Implementácia disassebleru pre architektúru OpenRISC 1000

Jakub Dubec
Ústav počítačového inžinierstva a aplikovanej informatiky
Fakulta informatiky a informačných technológií STU v Bratislave
Bratislava, Slovakia
jakub.dubec@gmail.com

***Abstract*—Tento dokument opisuje návrh a implementáciu spätného prekladača, ktorého vstup sú ELF komplikované pre architektúru OpenRISC 1000. Analyzuje základne opisuje črty spomínanej architektúry a kategorizuje jej inštrukčnú sadu. Budeme sa venovať aj stručnej charakteristike ELF súborov, využívaných na UNIX kompatibilných platformách. Technická realizácia je postavená na aplikačnom rámci Xamarin.macOS a Cocoa (cieľom bolo vytvorenie natívnej stolovej aplikácie).**

***Index Terms*—openrisc, elf, disassembler, inštrukčné sady, macos, počítačévé architektúry**

I. Úvod

Pod pojmom spätný preklad chápeme proces transformácie spustiteľného binárneho súboru do jazyka symbolických inštrukcií. Inštrukčná sada (množina všetkých inštrukcií) je definovaná počítačovou architektúrou, ktorá opisuje abstraktnú internú organizáciu počítača spolu s opisom jej výpočtových schopností [1]. Výrobcovia hardvéru štandardne špecifikujú, ktorú architektúru implementujú pre daný produkt.

Tento proces sa najčastejšie spája s oblasťami ako je bezpečnosť a ladenie aplikácií. Ak máme vedomosť o štruktúre a formáte spustiteľného súboru pre konkrétnu platformu, vieme tak upravovať jeho správanie. Podobne, ak vieme aké inštrukcie daná aplikácia spúšťa vieme lepšie navrhovať optimalizačné riešenia a využiť tak maximum z ponúkaného hardvéru.

Pri návrhu implementácie spätného prekladača sme uvažovali nad nasledujúcimi požiadavkami: čitateľnosť zdrojového kódu (projekt má slúžiť ako ilustračný príklad a demonštrovať aspekty nami zvolenej implementačnej metodiky), algoritmickú zložitosť, praktickú použiteľnosť a jednoduchú rozšíriteľnosť.

Dokument je organizovaný nasledovne: v prvej kapitole predstavíme architektúru OpenRISC 1000 a opíšeme štandardizovaný formát pre (nie len) spustiteľné súbory, ELF (Executable and linkable format), ktorý je vo veľkom využívaný na širokej škále platforiem a operačných systémov. V druhej kapitole navrhneme všeobecný spôsob spätného prekladu aplikácie pre spomínanú platformu. V tretej kapitole opíšeme a zdôvodníme zvolenú metodiku implementácie výsledného prekladača. V záverečnej kapitole opíšeme spôsoby overenia nášho riešenia a analyzujeme existujúce možnosti. Nakoniec načrtneme možnosti, ktorými sa môže projekt ďalej uberať.

II. OpenRISC 1000

OpenRISC 1000 je opis počítačovej architektúry pre 32/64bit mikroprocesory rodiny RISC (reduced instruction set computer), ktorá bola vyvinutá ako súčasť open-source projektu OpenRISC. Opis architektúry počíta s podporou aritmetických operácií pre čísla s plávajúcou čiarkou, vektorové operácie a kladie dôraz na jednoduchosť, nízku spotrebu a škálovateľnosť [**?**]. Spomínaný projekt je kompletne riadený komunitou a jeho súčasťou je napríklad aj opis aj implementácie spomínanej infraštruktúry OpenRISC 1200 [2], ktorá bola vypracovaný vývojármi z OpenCores.org komunity. Implementácia je formálne spísaná v jazyku HDL, ktorá sa používa ako vstup pre hardvérovú syntézu ako je ASIC alebo FPGA, prípadne pre RTL simulácie. Vďaka tomuto faktu, si projekt získal úspech v praktických riešeniach (hlavne v oblasti počítačových sietí) alebo aj na akademickej úrovni (vďaka jednoduchej simulácii - samostatný simulátor, podpora v QEMU).

Architektúra je podporovaná priamo v linuxovom jadre od verzie 3.1 [3], jej využitie bolo demonštrované na FPGA kompatibilných soft-core procesoroch []. Preklad inštrukcií podporujú kompilátory gcc a llvm.

V čase písania tohoto dokumentu bola k dispozícií špecifikácia architektúry OpenRISC 1000 verzie 1.3 z mája 2019. Podporuje lineárny 32 alebo 64 bitový logický adresový priestor a popisuje jeho fyzickú implementáciu.

Pamäť programu sa môže aj nemusí zdieľať s dátovou oblasťou, vieme tak implementovať Hardvard (pamäť údajov je oddelená od pamäte inštrukcií) aj Standford (inštrukcie a údaje sa nachádzajú v rovnakej pamäti) model architektúry.

Samotný opis architektúry je veľmi rozsiahly dokument, ktorý ide nad rámec tejto publikácie, v nasledujúcich podkapitolách sme analyzovali niekoľko pre nás kľúčových častí (vzhľadom k riešenému problému, prípadne charakteristické črty architektúry).

A. Registre

V procesore sú registre rozdelené do dvoch skupín: univerzálne registre (ktorých je 32 a ich veľkosť záleží od zvoleného adresného priestoru) a špeciálne. Špeciálne registre sú ďalej rozdelené do 32 podskupín na základe ich určenia a hardvérových závislostí. Každá skupina ma rôzne spôsoby prekladu adries, ktorá závisí od teoretického maximálneho

počtu pre danú skupinu. Takáto skupina môže obsahovať registre rôznych veľkostí, účelu, typu a môžu patriť do rôznych modulov. Niektoré špeciálne registre môžu byť prístupné iba v privilegovanom. Implementácia funkčného OpenRISC procesora vyžaduje implementáciu špeciálnych registrov aspoň zo skupiny 0. Ostatné registre sú voliteľné a ich prítomnosť závisí od prítomnosti rôznych hardvérových modulov (špecifikácia sa snaží byť modulárna a pokrývať širšie možnosti jej reálneho využitia s dôrazom na efektivitu riešenia) [4].

B. Plánovanie a prepínanie procesov

Pre efektívnejšiu obsluhu prerušení máme k dispozícii takzvané tieňované registre (shadowed register), ktoré dokážu držať dve informácie naraz. Napríklad keď nastane prerušenie, vieme tak veľmi jednoducho "zálohovať" univerzálne registre práve bežiaceho programu a vykonať rutinu obsluhujúcu prerušenie. Pri obsluhe prerušenie tak vieme značne redukovať context task-switch latenciu [5].

Táto technika sa nazýva "Fast context switch", vieme ju používať pri obsluhe prerušení alebo pri plánovaní/prepínaní procesov. Pod zmenou kontextu chápeme "zálohovanie" univerzálnych registrov. Štandardne by sme museli tieto dáta ukladať do zásobníka, čo je časovo náročné. Procesor v jednom čase dokáže držať pre každý proces svoj vlastný kontext, používa na to tabuľku s názvom CID. Zmena kontextu môže nastať obsluhou výnimky alebo úpravou obsahu registra CRX (ten je ale prístupný iba v privilegovanom režime - celý proces zmeny kontextu prebieha v privilegovanom režime). Pri správnej implementácii tohoto procesu, vieme zmeniť kontext v jednom cykle procesora. Implementácia tejto funkcionality je ale voliteľná a nemusí byť k dispozícii v každom OpenRISC procesore. Najčastejšie sa spomína pri návrhu systémov reálneho času [6].

C. Vyrovnávacia pamäť

Architektúra definuje aj relatívne komplexný model využívania vyrovnávacích pamätí, pripravený pre prostredia s viacerými vláknami. Návrh nerieši konkrétnu hardvérovú implementáciu, ale iba jej abstraktné procesy. Model počíta s využitím vyrovnávacej pamäte pre univerzálne registre, prístup k dátam v externých dátových úložiskách alebo pri spracovávaní inštrukcií programu. Tento model je prítomný v každej OpenRISC implementácií, líši sa iba konkrétnou hardvérovou implementáciou.

D. Adresovacie módy a konvencie

Inštrukcie využívajú dva adresovacie módy:

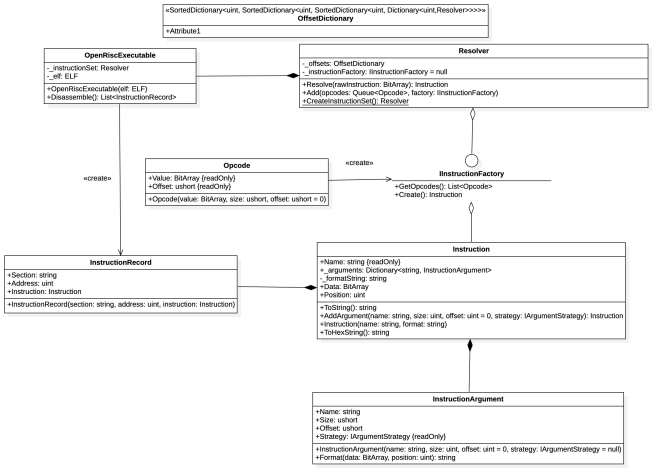

Fig. 1. Example of a figure caption.

E. Inštrukčná sada

III. ELF súbory

IV. Návrh implementácie

A. Algoritmus

B. Štruktúra aplikácie

V. Zhodnotenie

References

[1] A. Clements, *Principles of Computer Hardware*, 3rd ed. USA: Oxford University Press, Inc., 2000.

[2] A. F. A. Faroudja, I. N. I. Nouma, S. L. S. Leila, T. S. T. Sabrina, L. D. L. Dalila, and L. F. L. Fatiha, "Embedded network soc application based on the openrisc soft processor," 2012.

[3] M. Bakiri, S. Titri, N. Izeboudjen, F. Abid, F. Louiz, and D. Lazib, "Embedded system with linux kernel based on openrisc 1200-v3," in *2012 6th International Conference on Sciences of Electronics, Technologies of Information and Telecommunications (SETIT)*, March 2012, pp. 177–182.

[4] D. Lampret, "OpenRISC 1000 Architecture Manual 1 Architecture Version 1.3 Document Revision 1," pp. 1–379, 2019. [Online]. Available: https://raw.githubusercontent.com/openrisc/doc/master/openrisc-arch-1.3-rev1.pdf

[5] J. Jayaraj, P. L. Rajendran, and T. Thirumoolam, "Shadow register file architecture : A mechanism to reduce context switch latency," 2002.

[6] J. S. Snyder, D. Whalley, and T. Baker, "Fast context switches: compiler and architectural support for preemptive scheduling," *Microprocess. Microsystems*, vol. 19, pp. 35–42, 1995.