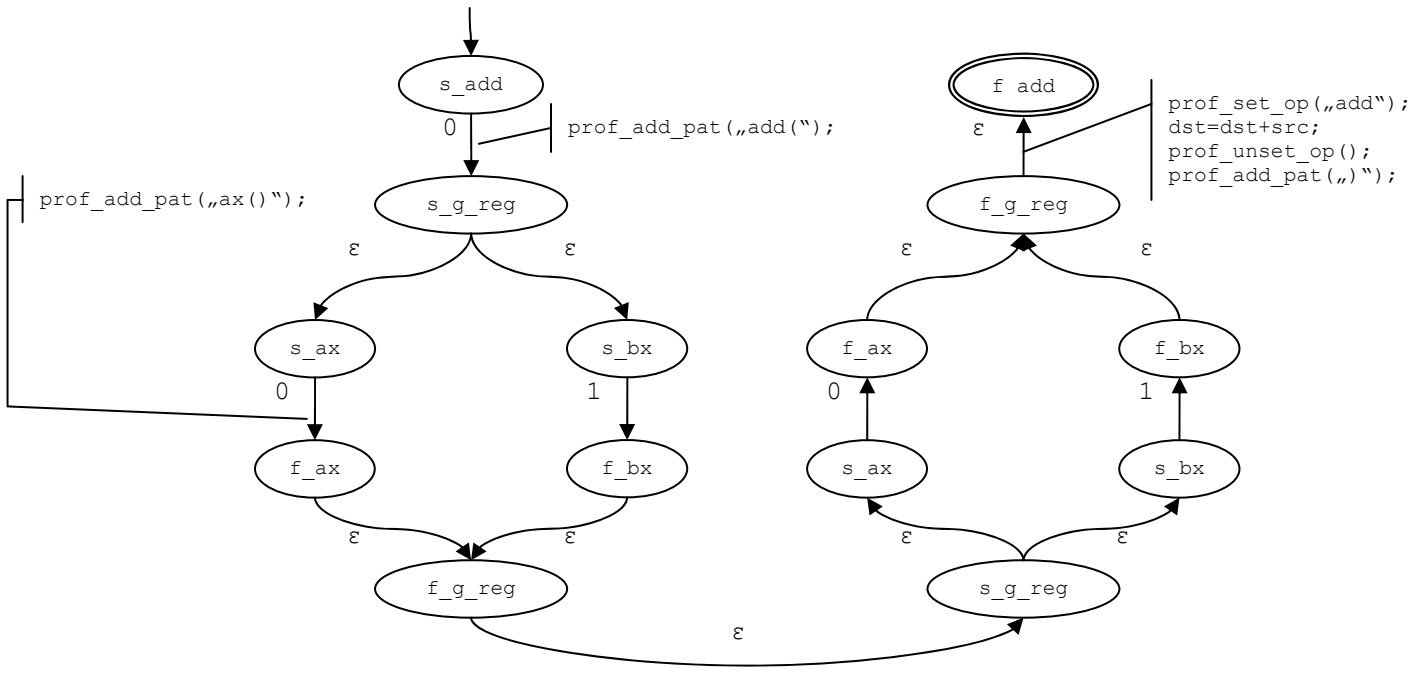
Pro implementaci profileru vyjdeme z implementace simulátoru. Ten využívá dva druhy automatů. Je to jednak automat pro dekódování a druhý automat se stará o události a aktivace v systému. Profiler ovlivní oba dva automaty. Dekódovací automat obohatíme o sémantické akce, které ponesou informaci o tom, co se dekódovalo. Tedy po dekódování, díky těmto sémantickým akcím obdržíme vzor. Pouze textová

reprezentace je nedostatečná. Potřebujeme k jednotlivým částem i informace o tom, k jakým zdrojům se přistoupilo. Proto sémantické akce dále obohatíme o toto sledování. Tyto dva typy sémantických akcí zajistí, že po dekódování budeme mít informaci o tom, co se dekódovalo spolu s informacemi o tom, které části vzoru kam přistupovaly. Na obrázku číslo 3 je znázorněn automat, který je použit pro dekódování instrukce add. Na jeho hranách je část binárního kódu který je použit pro strojový jazyk. Na stejných hranách jsou pak navěšené jednotlivé sémantické akce. Z pohledu vzorů, se dají sémantické akce rozdělit na čtyři druhy.

- Hrana vychází z s_*X* uzlu a končí v s_*Y* uzlu – vloží se sémantická akce, která vytváří vzor tvaru „*X*(„.
- Hrana vychází z s_*X* uzlu a končí v f_*Y* uzlu – vloží se sémantická akce, která vytváří vzor tvaru „*X*()“.
- Hrana vychází z f_*X* uzlu a končí v s_*Y* uzlu – nevkládá se žádná sémantická akce. Jedná se o uzly, které jsou na stejné úrovni v dekódovacím stromu a mají stejného rodiče.
- Hrana vychází z f_*X* uzlu a končí v f_*Y* uzlu – vloží se sémantická akce, která vytváří vzor tvaru „)“.

Sémantické akce, které jsou vzaty ze sekce BEHAVIOR jednotlivých operací, se provádějí na hranách, které končí v f_*Z* uzlech. Před a po těchto sémantických akcích se musí vložit dodatečné sémantické akce, které dané operaci se jménem *Z* zpracují statistiky. *Z* operace se najde v doposud vytvořeném vzoru a bude se jednat o první nalezenou takto pojmenovanou operaci směrem ke kořenu. Tímto nebude docházet k špatným identifikacím operací v případě, že se ve vzoru pracuje s více instancemi téže operace.

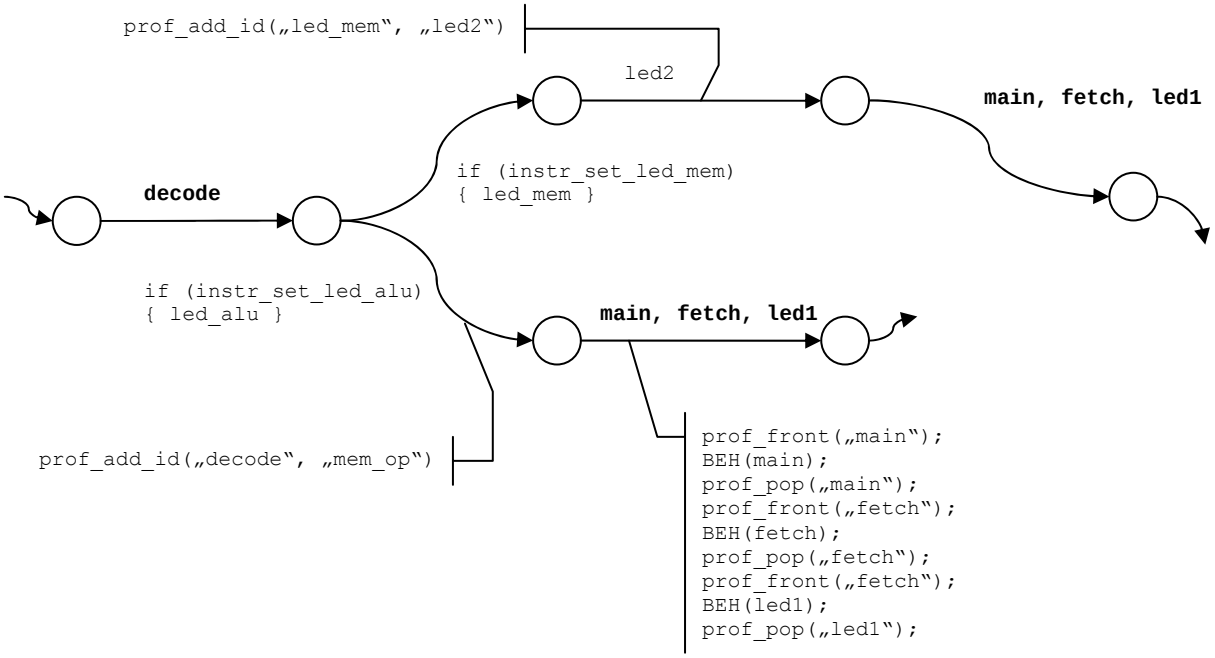
Na obrázku číslo 3 jsou pro přehlednost uvedeny jen některé.



Obr. č.3: Automat pro dekódování instrukce add s vybranými sémantickými akcemi

Další obohacení v podobě sémantických akcí přidáme do automatu událostí. Automat událostí na hranách může obsahovat plánovací sémantickou akci nebo provádění naplánované sémantické akce. Využijeme tohoto rozlišení a dle obsahu budeme přidávat dodatečné sémantické akce pro profiler. Ty budou dvojího druhu. První typ bude taková sémantická akce, která se provede před průchodem hranou. Druhým typem bude sémantická akce, která se provede po průchodu hranou. Bude-li se jednat o plánovací sémantickou akci, vložíme první druh sémantické akce, která bude vkládat ID do front. Druhý typ sémantické akce zde neupotřebíme. Bude-li se jednat o prováděcí sémantickou akci, tak jako první typ sémantické akce vložíme funkci, která bude informovat profiler o tom, jaké ID se zpracovává (ID z fronty). Jako druhý typ sémantické akce vložíme odstranění ID z fronty.

Na obrázku číslo 4 je vidět část automatu událostí, který je vytvořen z architektury procesoru v příloze A. Ten ilustruje výše uvedený algoritmus pro přidávání sémantických akcí na hrany. Plánovací sémantická akce je znázorněna pomocí normálního písma, provádění sémantické akce události je pak tučným písmem. V obou případech se jedná o řetězce na hranách. Pokud se jedná o podmíněnou aktivaci, je na hraně ještě uvedena i podmínka.



Obr. č. 4: Část automatu událostí spolu s vybranými sémantickými akcemi

3.1.5 Prezentace výsledků

Po skončení simulace profiler vyhodnotí data a zašle je prezentační vrstvě. Obdobně jako jinde v projektu Lissom, i zde volíme pro komunikaci formu zasílání zpráv. Zpráva jako taková bude tvořena fragmentem jazyka XML. Prezentační vrstva by se již neměla starat o přepočítávání nebo doplňování údajů v případě, že uživatel bude rozklikávat dekódovací strom nebo když bude chtít vidět TOP statistiky. Zpráva bude tedy obsahovat redundantní informace, ale na druhou stranu se urychlí zobrazování. Strom dekódování

bude na začátku tvořen jen jeho kořenem a po rozkliknutí se objeví další úroveň. Takto si uživatel bude brouzdat po stromu. Mimo to budou k dispozici náhledy na statistiky, které byly zmiňované v úvodu a náhledy na stavy zdrojů.

3.2 Návrh profileru pro víceprocesorový systém

V předchozí části jsme se věnovali profilingu jednoprocesorové architektury. Tento koncept musíme rozšířit na víceprocesorovou simulaci. Koncept profilingu víceprocesorových systémů bude podobný, jako u víceprocesorové simulace. Simulátory budou obohaceny o sémantické akce pro profiling a stejným principem jako pro normální simulaci můžou být simulátory rozdistribuovány na různé uzly v síti. Poté dojde k zahájení simulace. Pokud bude simulátor procesoru přistupovat ke zdroji, který bude vlastnit jiný procesor, tak se musí řídit dle synchronizačního protokolu. V našem případě se jedná o modifikovanou verzi synchronizačního protokolu MSI. Pokud lokální kopie bude platná, pak se jako latence vezme doba přístupu do lokální cache. Pokud lokální kopie hodnoty zdroje platná nebude, dojde k zaslání zprávy pro simulátor, který je vlastníkem tohoto zdroje. Zpráva bude obsahovat žádost o vydání aktuální hodnoty. Odpověď na požadavek bude obsahovat nejen aktuální hodnotu, ale i dobu potřebnou k jejímu vydání, tj. doba přístupu ke zdroji v procesoru, který je vlastníkem. Tato hodnota se pak připočítá k celkové latenci provádění operace. Na druhé straně procesor, který byl požádán o vydání hodnoty zdroje, si zaznamená do statistik, který procesor k tomuto zdroji přistupoval a inkrementuje počet přístupů. Přístupy k lokálním zdrojům se ve statistikách budou zaznamenávat stejně jako u jednoprocesorového systému. Po skončení simulace každý simulátor zašle zprávu střední vrstvě. Každá zpráva bude obsahovat informace o profilingu pro daný procesor. Střední vrstva pouze přepošle tyto zprávy presentační vrstvě, která vytvoří záložky pro každý procesor.

Tímto můžeme odhalit úzká místa v aplikačně specifickém SoC. Například tak, že pokud jistý sdílený zdroj bude často v neplatném stavu (bude docházet často k zápisům a ke čtení různými procesory), bude docházet k velkým latencím na získání jeho hodnoty. To povede k celkovému zpomalení systému. Když toto zjistíme, můžeme např. zdroj zdvojit a upravit program tak, aby jednotlivé ASIP využívali pouze určitý zdroj.

4 Závěr

V tomto dokumentu je shrnut stav víceprocesorové simulace, návrh profileru pro jedno a více procesorové systémy. Profiler pro jednoprocesorový systém využívá konceptu párových překladových automatů a automatu událostí. Přidává na vhodné hrany těchto automatů speciální sémantické akce, které zajistí sběr potřebných informaci. Instrukční sada a počty vykonání budou dostupné v rozklikávací formě dekódovacího stromu, která bude dostupná v presentační vrstvě. Jako způsob komunikace je zvoleno zasílání zpráv, které obsahují fragmenty jazyka XML. Tato zpráva jako celek obsahuje všechny informace, a to i redundantní. Tedy každé rozkliknutí bude znamenat jen vyhledání informace ve zprávě. Důvodem je to, že nechceme zatěžovat presentační vrstvu výpočty. Profiler pro víceprocesorový systém bude využívat stávající koncept víceprocesorové simulace. Tedy půjde o rozdistribuování jednoprocesorových simulátorů s profilery na uzly v síti. Po ukončení simulace dojde k zaslání výsledků střední vrstvě. Ta je jen přepošle presentační vrstvě a zobrazí separátně jednotlivé výsledky pro procesory. Oba koncepty využívají současných formálních modelů a jejich implementace a začlenění do projektu by nemělo byt příliš problematické.

Dalším krokem ve vývoji pak bude „pravý“ simulátor na úrovni taktů. Bude se jednat o rozšíření stávajícího simulátoru. Pravděpodobně se bude jednat o rozšíření stávajícího konceptu, kde stav systému je charakterizován pomocí jedné stavové proměnné centrálního řadiče. Jednotlivé sémantické akce operací, které trvají déle než jeden takt, budou rozděleny na části, které trvají jeden takt. Pro jednotlivé části bude přiřazena hodnota vnitřní stavové proměnné, která bude charakterizovat, kde v provádění sekce jsme. Poté může dojít k vzájemné kombinaci nebo sjednocení těchto stavových proměnných v jednu, která bude definovat co se má provést v prováděný takt a co v příštích taktech.

Literatura

[1] Lukáš, R., Hruška, T., Kolář, D., Masařek, K. *Two-Way Coupled Finite Automata*. Interní materiál projektu Lissom, Brno, 2006.

[2] Hruška, T. *Instruction Set Architecture C*, Interní materiál projektu Lissom, Brno, 2004.

[3] Masařek, K. *HW/SW Codesign*, Disertační práce, Brno, 2008.

[4] Moskovčák, J. *Návrh komunikačního protokolu pro generické mikroprocesorů*, Diplomová práce, Brno, 2004.

[5] Přikryl, Z. *Formalizace časového modelu jazyka ISAC*, Interní materiál projektu Lissom, Brno, 2008.

[6] Přikryl, Z., Hruška, T., Masařík, K.: *Simulation ASIP on SoC*, Interní materiál projektu Lissom, Brno, 2008.

Příloha A – Model architektury

Model architektury neobsahuje BEHAVIOR sekce a neobsahuje registry v pilelne.

```
RESOURCE {
    PC REGISTER  bit[8]  pc;

    REGISTER     bit[8]  ax;
    REGISTER     bit[8]  bx;

    RAM bit[8] program_mem {
        ENDIANESS   (BIG);
        BLOCKSIZE   (8, 8);
        SIZE (255);
        FLAGS (X);
    };

    RAM bit[8] data_mem {
        ENDIANESS   (BIG);
        BLOCKSIZE   (8, 8);
        SIZE (255);
        FLAGS (R, W);
    };

    MEMORY_MAP defaultmap {
        RANGE(0, 254)->program_mem
                            [(7..0)];
        RANGE(255, 509)->data_mem
                            [(7..0)];
    };


    REGISTER bit[8] fetch, fetch_pc;

    PIPELINE pipe
    {
        FE : (fetch, fetch_pc);
        DE : ;
        EX : ;
        WB : ;
    };
}

OPERATION reset {
    BEHAVIOR {
        pc = 0;
    };
}

OPERATION halt {
    BEHAVIOR {
        HALT;
    };
}

OPERATION led1 {
    BEHAVIOR {
        ;// kod pro led1
    };
}

OPERATION led2 {
    BEHAVIOR {
        ;// kod pro led2
    };
}

OPERATION led_alu {
    BEHAVIOR {
        ;// kod pro led_alu
    };
}

OPERATION led_mem {
    INSTANCE led2 ALIAS { led2 };
    BEHAVIOR {
        ;// kod pro led_mem
    };
    ACTIVATION {
        %1 led2;
    };
}

OPERATION wb IN pipe.WB {
    BEHAVIOR {
        ;// kod pro write-back
    };
}


OPERATION ex IN pipe.EX {
    BEHAVIOR {
        ;// kod pro write-back
    };
}

OPERATION ax
{
    ASSEMBLER {"ax"};
    CODING {0b0};
    EXPRESSION { 0; };
}

OPERATION bx
{
    ASSEMBLER {"bx"};
    CODING {0b1};
    EXPRESSION { 1; };
}

GROUP reg = ax, bx;

OPERATION imm_8bit
{
    ASSEMBLER { imm=#U };
    CODING { imm=0bx[8] };
    EXPRESSION { imm; };
}

OPERATION add {
    INSTANCE reg ALIAS { dst, src };
    ASSEMBLER { "add" dst "," src };
    CODING { 0b0 dst src};
    BEHAVIOR {
        ;// kod pro add
    };
}
```

```
OPERATION sub {
    INSTANCE reg ALIAS { dst, src };
    ASSEMBLER { "sub" dst "," src };
    CODING { 0b1 dst src};
    BEHAVIOR {
        ;// kod pro sub
    };
}

GROUP g_alu_op = add, sub;

OPERATION load {
    INSTANCE reg ALIAS { dst };
    INSTANCE imm_8bit ALIAS { imm };
    ASSEMBLER { "load" dst "," imm };
    CODING { 0b0 dst imm};
    BEHAVIOR {
        ;// kod pro load
    };
}

OPERATION save {
    INSTANCE reg ALIAS { dst };
    INSTANCE imm_8bit ALIAS { imm };
    ASSEMBLER { "save" "@" dst "," imm
};
    CODING { 0b1 dst imm};
    BEHAVIOR {
        ;// kod pro save
    };
}

GROUP g_mem_op = load, save;

OPERATION alu_op {
    INSTANCE led_alu ALIAS { led_alu };
    INSTANCE g_alu_op ALIAS { g_alu_op
};
    ASSEMBLER { g_alu_op };
    CODING { 0b0 g_alu_op };
    ACTIVATION {
        %1 led_alu;
    };
}

OPERATION mem_op {
    INSTANCE led_mem ALIAS { led_mem };
    INSTANCE g_mem_op ALIAS { g_mem_op
};
    ASSEMBLER { g_mem_op };
    CODING { 0b1 g_mem_op };
    ACTIVATION {
        %1 led_mem;
          };
}

GROUP instr_set = alu_op, mem_op;

OPERATION decode IN pipe.DE {
    INSTANCE instr_set ALIAS { instr_set
};
    CODINGROOT {
        instr_set(fetch[fetch_pc]);
    };
}

OPERATION fetch IN pipe.FE {
    INSTANCE led1 ALIAS { led1 };
    BEHAVIOR {
        ;// kod pro fetch
    };
    ACTIVATION {
        led1;
    };
}

OPERATION main {
    INSTANCE fetch ALIAS { fetch };
    INSTANCE decode ALIAS { decode };
    INSTANCE wb ALIAS { wb };
    INSTANCE ex ALIAS { ex };

    ACTIVATION {
        fetch;
        decode;
        ex;
        wb;
    };
}
```