Zbornik radova Fakulteta tehničkih nauka, Novi Sad

UDK: 621.38
DOI: https://doi.org/10.24867/30BE36Vidovic

REALIZACIJA PROGRAMSKOG PREVODIOCA, ASEMBLERA I EMULATORA ZA RISC-V MIKROPROCESOR

REALISATION OF COMPILER, ASSEMBLER AND EMULATOR COMPONENTS FOR RISC-V MICROPROCESSOR

David Vidović, *Fakultet tehničkih nauka, Novi Sad*

Oblast– ELEKTROTEHNIKA I RAČUNARSTVO

Kratak sadržaj – *U ovom radu opisan je razvoj i realizacija programskog prevodioca C programskog jezika višeg nivoa na asemblerski jezik, implementacija asembler komponente zadužene za prevođenje asemblerskog koda u mašinski jezik i softverskog emulatora koji simulira rad RISC-V mikroprocesora sa jednom fazom obrade. Opisan je tok izrade korisničke aplikacije za grafički prikaz izlaznih podataka opisanih komponenti.*

Ključne reči: *RISC-V, programski prevodilac, asembler, emulator*

Abstract – *This paper describes the development and implementation of a high-level C programming language translator (compiler) into assembly language, the implementation of an assembler component responsible for translating assembly code into machine language, a software emulator that simulates the work of a single-cycle RISC-V microprocessor, and user application for graphical display of the output data of the described components.*

Keywords: *RISC-V, compiler, assembler, emulator*

1. UVOD

Postepenim prestankom važenja Murovog zakona došlo je drastičnih promena u svetu poluprovodnika i mikroprocesora. Postalo je sve teže za proizvođače tradicionalnih mikroprocesora da unaprede performanse uređaja, pre svega zbog fizičkih ograničenja prilikom fabrikacije komponente. Da bi se performanse izvršavanja postojećih i novih programa unapredile, pažnja je okrenuta ka načinu prevođenja programa iz programskih jezika visokog nivoa na mašinski jezik i na optimizacije korisničkog izvornog koda [1]. Takav nivo apstrakcije je suviše nizak za korisnika koji program opisuje u programskom jeziku visokog nivoa i ovaj posao pripada komponenti koja prevodi korisnički izvorni kod na skup asemblerskih instrukcija koje na najnižem nivou apstrakcije predstavljaju program za izvršavanje. U ovom radu opisan je razvoj jednostavnog programskog prevodioca koji prevodi podskup *ANSI* C programskog jezika na RV64I podskup instrukcija RISC-V arhitekture [2], asembler komponente koja generiše mašinski kod u vidu binarnih reči koje predstavljaju instrukcije, i softverskog emulatora koji simulira rad 64-bitnog mikroprocesora RISC-V arhitekture. Radovi koji pokrivaju sličnu temu već postoje, kao što su [3] i [4].

NAPOMENA:
Ovaj rad proistekao je iz master rada čiji mentor je bio dr Ivan Mezei, red. prof.

2. REALIZACIJA PROGRAMSKOG PREVODIOCA

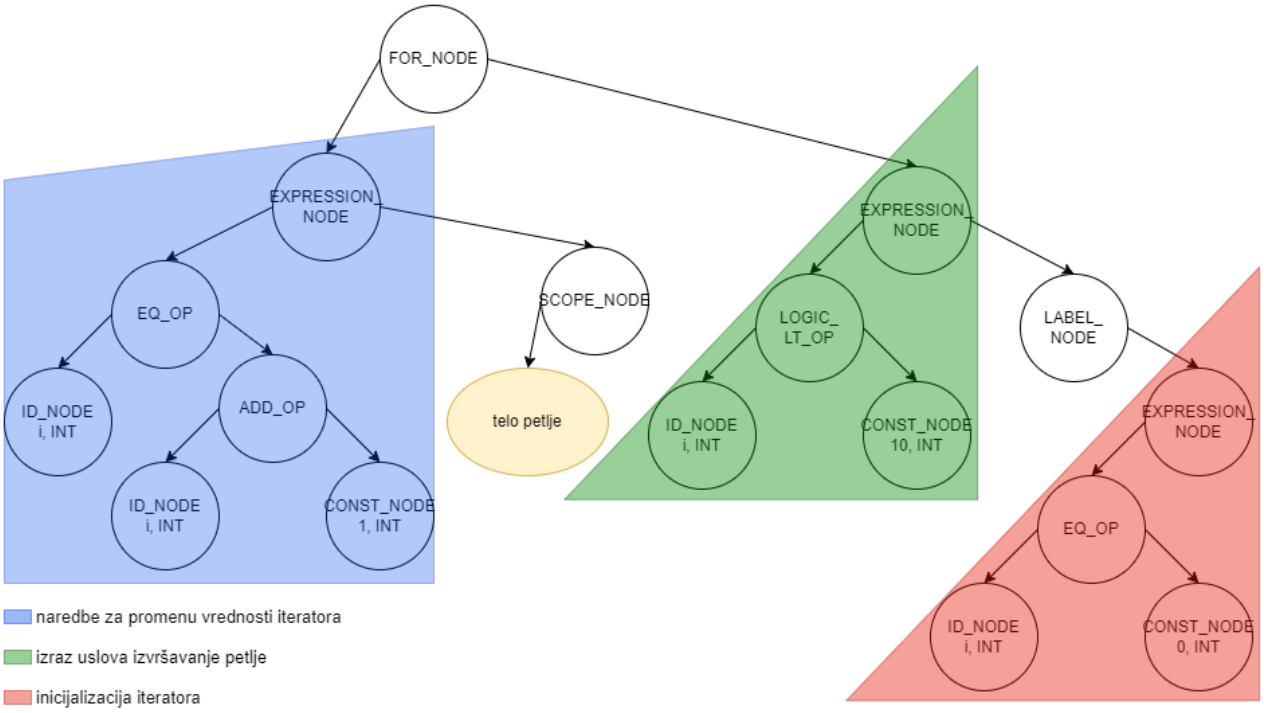
Programski prevodilac ili kompajler (eng. *Compiler*) je računarski program namenjen za prevođenje koda jednog programskog jezika u drugi programski jezik. Kod koji se prevodi zove se izvorni kod, a kod dobijen transformacijom je obično mašinski ili asemblerski kod. Struktura modernih programskih prevodilaca podeljena je na tri dela: prednji deo (eng. *front-end*), srednji deo (eng. *middle-end*) i zadnji deo (eng. *back-end*). Prednji deo je zadužen za skeniranje i parsiranje izvornog koda koji se posmatra kao ulazni tekst. Vrši se leksička analiza izvornog koda, nakon čega se proverava sintaksna ispravnost napisanog teksta (koda) kroz sintaksnu analizu i semantička ispravnost programa kroz semantičku analizu. Rezultat izvršavanja prednjeg dela prevodioca je obično međukod reprezentacija programa koja može biti tekstualna ili vidu strukture stabla, grafa ili neke druge strukture podataka. Srednji deo služi za dodatnu obradu i eventualne optimizacije generisanog međukoda, dok zadnji deo prevodioca rešava probleme kao što su alokacija registara i generisanje konačnog koda za ciljanu arhitekturu.

2.1 Realizacija leksera

Leksičku analizu izvršava lekser komponenta koja generiše tok (eng. *stream*) tokena koje prosleđuje parser komponenti. Za potrebe ovog rada korišćen je program Flex koji služi kao generator leksera napisanog u C programskom jeziku na osnovu opisa željenog leksera prateći sintaksu programa. Srž programa predstavlja deo sa pravilima za generisanje tokena. Lekser skenira ulazni tekst s leva na desno i na svako poklapanje dela teksta sa nekim pravilom izvršava skup naredbi definisanih za to pravilo i generisani token prosleđuje parseru. U najvećem broju slučajeva lekser treba samo da izbroji red i kolonu u kojoj je pronašao neko pravilo (posredstvom metode *count()* u svrhu pružanja detaljnije prijave greški korisniku) i da parseru prosledi generisani token, dok kod nekih pravila treba i da popuni polja strukture koja predstavlja vrstu podatka tokena atributima koje je pronašao u izvornom tekstu. Tako na primer za skeniran identifikator lekser u strukturi tokena *IDENTIFIER* popunjava polja sa nazivom promenjive, brojem reda i brojem kolone u kojoj je token generisan.

2.2 Realizacija parsera

Zadatak parsera jeste da na osnovu dobijenog toka tokena i opisane gramatike programskog jezika generiše stablo parsiranja. Za opis gramatike C programskog jezika i za generisanje parsera na osnovu datog opisa korišćen je program Yacc, koji generiše konačni automat u C jeziku na osnovu datog opisa gramatike. Prilikom parsiranja programa generiše se stablo parsiranja i za svako pronađeno pravilo u skupu tokena moguće je definisati skup naredbi u C jeziku koje parser treba da izvrši. Ova osobina generisanih parsera je iskorišćena u ovom radu da bi se prilikom parsiranja programa generisalo apstraktno sintaksno stablo koje predstavlja međukod ovog prevodioca.



Slika 1. *Primer podstabla FOR petlje*

Parsiranje počinje od najjednostavniji gramatičkih pravila koje zadovoljava pojedinačni token, i završava se formiranjem pravila *translation_unit* koje podrazumeva kompletan ispravno napisan C program sa definicijom bar jedne funkcije koja ima deklarisan tip podatka i telo funkcije omeđeno vitičastim zagradama.
Ukoliko parser pronađe skup tokena koji ne zadovoljavaju ni jedno pravilo generiše se greška jer izvorni kod nije sintaksno ispravan.

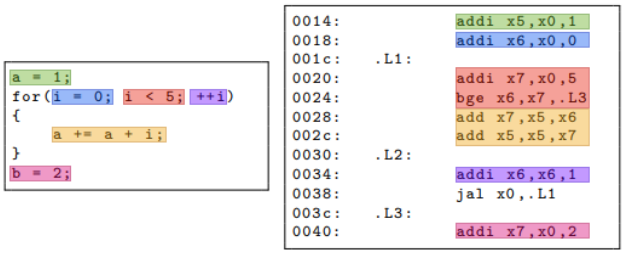
2.2.1 Tabela simbola

Veoma važna struktura podataka unutar programskog prevodioca je tabela u kojoj se čuvaju svi potrebni podaci o identifikatorima u korisničkom programu koji se prevodi. U ovom radu tabela simbola je realizovana kao heš (eng. *hash*) tabela zbog dobrih performansi prilikom pretrage tabele (što je u ovoj primeni česta pojava) i dobrog uklapanja u potrebe projekta.
Ključ na osnovu kojeg se izračunava indeks elementa u nizu jeste naziv identifikatora i za heš funkciju korišćen je algoritam FNV-1a koji koristi kombinaciju operacije XOR i množenja sa 64-bitnim prostim brojem za generisanje *heša*.
Važni atributi svaki promenjive se čuvaju u poljima strukture *ht_entry* i svaka tabela sadrži vrste koje predstavljaju pokazivače na objekte ove strukture. Za svaki domen važenja u programu kreira se posebna tabela (globalne promenjive i funkcije, i lokalne promenjive unutar svake funkcije).

2.3 Generisanje međukoda

Međukod u ovom prevodiocu predstavlja apstraktnog sintaksno stablo. Stablo je binarno i unidirekciono, što znači da svaki čvor stabla ima najviše dva naslednika. Čvorovi stabla su pokazivači na instance strukture *ASTnode* koja sa svojim poljima predstavlja kompletan opis instrukcije međukoda. Pomenuta struktura sadrži pokazivače na dva čvora stabla: *ASTnode *left* i *ASTnode *right*, koji predstavljaju njegove naslednike, odakle unidirekciona osobina stabla. Stablo se generiše u toku parsiranja, počevši od listova ka korenu, prateći unapred određena pravila za generisanje stabla, kao na primer: čvorovi identifiaktora i konstanti su uvek listovi stabla, čvorovi izraza uvek sa leve strane pokazuju na operacije unutar izraza a sa desne na prethodni izraz, čvorovi operacija uvek pokazuju na svoje operande, čvorovi funkcija sa leve strane pokazuju na čvor identifikator funkcije a sa leve strane na telo funkcije itd. Kroz stablo se prolazi rekurzivnom funkcijom i generisano je tako da rekurzivnim prolazom zagarantuje ispravan redosled programa.
Specifične su naredbe granjanja i petlje kod kojih su usvojena posebna pravila za očuvanje ispravnosti programa. Generisani čvor *IF* sa leve strane uvek pokazuje na telo petlje, a sa desne na izraz koji je potrebno izvršiti kako bi bila doneta odluka da li je potrebno da dođe do programskog skoka. Slično pravilo važi i za *IF-ELSE*, *WHILE*, *DO-WHILE* i *SWITCH* petlje. Razlikuje se samo *FOR* petlja zbog kompleksnijeg

zaglavlja, kao što je prikazano na slici 1., gde je osigurano da inicijalizacija iteratora bude prevedena u instrukcije koje su van petlje, zatim sledi provera uslova i telo petlje, a na kraju tela petlje



Slika 2. *Generisan kod za jednostavnu FOR petlju*

se vrši promena vrednosti iteratora i programski skok na proveru zadovoljenosti uslova petlje. Programski skokovi se vrše posredstvom labela u asemblerskom kodu, a pošto u trenutku generisanja naredbi granjanja kod petlji nisu poznate adrese labela programskih skokova, popunjavanje polja namenjenog za adresu labele kod ovakvih čvorova je postignuto posredstvom virtuelnog steka (eng. *stack*), odnosno LIFO strukture. Svaki put kada se generiše čvor naredbe granjanja pokazivač na njega se postavlja na vrh steka, i svaki put kada se generiše čvor labele uzima se jedan pokazivač sa vrha steka i dodeljuje mu se generisana labela. Na ovaj način je osiguran ispravan prevod ugneždenih petlji.

Ovaj programski prevodilac nad međukodom vrši i optimizaciju izvornog koda, u vidu eliminacije mrtvog koda. Mrtvi kod je kod koji se nikada neće izvršiti i obično se odnosi na deo programa koji se nalazi posle bezuslovne *return* naredbe u funkciji. Ukoliko u stablu postoji *RETURN_NODE* čvor, iz stabla se uklanjaju svi njegovi čvorovi roditelji u posmatranom telu funkcije i ovaj čvor postaje direktan naslednik čvora za telo funkcije. Na sličan način postignut je mehanizam *break* i *continue* naredbi.U toku generisanja međukoda ne vrši se alokacija registara, i mikroprocesor se tretira kao da poseduje beskonačno mnogo registara.

2.4 Generisanje asemblerskog koda

Rekurzivnim prolazom kroz apstraktno sintaksno stablo metoda *populate_IR* generiše konačni kod za ciljanu strukturu. Kod je unutar prevodioca realizovan u vidu bidirekcione spregnute liste sačinjenje od čvorova koji su pokazivači na instance strukture *IR_node* koja poseduje sve potrebne atribute za opis instrukcije. Svaki čvor liste predstavlja jednu instrukciju.

2.5 Alokacija registara

Alokacija registara u programskim prevodiocima je NP-kompletan problem [5]. Koriste se razni algoritmi za pokušaj pronalaska optimalnog rešenja za bilo koji ulazni program, kao što su analiza živih intervala ili tehnika bazičnih blokova i lineranog skeniranja [6]. U ovom radu izabran je daleko jednostavniji algoritam za implementaciju, koji ima pohlepnu osobinu. Vodi se evidencija o zauzetosti svakog dostupnog registra za lokalne promenjive, koji u RISC-V arhitekturi ukupno ima sedam, i popunjavaju se redom dok svi ne postanu puni. Od tada nadalje, kada god dođe do referenciranja promenjive čija vrednost nije u nekom registru, bira se jedan registar za prosipanje vrednosti i vrše se *load* i *store* operacije zamene vrednosti u odabranom registru. Algoritam je pohlepan jer će ostalih 6 registara uvek biti zauzeti od

```
0058: addi x5 ,x0 ,1
005c: addi x6 ,x0 ,2
0060: addi x7 ,x0 ,3
0064: addi x28 ,x0 ,4
0068: addi x29 ,x0 ,5
006 c: addi x30 ,x0 ,6
0070: addi x31 ,x0 ,7
0074: sw x6 , -44( x8)
0078: addi x6 ,x0 ,8
007c: sw x6 , -20( x8)
0080: addi x6 ,x0 ,9
0084: sw x6 , -16( x8)
0088: addi x6 ,x0 ,10
008c: sw x6 , -12( x8)
0090: addi x6 ,x0 ,11
```

Kodni fragment 1. *Primer pohlepne alokacije registara*

strane prvih 6 promenjivih u programu, kao što je slučaj u primeru prikazanom na kodnom fragmentu 1.

3. REALIZACIJA ASEMBLERA

Asembler komponenta je realizovana u C programskom jeziku, većinom posredstvom makro funkcija i metoda za dekodovanje i manipulaciju stringovima. Realizovana je kao deo programskog prevodioca, što je opravdano jer prevodilac ima samo jednu ciljanu arhitekturu. Ulazni tekst u vidu asemblerskih instrukcija obrađuje i na osnovu njega generiše izlazni fajl ekstenzije *.bin* sa mašinskim kodom programa, odnosno instrukcijama napisanom u formi binarnih reči.

4. REALIZACIJA EMULATORA

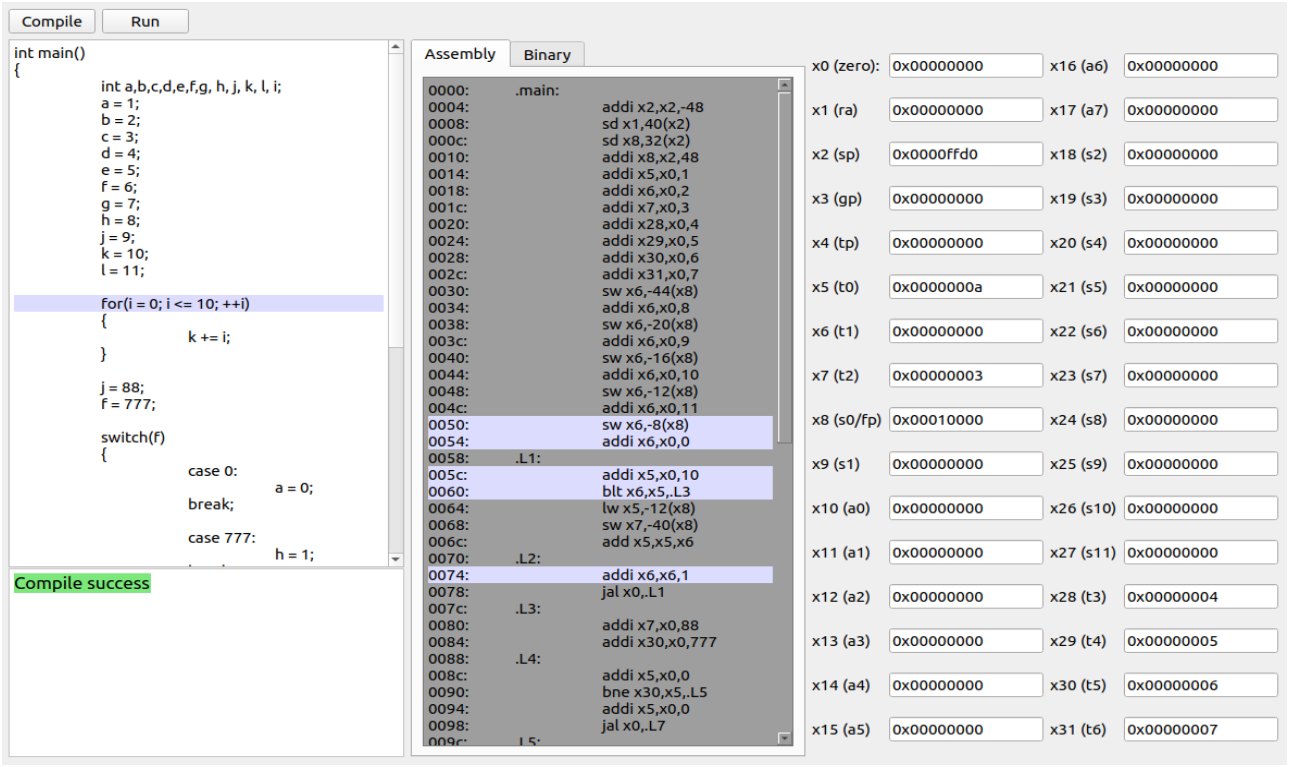
Emulator je razvijen u programskom jeziku C++ posredstvom klasa za memoriju, magistralu podataka i procesorsku jedinicu (eng. *CPU*), po uzoru na [7]. Klasa DRAM predstavlja memorijski sistem emulatora realizovan statičkim nizom u programu. Instrukcije programa za izvršavanje se čitaju i postavljaju na adresu 0x80000000 odakle počinje programska memorija. Preko klase BUS koja simulira rad magistrale podataka instrukcija sa adrese na koju pokazuje programski brojač se prenosi do procesora svaki put kada se pozove metoda *cpu_fetch*. Za simulaciju izvršavanja programa koristi se metoda *cpu_exec* koja izvršava zahvaćenu instrukciju tako što je dekoduje i vrši operaciju iz instrukcije na procesoru domaćinu. Emulator prestaje sa radom kada se iz memorije zahvati nevalidna instrukcija, što uključuje i kraj programa (memorijska lokacija nakon poslednje instrukcije u programu sadrži vrednost 0).

Pomoćne metode *dump_regs* i *dump_stack* na terminal i u izlazne fajlove ispisuju stanje svih registara i ne-nulte vrednosti na steku sistema na kraju izvršavanja programa.

5. REALIZACIJA KORISNIČKE APLIKACIJE

Korisnička aplikacija razvijena je u programskom jeziku C++ koristeći postojeći skup biblioteka za razvoj grafičkog okruženja *Qt*. Glavni i jedini prozor aplikacije

podeljen vertikalno na tri dela. Skroz levi deo sadrži tekstualni editor i tastere *Compile* i *Run*, te mali tekstualni prozor nalik terminalu za ispis potencijalni greški u programu. Pritiskom na taster *Compile* u srednjem delu generiše se asemblerski kod u jednom *tabu* i mašinski kod u drugom *tabu*, kao rezultat rada programskog prevodioca i asemblera. Pritiskom na taster *Run* u skroz desnom delu ispisuje se stanje registara procesora nakon izvršavanja programa, kao rezultat rada emulatora.



Slika 3. *Primer izgleda korisničke aplikacije*

Glavna osobina aplikacije jeste mogućnost označavanja kursorom pojedinačnih linija koda u izvornom kodu u tekst editoru, na osnovu čega aplikacija prikazuje u koje instrukcije u asemblerskom i mašinskom kodu se pretvorila označena linija izvornog koda.

6. ZAKLJUČAK

Cilj ovog projekta bilo je upoznavanje sa osnovnim tehnikama realizacije programskih prevodilaca i ostalih softverskih alata niskog nivoa. Uspešno je realizovan programski prevodilac koji prevodi podskup *ANSI* C gramatike na *RV64I* asemblerski jezik. Mana ove komponente je što ne podržava celi skup C89 standarda, kao što je rad sa strukturama, rezervisanje memorije na *heap*-u, rad sa brojevima u pokretnom zarezu, instrukcije množenja i deljenja itd. Moguća unapređenja uključuju pisanje prevodioca u C++ ili Rust jeziku zbog većeg nivoa apstrakcije i raznovrsnijih mogućnosti za realizaciju komponenti.

7. LITERATURA

[1] D. A. Patterson, J. L. Hennessy, *"Computer Organization and Design RISC-V Edition: The Hardware Software Interface"*, Morgan Kaufmann, 2017

[2] RISCV, History of RISC-V https://riscv.org/about/history/ (pristupljeno u junu 2024.).

[3] Z. Vujadzin Rakic, P. Rakic, T. Petric, *"miniC Project for Teaching Compilers Course"*, University of Novi Sad/Faculty of Technical Sciences, Novi Sad, Serbia, 2014.

[4] A. Z. Henley, *"Let's make a Teeny Tiny compiler"*, Carnegie Mellon University, 2020.

[5] G. J. Chaitin, *"Register Allocation and Spilling via Graph Coloring"*, IBM Research P.O.Box 218, Yorktown Heights, NY 10598, 1981.

[6] D. R. Koes, S. C. Goldstein, *"Register Allocation Deconstructed"*, Carnegie Mellon University Pittsburgh, PA

[7] Writing a simple RISC-V emulator in plain C, https://fmash16.github.io/content/posts/riscv-emulator-in-c.html (pristupljeno u februaru 2024.)

Kratka biografija:

David Vidović rođen je u Derventi 2000. god. Diplomski rad na Fakultetu tehničkih nauka iz oblasti Elektronike – Embedded sistemi i algoritmi odbranio je 2023.god. od kada radi kao saradnik u nastavi na Katedri za elektroniku.
kontakt: david.vidovic@uns.com