УНИВЕРЗИТЕТ У НОВОМ САДУ

**ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА У НОВОМ САДУ**

Маја Гагић

# Проширење алата за динамичку анализу кода Велгринд на инструкцијски скуп MIPS DSP ASE

МАСТЕР РАД

Нови Сад, 2013.

УНИВЕРЗИТЕТ У НОВОМ САДУ ● **ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА**
21000 НОВИ САД, Трг Доситеја Обрадовића 6

# КЉУЧНА ДОКУМЕНТАЦИЈСКА ИНФОРМАЦИЈА

| | | | |
|---|---|---|---|
| Редни број, **РБР**: | | | |
| Идентификациони број, **ИБР**: | | | |
| Тип документације, **ТД**: | | Монографска документација | |
| Тип записа, **ТЗ**: | | Текстуални штампани материјал | |
| Врста рада, **ВР**: | | Дипломски – мастер рад | |
| Аутор, **АУ**: | | Маја Гагић | |
| Ментор, **МН**: | | проф. др Никола Теслић | |
| Наслов рада, **НР**: | | Проширење алата за динамичку анализу кода Велгринд на инструкцијски скуп MIPS DSP ASE | |
| Језик публикације, **ЈП**: | | Српски / латиница | |
| Језик извода, **ЈИ**: | | Српски | |
| Земља публиковања, **ЗП**: | | Република Србија | |
| Уже географско подручје, **УГП**: | | Војводина | |
| Година, **ГО**: | | 2013 | |
| Издавач, **ИЗ**: | | Ауторски репринт | |
| Место и адреса, **МА**: | | Нови Сад; трг Доситеја Обрадовића 6 | |
| Физички опис рада, **ФО**: (поглавља/страна/ цитата/табела/слика/графика/прилога) | | | |
| Научна област, **НО**: | | Електротехника и рачунарство | |
| Научна дисциплина, **НД**: | | Рачунарска техника | |
| Предметна одредница/Кqучне речи, **ПО**: | | Компајлери, алати за анализу кода, RISC архитектура, MIPS | |
| **УДК** | | | |
| Чува се, **ЧУ**: | | У библиотеци Факултета техничких наука, Нови Сад | |
| Важна напомена, **ВН**: | | | |
| Извод, **ИЗ**: | | У овом раду, описан је процес прилагођења Велгринд алата за динамичку анализу програмског кода додавањем подршке за проширење стандардног MIPS инструкцијског скупа намењеног дигиталној обради сигнала (DSP ASE). Ово проширење дефинише инструкције за векторску обраду 32-битних података. Проширење Велгринда је проверено на системима са процесорским језгром MIPS 74K и оперативним системом Линукс. | |
| Датум прихватања теме, **ДП**: | | | |
| Датум одбране, **ДО**: | | | |
| Чланови комисије, **КО**: | Председник: | | |
| | Члан: | | Потпис ментора |
| | Члан, ментор: | | |

UNIVERSITY OF NOVI SAD ● **FACULTY OF TECHNICAL SCIENCES**
21000 NOVI SAD, Trg Dositeja Obradovića 6

# KEY WORDS DOCUMENTATION

| | |
|---|---|
| Accession number, **ANO**: | |
| Identification number, **INO**: | |
| Document type, **DT**: | Monographic publication |
| Type of record, **TR**: | Textual printed material |
| Contents code, **CC**: | Master Thesis |
| Author, **AU**: | Maja Gagić |
| Mentor, **MN**: | Nikola Teslić, PhD |
| Title, **TI**: | Extending Valgrind framework with MIPS DSP ASE support |
| Language of text, **LT**: | Serbian |
| Language of abstract, **LA**: | Serbian |
| Country of publication, **CP**: | Republic of Serbia |
| Locality of publication, **LP**: | Vojvodina |
| Publication year, **PY**: | 2013 |
| Publisher, **PB**: | Author's reprint |
| Publication place, **PP**: | Novi Sad, Dositeja Obradovica sq. 6 |
| Physical description, **PD**: (chapters/pages/ref./tables/pictures/graphs/appendixes) | |
| Scientific field, **SF**: | Electrical Engineering |
| Scientific discipline, **SD**: | Computer Engineering, Engineering of Computer Based Systems |
| Subject/Key words, **S/KW**: | Compilers, code analysis tools, RISC, MIPS |
| **UC** | |
| Holding data, **HD**: | The Library of Faculty of Technical Sciences, Novi Sad, Serbia |
| Note, **N**: | |
| Abstract, **AB**: | This paper presents the process of adapting Valgrind tools for dynamic source code analysis by adding support for extension of standard MIPS instruction set designed for digital signal processing (DSP ASE). This instruction set extension defines instructions for vector (SIMD) processing of 32bit data. Modified Valgrind tools are verified on MIPS 74K architecture and Linux operating system. |
| Accepted by the Scientific Board on, **ASB**: | |
| Defended on, **DE**: | |
| Defended Board, **DB**: President: | |
| Member: | Mentor's sign |
| Member, Mentor: | |

# SADRŽAJ

# SPISAK SLIKA

# SKRAĆENICE

**SIMD** - *Single Instruction Multiple Data*, Vektorska obrada podataka

**RISC** - *Reduced Instruction Set Computer*, Računar sa smanjenim instrukcijskim skupom

**STB** - *Set-top Box*, prijemnik DTV signala

**DSP** - *Digital Signal Processing*, Obrada digitalnog signala

**STB** - *Set-top Box*, prijemnik DTV signala

# 1. Uvod

Velgrind predstavlja okvir za dinamičku binarnu analizu programa i dolazi sa skupom alata koji mogu da otkriju širok spektar programskih grešaka.

Platforme sa ograničenim resursima su sve rasprostranjenije na tržištu potrošačke elektronike, posebno u domenu digitalne televizije, i javlja se potreba za kvalitetnom programskom podrškom koja će omogućiti izvršavanje naprednih algoritama za audio i video obradu na njima. Da bi se postigla efikasna obrada digitalnog signala, koju po pravilu karakteriše velika količina podataka koje je potrebno obraditi, često u veoma kratkom vremenskom periodu, potrebno je maksimalno iskoristiti dostupne resurse. U današnje vreme, to se najčešće postiže paralelizacijom obrade (na visokom nivou) ili upotrebom vektorskih instrukcija (na niskom nivou), a vrlo često i kombinovanjem obe tehnike. Uzimajući u obzir da programski jezici i okruženja za razvoj programske podrške još uvek favorizuju pisanje sekvencijalnog koda, SIMD optimizacije audio i video obrada su i dalje najčešći izbor prilikom optimizacije programskog koda. Kako bi se razvoj i verifikacija učinili što jednostavnijim, potrebno je obezbediti alat koji će programerima olakšati svakodnevne izazove sa kojima se suočavaju prilikom pronalaženja grešaka u ovakvom kodu.

Imajući to u vidu, kao i činjenicu da Velgrind predstavlja jedan od najpopularnijih alata za dinamičku analizu koda, koji je ujedno i projektovan sa posebnim osvrtom na SIMD proširenja instrukcijskih skupova, kao što su SSE, MMX, i NEON, za proširenje je odabran SIMD instrukcijski skup MIPS32 arhitekture, koji je stvoren za potrebe digitalne obrade signala.

Cilj ovog rada je da objasni proces proširenja alata Velgrind podrškom za drugu reviziju MIPS [1] vektorskog (engl. *Single Instruction Multiple Data*, SIMD) instrukcijskog seta namenjenog digitalnoj obradi signala (MIPS DSP ASE [2]).

Kao ciljna platforma za rad, izabrana je procesorska arhitektura MIPS i operativni sistem Linuks. MIPS je izabran jer predstavlja dominantnu arhitekturu na tržištu ugrađenih (engl.

*embedded*), namenskih uređaja za domaćinstvo, prvenstveno digitalnih televizora. U poslednjih par godina, procesorska jezgra MIPS nude paralelni potencijal kroz implementaciju vektorskih ekstenzija postojećeg instrukcijskog seta, koje, zadržavajući ugrađeni karakter, nude mogućnost realizacije kompleksnih programskih rešenja kao što je napredna obrada slike u realnom vremenu.

Rad je podeljen u nekoliko delova. U poglavljima 2 i 3, dati su pregledi glavnih karakteristika okruženja Velgrind i arhitekture MIPS, sa posebnim naglaskom na osobine instrukcijskog skupa MIPS DSP ASE. U poglavlju 4, urađena je analiza problema, dok je u poglavlju 5 opisano njihovo rešenje. Detaljan opis translacije, kao i primer translacije DSP instrukcije, dat je u poglavlju 6. U poglavlju 7, opisano je okruženje za ispitivanje, dok su u poglavlju 8 rezimirani postignuti rezultati i dati mogući pravci daljeg razvoja.

# 2. Velgrind

Velgrind je jedno od najrasprostranjenijih okruženja za dinamičku binarnu analizu programa i dolazi sa skupom alata koji mogu da otkriju širok spektar programskih grešaka. Novi alati se dodaju na jezgro Velgrinda kao programi pisani u programskom jeziku C.

Trenutna verzija Velgrinda sadrži sledeće alate:

- *Memček (eng. Memcheck)* – alat za otkrivanje memorijskih grešaka.
- *Kešgrind (eng. Cachegrind)* – alat za analiziranje problema u radu sa skrivenom memorijom (engl. *cache memory*).
- *Kolgrind (eng. Callgrind)* – proširenje *Cachegrind*-a, omogućava vizualizaciju podataka dobijenih *Cachegrind*-om.
- *Masif (eng. Massif)* – alat za analizu dinamičke memorije.
- *Helgrind* – alat za analizu konkurentnih programa, koji rade sa više niti.
- *DRD* – alat za analizu višenitnih programa; za razliku od *Helgrind*-a, zauzima manje memorije tokom svog rada.
- *Laki (eng. Lackey)* i *Nulgrind* – služe za testiranje Velgrinda i drugih alata, kao i demonstracije njihovih mogućnosti.

Okruženje Velgrind podržava sledeće platforme:

- x86/Linux
- AMD64/Linux
- PPC32/Linux
- PPC64/Linux
- x86/Darwin (Mac OS X)
- AMD64/Darwin (Mac OS X)
- S390X/Linux
- ARM/Linux

- ARM/Android
- MIPS32/Linux
- MIPS64/Linux
- x86/FreeBSD

Pre same analize koda, potrebno je pokrenuti okruženje sa željenim alatom koji će onda učitati klijentsku aplikaciju. Osnovna komanda za pokretanje Velgrinda je:

```
valgrind --tool=memcheck client_prog
```

Opcija `--tool` služi za odabir alata kojim će se obaviti analiza koda.

Nakon toga, započinje se izvršavanje klijentskog programa. Klijentski kod se, blok po blok, ponovo prevodi i dinamički, tokom izvršavanja, analizira. Ovaj proces uključuje pretvaranje mašinskog koda u među-kod (engl. *Intermediate Representation*, IR), koji se analizira željenim alatom, i prevođenje dobijenog među-koda u mašinski kod željene arhitekture, koji se pokreće po potrebi.

Tokom analize klijentskog koda, jezgro Velgrinda najviše vremena potroši na proces prevođenja originalnog mašinskog koda u među-kod, dodavanje instrukcija za analizu među-koda (samim tim i originalnog koda), kao i na proces pretvaranja novog među-koda u mašinske instrukcije ciljne platforme, koje će se izvršavati po potrebi. U zavisnosti od alata koji se koristi, analiza koda pomoću Velgrinda je 20-100 puta sporija nego kad se program pokreće bez posredstva Velgrinda.

Velgrind je podeljen u dve celine; jezgro Velgrinda i Vex (prevodilac JIT tipa u okviru Velgrind sistema). Vex je zadužen za dinamičko prevođenje koda i pozivanje alata, a jezgro za ostatak (raspoređivanje procesa, vođenje računa o dodeljenoj memoriji, itd.). U ovom radu, fokus je stavljen na Vex, uz manji osvrt na jezgro Velgrind okruženja. Programska podrška Velgrinda je logički podeljena na dve suštinski različite celine: *host* i *guest*. Host deo predstavlja spregu između Velgrindove interne reprezentacije (IR) i ciljne arhitekture. U ovom radu, ova celina je zadužena za mapiranje IR instrukcija na mašinske instrukcije koje se mogu izvršiti na procesoru baziranom na MIPS32 arhitekturi sa podrškom za DSP. Za razliku od host dela, guest deo predstavlja spregu između klijentskog programa i interne Velgrindove reprezentacije. Ovaj deo modeluje instrukcije MIPS32 DSP instrukcijskog skupa IR instrukcijama.

Velgrind deli originalni kod u sekvence koje se nazivaju osnovni blokovi. Osnovni blok je pravolinijska sekvenca mašinskog koda na čiji se početak skače, a koja se završava skokom, pozivom funkcije ili povratkom. Programski kod svakog programa koji se analizira ponovo se prevodi na zahtev, pojedinačno po osnovnim blokovima, neposredno pre samog izvršavanja osnovnog bloka. Blokovi sadrže niz iskaza koji predstavljaju operacije sa posledicama, kao što

su pisanje u registre, čuvanje podataka u memoriji ili dodela vrednosti pomoćnim promenljivama. Ovi iskazi se sastoje od izraza koji nemaju posledice. Tu spadaju konstante, čitanje iz registara, dobavljanje podataka iz memorije, kao i aritmetičke operacije. Ako se za primer uzme jednostavan iskaz za čuvanje podatka u memoriji, on bi sadržao jedan izraz za adresu lokacije na koju treba upisati podatak, i jedan izraz koji predstavlja vrednost podatka koji treba sačuvati. Izrazi mogu biti proizvoljno kompleksna stabla (izrazi se mogu naći unutar drugih izraza), ili se mogu linearizovati upotrebom iskaza koji upisuju vrednosti u pomoćne promenljive. Veličina osnovnog bloka je ograničena na maksimalno 60 mašinskih instrukcija. Na procesorima baziranim na MIPS arhitekturi, instrukcije skoka i grananja imaju takozvano „odloženo izvršavanje“, što znači da se prilikom njihovog izvršavanja izvršava i instrukcija koja se nalazi neposredno iza instrukcije grananja ili skoka. U slučaju da je poslednja (60.) instrukcija osnovnog bloka instrukcija grananja, Velgrind učitava i instrukciju koja se nalazi neposredno iza nje. Time se omogućava konzistentno izvršavanje programa koji se analizira, kao i u slučaju da se program izvršava bez posredstva Velgrinda.

Velgrindova interna reprezentacija, IR, nezavisna je od bilo koje arhitekture i ima osobine instrukcijskih skupova sa smanjenim brojem instrukcija (RISC). Svaka primitivna operacija radi tačno jednu stvar (mnoge kompleksne instrukcije se raščlanjuju u više operacija) i, kada se među-kod linearizuje, sve operacije rade isključivo nad pomoćnim promenljivama i konstantama. Za potpunu reprezentaciju svih standardnih operacija nad celobrojnim, vektorskim (SIMD) i podacima u pokretnom zarezu, kao i njihovih verzija za podatke različitih veličina, Velgrind obezbeđuje više od 200 primitivnih operacija. Iako je ovoliki skup pomoćnih operacija dovoljan za skoro sve instrukcije, neke arhitekture imaju specifične instrukcije koje nije moguće razložiti na standardne IR operacije. Jedan od takvih primera je instrukcija `cpuid` x86 arhitekture. Za simuliranje ovakvih instrukcija, koriste se pomoćne funkcije pisane u C programskom jeziku.

Iako skup IR operacija sadrži i vektorske operacije, zbog specifičnosti nekih vektorskih instrukcija instrukcijskog skupa MIPS DSP ASE, nije moguće sve primitivne IR vektorske operacije direktno mapirati na funkcionalno iste DSP instrukcije. Ovaj problem je detaljno opisan u poglavlju 5.1.3.

Prevođenje koda se obavlja u sledećih 8 koraka, detaljno objašnjenih u [3]:

1. ***Disasembliranje*** – Proces prevođenja mašinskog koda ciljne arhitekture u IR, koji ne zavisi ni od jedne arhitekture. Svaka instrukcija se nezavisno od ostalih disasemblira u jedan ili više IR izraza, koji u potpunosti ažuriraju stanje *guest* registara.

2. ***Optimizacija IR*** – Primena standardnih optimizacija programskih prevodilaca na IR; uklanjanje redundantnog koda, eliminacija podizraza, jednostavno odmotavanje petlji i sl.
3. ***Instrumentacija*** – Pozivanje odabranog alata za analizu koda. U ovom koraku, ubacuju se nove IR instrukcije, koje pomažu alatu prilikom analize, u kod dobijen prethodnom optimizacijom. Dodate instrukcije ne narušavaju konzistentno izvršavanje originalnog koda.
4. ***Optimizacija IR*** – Optimizacija slična prethodnoj, ali nešto jednostavnija.
5. ***Gradnja stabla*** – Linearan IR se organizuje u stablo kako bi se pojednostavio sledeći korak.
6. ***Odabir instrukcija*** – Zamena IR instrukcija mašinskim (u ovom koraku se još uvek koriste virtualni registri).
7. ***Dodela registara*** – Zamena svakog od virtualnih registara, korišćenih u prethodnim koracima, nekim od, za to određenih, registara ciljne arhitekture.
8. ***Asembliranje*** – Generisanje mašinskog koda ciljne arhitekture kodovanjem prethodno generisanih instrukcija i njihovim čuvanjem u osnovnom memorijskom bloku.

Za izvršavanje svih prethodno opisanih koraka, osim instrumentacije koju obavlja alat, zaduženo je jezgro Velgrinda. Sav prevedeni kod se čuva u za to namenjenoj tabeli prevoda fiksne veličine koja se zbog svoje veličine (sadrži nešto više od 400 hiljada polja) retko skroz popuni. Ukoliko do toga dođe, pojedini blokovi prevedenog koda se iz nje brišu po principu FIFO reda.

U poglavlju 6, navedeni koraci translacije biće razmatrani kroz primere prevođenja nekoliko instrukcija iz instrukcijskog skupa MIPS DSP ASE.

## 2.1 Memček

Memček je alat za dinamičku analizu koda i služi za otkrivanje grešaka koje se javljaju prilikom rukovanja memorijom. Ovaj alat, između ostalog, prati adresiranje svih memorijskih lokacija kao i zauzimanje dinamičke memorije i vrši proveru definisanosti podataka. Na ovaj način može otkriti greške poput pristupanja nealociranoj memorijskoj adresi, curenje memorije, ukoliko zauzeta memorija nije pravilno oslobođena ili greške koje se javljaju prilikom korišćenja nedefinisanih vrednosti i to sa preciznošću na nivou bita.

- **Vbit test**

Vbit test je test, u okviru alata Memček, koji služi za proveru definisanosti vrednosti i zasniva se na principu čuvanja meta-podataka za svaki bit podataka nad kojim se radi (i onih u

registrima, i onih u memoriji) u tzv. *shadow* registrima. Ovi meta-podaci se zovu biti definisanosti, a označavaju kao V-biti. Svaki od ovih V-bita „boji“ jedan od bita na istoj toj poziciji u originalnom registru, ili memorijskoj lokaciji, i pokazuje da li on ima ispravno definisanu vrednost. Takođe, svaka od operacija koja kao rezultat daje novu vrednost, obojena je operacijom koja računa V-bite na izlazu, na osnovu V-bita na ulazu operacije.

Teorijski, moguće je definisati ispravnu transformaciju V-bita za svaku instrukciju instrukcijskog skupa neke arhitekture, ali zbog toga što su neki instrukcijski skupovi veoma kompleksni, ovo se radi nad operacijama iz interne reprezentacije Velgrinda koja ima RISC osobine, a ujedno je i nezavisna od bilo koje arhitekture.

Dodatna pogodnost IR-a je podrška za korišćenje beskonačnog broja virtualnih registara, u odnosu na koje se vrši početni prevod nekog koda. Memček vrši instrumentaciju ubacujući nove IR instrukcije koje koriste virtualne registre, dok jezgro Velgrinda ima zadatak da izvrši odabir pravih instrukcija, koje će se izvršavati na ciljnoj platformi u odnosu na IR instrukcije.

I pored toga što Vbit test udvostručuje količinu memorije koja se koristi, izuzetno je koristan jer otkriva greške sa preciznošću na nivou bita, koje često nije moguće otkriti regularnim ispitivanjem.

# 3. Arhitektura MIPS

MIPS je arhitektura sa procesorom smanjenog skupa naredbi (RISC) i jedna je od dominantnih arhitektura na tržištu ugrađenih (engl. *embedded*) uređaja potrošačke elektronike (digitalni televizori, prijemnici digitalnog TV signala (engl. *set-top box*, STB), igračke konzole, itd.). Cena integrisanih kola zasnovanih na ovoj arhitekturi, njihova niska potrošnja energije, slaba disipacija toplote i dostupnost alata za razvoj programske podrške čine je pogodnom za različite namene.

Pored standardnog instrukcijskog skupa, pojedini čipovi bazirani na arhitekturi MIPS nude i njegovo proširenje namenjeno digitalnoj obradi signala. Ovo proširenje specijalne namene (DSP ASE) nudi vektorske instrukcije koje olakšavaju i ubrzavaju obradu velike količine podataka istog tipa, što je najčešće slučaj u algoritmima za obradu digitalnog signala.

## 3.1 MIPS32 DSP(r2) ASE

MIPS32 DSPr2 ASE predstavlja drugu reviziju proširenja standardnog MIPS-ovog instrukcijskog skupa namenjenog digitalnoj obradi signala (engl. *Digital Signal Processing*, DSP) i od prve revizije instrukcijskog skupa se razlikuje samo u tome što poseduje veći broj instrukcija, suštinske razlike ne postoje. Dizajnirano je da odgovara velikom broju uobičajenih DSP algoritama, kao i drugim algoritmima koji obrađuju podatke na sličan način (prvenstveno vektorski). Pored standardnih opcija koje nude i druge DSP ekstenzije, poput operacija nad razlomljenim brojevima i vektorskih aritmetičko-logičkih operacija nad 32-bitnim registrima, MIPS32 DSP ASE sadrži i instrukcije kojima se na jednostavan način vrše i složenije operacije specifične za DSP aplikacije, poput kompleksnih množenja, umetanja i izdvajanja promenljivog broja bita, realizaciju virtualnih kružnih bafera, itd. MIPS32 DSP ASE podržava tri tipa podataka: označene 32-bitne, označene 16-bitne i neoznačene 8-bitne podatke. Poslednji format je posledica činjenice da video aplikacije najčešće koriste osmobitne neoznačene vrednosti za

čuvanje podataka. Zbog toga je, pomoću instrukcija iz ovog seta, moguće vršiti standardnu obradu 32-bitnih podataka, ili vektorsku obradu 2x16-bitnih ili 4x8-bitnih podataka, što je od velike koristi pri obradi digitalnog audio i video signala gde za potrebne proračune često nije neophodna 32-bitna preciznost. Dodatno, kako većina DSP algoritama koristi aritmetiku nepokretnog zareza, instrukcijski set sadrži brojne instrukcije za standardnu i vektorsku obradu podataka u ovom formatu, uz opcionu saturaciju ili zaokruživanje prilikom prekoračenja opsega.

Da bi se omogućile sve pomenute opcije, MIPS DSP ASE dodaje četiri nova registra u već postojeći skup. Pored postojećeg akumulatorkog para 32-bitnih `HI` i `LO` registara, dodata su još tri čime su dobijena četiri 64-bitna akumulatora koja se koriste prilikom izvršavanja operacija koje koriste akumulaciju (jedna od najčešćih takvih operacija u DSP obradi je konvolucija). Četvrti dodati registar je 32bitni kontrolni registar, `DSPControl`, koji čuva statusne bite potrebne za podršku novih instrukcija. Na primer, polje *ccond* čuva uslovne bite koji se postavljaju prilikom izvršenja instrukcija poređenja, a koriste ih instrukcije za permutaciju bajtova.

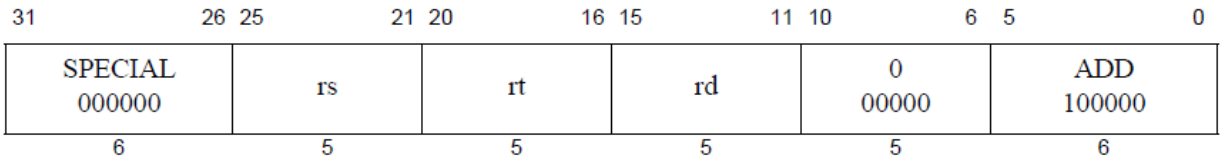
Opšte karakteristike skupa su sledeće. Skup sadrži 125 instrukcija, od čega 65 standardnih aritmetičkih (sabiranje, oduzimanje, …), 18 instrukcija pomeranja, ali i 36 instrukcija za manipulaciju sadržajem registara. Saturaciju podržavaju 42 instrukcije, a zaokruživanje 18.

Iako u okviru Velgrind IR-a postoji podrška za 32-bitne SIMD operacije, ne postoji direktno preslikavanje svake IR operacije na odgovarajuću MIPS DSP instrukciju. Jedan od razloga za to je postojanje `DSPControl` registra koji mora da se ažurira kod većine DSP instrukcija. Od svih IR operacija realizovanih za MIPS instrukcijske skupove (osnovni i DSP ASE), oko 40% nije bilo moguće direktno mapirati na neku od instrukcija MIPS instrukcijskih skupova.

# 4. Analiza problema

Okruženje Velgrind je prethodno već bilo prilagođeno arhitekturi MIPS32, kao što je opisano u [4], i tada su odabrani fizički registri koje će Velgrind koristiti prilikom izvršavanja prevedenog koda, kao i registri u kojima će se uvek nalaziti pokazivači na stanje klijentskih registara, originalan kod i sledeći blok koji treba izvršiti. MIPS DSP instrukcije koriste isti registarski stek kao i instrukcije iz osnovnog skupa, pa je u njega bilo potrebno dodati nove registre (četiri 64-bitna akumulatora i 32-bitni kontrolni registar).

Sl. 1 prikazuje primer polja u jednoj instrukcijskoj reči MIPS instrukcije.




Sl. 1 – Polja u jednoj instrukcijskoj reči MIPS instrukcije

Za razliku od osnovnog instrukcijskog skupa, gde su instrukcije opisane sa nekoliko različitih vrednosti *opcode* polja instrukcije (biti 31..26 instrukcijske reči), ceo DSP ASE instrukcijski skup (izuzev 11 instrukcija) spada pod *Special3 opcode*, a razlikuju se po *op* (biti 10..6) i *function* (biti 5..0) poljima instrukcijske reči. Jedanaest instrukcija koje ne spadaju pod *Special3 opcode* su zajedničke za osnovni i DSP instrukcijski skup. Od njih jedanaest, šest instrukcija ima *Special opcode* (`MFHI`, `MFLO`, `MTHI`, `MTLO`, `MULT`, `MULTU`), četiri *Special2* (`MADD`, `MADDU`, `MSUB`, `MSUBU`), i jedna *Regimm* (`BPOSGE32`).

Da bi se u okruženje Velgrind dodala podrška za DSP instrukcijski skup, bilo je potrebno obaviti izmene u postojećem disasembleru (*guest* deo okruženja Velgrind), koji svaku instrukciju pretvara u jednu ili više među-kod instrukcija koje u potpunosti ažuriraju stanje klijentskog programa u memoriji. Zbog činjenice da sve instrukcije iz DSP ASE skupa spadaju pod već

postojeći *opcode* osnovnog instrukcijskog skupa MIPS32 arhitekture, nije bilo potrebe da se piše potpuno novi disasembler.

Pored toga, bilo je potrebno izmeniti i *host* deo okruženja, koji se odnosi na realizacije Velgrindovih IR operacija specifične za ciljnu platformu.

# 5. Realizacija

Ovo poglavlje opisuje korake koje je bilo potrebno ispratiti kako bi se postiglo proširenje alata Velgrind podrškom za drugu reviziju MIPS vektorskog instrukcijskog seta namenjenog digitalnoj obradi signala.

Bilo je potrebno prilagoditi jezgro Velgrinda novom instrukcijskom skupu, kao i realizovati spregu između Velgrindove interne reprezentacije i klijentskih programa koji upotrebljavaju DSP instrukcijski skup.

## 5.1 Prilagođenje okruženja

Kako bi se dodala podrška za instrukcijski skup MIPS DSP ASE u okruženje Velgrind, bilo je potrebno napraviti neke izmene u postojećem kodu, kako u *host* delu Velgrinda, koji povezuje Velgrind sa ciljnom arhitekturom, tako i u delu koji spaja klijentski kod sa internom reprezentacijom Velgrinda (*guest* deo). Ovo je opisano u tri celine:

- Registarska proširenja
- Proširenja programske podrške jezgra Velgrinda za određivanje mogućnosti fizičke arhitekture
- Izmene u *host* delu Velgrinda

### 5.1.1 Registarska proširenja

Pre svega, skupu već definisanih registara arhitekture MIPS trebalo je dodati kontrolni registar i akumulatore. Podatke o registrima arhitekture MIPS sadrži struktura `VexGuestMIPS32State` iz datoteke `libvex_guest_mips32.h`, koja je bila proširena jednim 32-bitnim registrom (`DSPControl`) i dodatnim 64-bitnim akumulatorima (`ac0`-`ac3`). Uz to, dodate su i pomoćne funkcije koje dobavljaju vrednosti iz tih registara i funkcije koje služe za upis vrednosti u njih. Pošto je akumulator `ac0` u osnovnom instrukcijskom skupu arhitekture MIPS

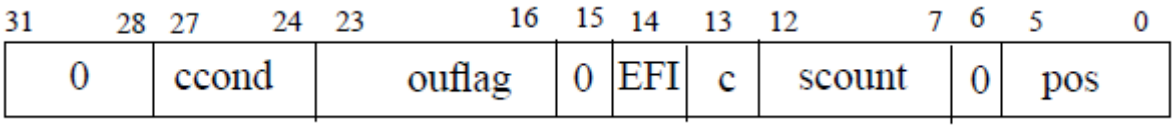
predstavljen kao dva 32-bitna registra, `HI` i `LO`, postojeće funkcije za upis vrednosti u `HI` i `LO` registre su proširene upisom vrednosti na odgovarajuću poziciju akumulatora ac0, a funkcija za upis u akumulatore napisana je tako da obezbeđuje kompatibilnost između osnovnog instrukcijskog skupa i DSP ASE. Nakon ovoga, upis u `ac0` automatski znači i upis u `HI`/`LO` par registara, i obrnuto.

Za upis u kontrolni registar, `DSPControl`, napisana je jednostavna funkcija koja kao ulazni parametar prima vrednost koju registar treba da ima nakon upisa.

Registar `DSPControl` je u realnosti podeljen na 9 polja koja služe za čuvanje statusnih bita potrebnih za neke DSP instrukcije, kao što je prikazano na Sl. 2. Polja čuvaju sledeće podatke:

- Biti 31..28: rezervisani za MIPS64 arhitekturu, na MIPS32 obavezno imaju vrednost 0.
- Biti 27..24: polje *ccond*, sadrži uslovne bite koje postavljaju instrukcije poređenja.
- Biti 23..16: polje *outflag*, označava da se dogodilo prekoračenje opsega kod nekih instrukcija koje vrše aritmetičke operacije.
- Bit 15: obavezno 0.
- Bit 14: polje *EFI (Extract Fail Indicator)*, označava neuspešno izvršenu operaciju izvlačenja bita.
- Bit 13: polje *c*, označava bit prenosa za određene instrukcije koje vrše sabiranje.
- Biti 12..7: polje *scount*, koristi se u instrukciji `INSV` za određivanje broja bita koji treba da se umetnu u registar.
- Bit 6: uvek 0.
- Biti 5..0: polje *pos*, označava poziciju bita na koju se umeće niz bita, ili poziciju sa koje se biti izvlače u instrukcijama poput `INSV`, `EXTP`, `EXTPDP`, …

Pomoću instrukcija `RDDSP` i `WRDSP` ova polja se mogu nezavisno jedna od drugih čitati ili popunjavati. Prvobitno je pomoćna funkcija za upis u `DSPControl` simuliranog procesora u okviru Velgrinda bila realizovana tako da prima samo vrednost tog polja koje se želi izmeniti pa je unutar funkcije ta vrednost upisivana primenom logičke operacije ILI između ulazne vrednosti i trenutne vrednosti `DSPControl` registra. U takvoj realizaciji, javljao se problem ukoliko neko polje ima vrednost različitu od nule, a potrebno je postaviti sve njegove bite na 0. Zato je takva realizacija zamenjena trenutnom pa je pre upisa potrebno pročitati trenutnu vrednost `DSPControl` registra, postaviti željene bite i onda tu vrednost proslediti funkciji za upis.




Sl. 2 - Polja kontrolnog registra DSP jedinice

Izmene koje je bilo potrebno izvršiti kako bi se podržali novi registri, prikazane su delovima koda koji slede.

datoteka: guest_mips_toIR.c

```
static void putHI(IRExpr * e)
{
   if (mode64) {
      stmt(IRStmt_Put(offsetof(VexGuestMIPS64State, guest_HI), e));
   } else {
      stmt(IRStmt_Put(offsetof(VexGuestMIPS32State, guest_HI), e));
   /* Add value to higher 32 bits of ac0 to maintain compatibility between
      regular MIPS32 instruction set and MIPS DSP ASE. Keep lower 32bits
      unchanged. */
      IRTemp t_lo = newTemp(Ity_I32);
      IRTemp t_hi = newTemp(Ity_I32);
      assign(t_hi, e);
      assign(t_lo, unop(Iop_64to32, getAcc(0)));
      stmt(IRStmt_Put(accumulatorGuestRegOffset(0),
           binop(Iop_32HLto64, mkexpr(t_hi), mkexpr(t_lo))));
   }
}
```

datoteka: guest_mips_toIR.c

```
static void putLO(IRExpr * e)
{
   if (mode64) {
      stmt(IRStmt_Put(offsetof(VexGuestMIPS64State, guest_LO), e));
   } else {
      stmt(IRStmt_Put(offsetof(VexGuestMIPS32State, guest_LO), e));
   /* Add value to lower 32 bits of ac0 to maintain compatibility between
      regular MIPS32 instruction set and MIPS DSP ASE. Keep higher 32bits
      unchanged. */
      IRTemp t_lo = newTemp(Ity_I32);
      IRTemp t_hi = newTemp(Ity_I32);
      assign(t_lo, e);
      assign(t_hi, unop(Iop_64HIto32, getAcc(0)));
      stmt(IRStmt_Put(accumulatorGuestRegOffset(0),
           binop(Iop_32HLto64, mkexpr(t_hi), mkexpr(t_lo))));
   }
}
```

datoteka: guest_mips_toIR.c

```
/* Put value to accumulator (helper function for MIPS32 DSP ASE instructions). */
static void putAcc(UInt acNo, IRExpr * e)
{
   vassert(!mode64);
   vassert(acNo <= 3);
   vassert(typeOfIRExpr(irsb->tyenv, e) == Ity_I64);
   stmt(IRStmt_Put(accumulatorGuestRegOffset(acNo), e));
/* If acNo == 0, split value to HI and LO regs in order to maintain compatibility
   between MIPS32 and MIPS DSP ASE instruction sets. */
   if (0 == acNo) {
     putLO(unop(Iop_64to32, e));
     putHI(unop(Iop_64HIto32, e));
   }
}
```

### 5.1.2 Proširenja programske podrške jezgra Velgrinda za određivanje mogućnosti fizičke arhitekture

Kao što je već pomenuto, disasembler napisan za instrukcije iz osnovnog instrukcijskog skupa arhitekture MIPS bio je pogodan i za instrukcije iz DSP ASE skupa jer sve DSP instrukcije imaju neki od kodova operacije osnovnog instrukcijskog skupa (većina ima *Special3*, 6 *Special*, 4 instrukcije *Special2* i jedna *Regimm opcode*). Jedina izmena je bila izdvajanje koda

za modelovanje instrukcija iz DSP ASE skupa u zasebnu funkciju radi bolje organizacije koda celog jezgra. Da bi se pozvala funkcija u kojoj se disasembliraju instrukcije iz skupa DSP ASE, proverava se da li platforma na kojoj se kod analizira poseduje podršku za potreban instrukcijski skup (DSPr1 ili DSPr2). Za to služi polje `hwcaps` u strukturi `archinfo`, koja čuva podatke o platformi na kojoj se kod analizira i njenim specifičnostima.

Kako bi se odredile dodatne mogućnosti ciljne platforme, konkretno podrška za DSPr1 i DSPr2, bilo je potrebno napisati realizacije funkcija `setjmp` i `longjmp` specifične za arhitekturu MIPS. Datoteka `m_libcsetjmp.c` sadrži specifične realizacije ovih funkcija i za neke druge arhitekture koje podržava Velgrind okruženje, pa je ona i izmenjena dodavanjem njihovih realizacija za MIPS. Ove funkcije služe za čuvanje i vraćanje stanja procesora sa steka prilikom skokova i poziva funkcija i realizovane su u asemblerskom jeziku arhitekture MIPS, a korišćene su prilikom provere arhitekture i njenih specifičnosti. Realizacije funkcija `setjmp` i `longjmp` predstavlja blok programskog koda u nastavku.

datoteka: m_libcsetjmp.c

```
#if defined(VGP_mips32_linux)

__asm__(
".text                          \n\t"
".globl VG_MINIMAL_SETJMP;      \n\t"
".align 2;                      \n\t"
"VG_MINIMAL_SETJMP:             \n\t"  /* a0 = jmp_buf */
"   sw   $s0,  0($a0)           \n\t"  /* Save registers s0-s7. */
"   sw   $s1,  4($a0)           \n\t"
"   sw   $s2,  8($a0)           \n\t"
"   sw   $s3, 12($a0)           \n\t"
"   sw   $s4, 16($a0)           \n\t"
"   sw   $s5, 20($a0)           \n\t"
"   sw   $s6, 24($a0)           \n\t"
"   sw   $s7, 28($a0)           \n\t"
"   sw   $s8, 32($a0)           \n\t"  /* Frame pointer. */
"   sw   $ra, 36($a0)           \n\t"  /* Return address. */
"   sw   $gp, 40($a0)           \n\t"  /* Global data pointer. */
"   sw   $sp, 44($a0)           \n\t"  /* Stack pointer. */

"   move $v0, $zero             \n\t"  /* Return zero. */
"   j    $ra                    \n\t"
"   nop                         \n\t"
".previous                      \n\t"
"                               \n\t"
".globl VG MINIMAL LONGJMP;     \n\t"
".align 2;                      \n\t"
"VG_MINIMAL_LONGJMP:            \n\t"  /* a0 = jmp_buf */
"   lw   $s0,  0($a0)           \n\t"  /* Restore registers s0-s7. */
"   lw   $s1,  4($a0)           \n\t"
"   lw   $s2,  8($a0)           \n\t"
"   lw   $s3, 12($a0)           \n\t"
"   lw   $s4, 16($a0)           \n\t"
"   lw   $s5, 20($a0)           \n\t"
"   lw   $s6, 24($a0)           \n\t"
"   lw   $s7, 28($a0)           \n\t"
"   lw   $s8, 32($a0)           \n\t"  /* Frame pointer. */
"   lw   $ra, 36($a0)           \n\t"  /* Return address. */
"   lw   $gp, 40($a0)           \n\t"  /* Global data pointer. */
"   lw   $sp, 44($a0)           \n\t"  /* Stack pointer. */

/* Checking whether second argument is zero. */
"   bnez $a1, 1f                \n\t"
"   nop                         \n\t"
"   addi $a1, $a1, 1            \n\t"  /* We must return 1 if val=0. */
"1:                             \n\t"
"   move $v0, $a1               \n\t"  /* Return value of second argument. */
"   j    $ra                    \n\t"
"   nop                         \n\t"
".previous                      \n\t"
);
#endif  /* VGP mips32 linux */
```

Provera arhitekture na kojoj se kod izvršava i njenih mogućnosti vrši se u datoteci `m_machine.c`, za svaku od podržanih arhitektura. Provera MIPS arhitekture je već postojala pa je samo proširena proverom podrške za DSPr1 i DSPr2. Ideja algoritma za proveru je da se pokuša izvršavanje po jedne instrukcije za DSPr1 i DSPr2 i pravilno obrade signali, ukoliko se jave. Ako je instrukcija uspešno izvršena, platforma ima podršku za taj instrukcijski skup i određeni bit promenljive `hwcaps` se postavlja na 1. U slučaju da se javio signal ( `Illegal instruction - SIGILL`), on se hvata i obrađuje na poseban način, kako se program ne bi završio uz poruku o grešci, i određeni bit promenljive `hwcaps` se postavlja na 0.

### 5.1.3 Izmene u sprezi između IR i ciljne arhitekture

Kao što je već napomenuto, Velgrind nudi veliki broj primitivnih operacija u okviru svoje interne reprezentacije. Ove operacije služe za prevođenje instrukcija klijentskog koda u IR kod, nezavistan od bilo koje arhitekture, u prvom koraku translacije. IR instrukcije moraju u potpunosti da ažuriraju stanje klijentskog programa u memoriji pa se zato prvo dobavlja trenutno klijentsko stanje iz memorije u virtualne registre, obavljaju se željene operacije nad njima, a zatim se rezultati snimaju u klijentsku memoriju.

Svaka od ovih operacija je deklarisana u datoteci `libvex_ir.c`, koja je jedinstvena za ceo Velgrind, a za svaku podržanu arhitekturu postoji zasebna datoteka sa njihovim realizacijama, specifičnim za datu arhitekturu. Realizacije operacija predstavljaju vezu između Velgrindovih IR operacija i instrukcija ciljne arhitekture u koje se među-kod prevodi u poslednjem koraku translacije. Datoteke u kojima se taj kod nalazi imaju naziv `host_<arch_name>_isel.c`.

Iako je skup IR operacija zamišljen kao skup sa RISC osobinama, na samom začetku projekta Velgrind fokus je bio na instrukcijskim skupovima arhitekture Intel x86. Posledica toga je da neke operacije, iako zamišljene kao primitivne, ne mogu biti preslikane 1:1 na instrukcije instrukcijskog skupa arhitekture MIPS, već se moraju realizovati pomoću nekoliko ovakvih instrukcija. Na primer, operaciju `Iop_Sub32`, za oduzimanje dve 32-bitne vrednosti, moguće je realizovati samo instrukcijom `subu` osnovnog instrukcijskog skupa arhitekture MIPS, ali operaciju `Iop_16Uto32`, koja neoznačenu 16-bitnu vrednost proširuje do 32 bita, nije. Ova operacija je morala biti realizovana pomoću dve instrukcije pomeranja `sll` i `srl`.

Veliki broj potrebnih operacija je već bio realizovan prilikom proširenja Velgrind okruženja podrškom za osnovni instrukcijski skup arhitekture MIPS, dok je jedan broj operacija realizovan u okviru podrške za MIPS DSP ASE. Najveći deo dodatog koda se odnosi na operacije koje rukuju 64-bitnim podacima, a koje su bile potrebne prilikom modelovanja instrukcija koje rade sa akumulatorima.

I pored toga što je skup IR operacija projektovan tako da sadrži i primitivne SIMD operacije, zbog specifičnosti velikog broja MIPS DSP ASE vektorskih instrukcija, svega nekoliko takvih operacija je korišćeno u okviru modelovanja MIPS DSP instrukcija. Naime, veliki broj vektorskih instrukcija DSP ASE skupa, u zavisnosti od rezultata, postavlja neke od statusnih bita `DSPControl` registra, što onemogućava upotrebu čak i onih operacija koje funkcionalno odgovaraju pojedinim SIMD DSP instrukcijama jer sadržaj `DSPControl` registra simuliranog procesora ne može biti adekvatno ažuriran. Na primer, operacija `Iop_QAdd16Ux2` za vektorsko sabiranje neoznačenih 16-bitnih podataka, mogla bi se direktno preslikati na vektorsku instrukciju `addq.ph`, ali je to onemogućeno zbog toga što instrukcija `addq.ph` menja sadržaj kontrolnog registra ukoliko dođe do prekoračenja opsega prilikom sabiranja bilo kog od dva para

vrednosti u ulaznim registrima. U tom slučaju se sadržaj `DSPControl` registra klijentskog programa ne bi mogao pravilno ažurirati.

U okviru proširenja Velgrind okruženja podrškom za MIPS DSP ASE instrukcijski skup, dodate su, između ostalog, i realizacije sledećih operacija:

1. `Iop_QAdd16Ux2`
2. `Iop_HSub8Ux4`
3. `Iop_HSub16Sx2`
4. `Iop_QSub8Ux4`
5. `Iop_CmpNEZ16x2`
6. `Iop_CmpNEZ8x4`
7. `Iop_Shr64`
8. `Iop_Shl64`
9. `Iop_Sar64`
10. `Iop_Left64`

## 5.2 Modelovanje instrukcija iz skupa MIPS DSP ASE

Proces modelovanja instrukcija se odnosi na prvi korak translacije – disasembliranje. Model neke instrukcije je programski kod kojim se data instrukcija pravilno raščlanjuje na jednu ili više IR operacija.

Modelovanje instrukcija iz skupa MIPS DSP ASE je od originalnog disasemblera izdvojeno u zasebnu funkciju u cilju očuvanja preglednosti koda. U disasembleru su podržani slučajevi *opcode* i *function* kodova za DSP (r1 i r2) instrukcije ali se, ukoliko se na njih naiđe, prvo proverava da li platforma ima podršku za DSP (r1 ili r2). Ovo se utvrđuje proveravanjem određenih bita promenljive `hwcaps`. Ukoliko je DSP jedinica prisutna, poziva se funkcija u kojoj su modelovane odgovarajuće DSP instukcije, u suprotnom, skače se na kraj i ispisuje se poruka o grešci.

U osnovnom instrukcijskom skupu arhitekture MIPS postoje neke instrukcije, poput `RDHWR`, koje nije moguće predstaviti pomoću IR instrukcija. Instrukcija `RDHWR` služi za dobavljanje vrednosti iz hardverskih registara i, u nekim slučajevima, zahteva privilegovan režim rada. Takve instrukcije nije moguće predstaviti pomoću IR instrukcija i za njih su definisani blokovi asemblerskih instrukcija koji ih direktno izvršavaju. U slučaju kada neka od ovih instrukcija snima rezultat u registre ili memoriju procesora MIPS, isto se radi i sa klijentskim registrima ili memorijom simuliranog procesora MIPS u samom Velgrindu.

Slično instrukciji `RDHWR`, u DSP ASE instrukcijskom skupu postoje instrukcije `RDDSP` i `WRDSP` koje služe za čitanje i upis u kontrolni registar DSP jedinice. Kako za pristup kontrolnom

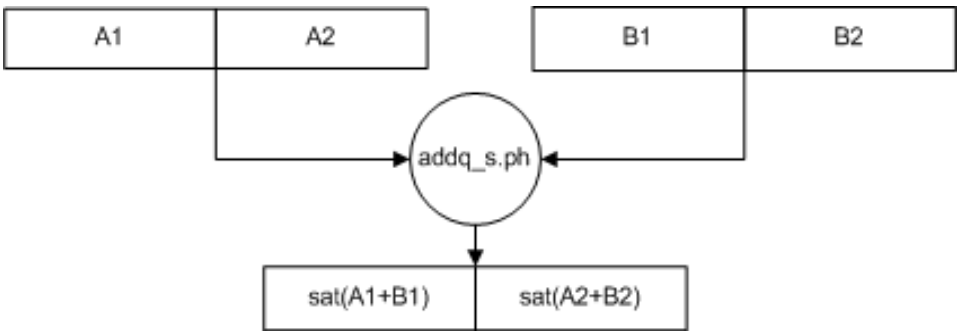
registru (`DSPControl`) nisu potrebna posebna prava, tj. mogu mu slobodno pristupati i programi iz korisničkog prostora, ovaj registar je simuliran zajedno sa opštim registrima arhitekture MIPS u okviru strukture `VexGuestMIPS32State` pa nije bilo potrebno pisati zasebne asemblerske blokove, kao u slučaju `RDHWR`.

U nastavku je deo osnovnog disasemblera u koji je dodat poziv funkcije za disasembliranje instrukcija iz skupa DSP ASE, za slučaj nekoliko *function* kodova iz *Special3 opcode*-a.

```
/*----          Disassemble a single instruction             ----*/
static DisResult disInstr_MIPS_WRK ( Bool(*resteerOkFn) (/*opaque */void *, Addr64),
                                     Bool         resteerCisOk, void* callback_opaque,
                                     Long         delta64, VexArchInfo* archinfo,
                                     VexAbiInfo*  abiinfo, Bool         sigill_diag )
{
  ...
  UInt opcode, cins, function, sa;
  cins=getUInt(code); opcode=get_opcode(cins); function=get_function(cins); sa=get_sa(cins);
  switch (opcode) {
   …
   case 0x1F: {  /* Special3 */
      switch (function) {
      /* -------- MIPS32(r2) DSP ASE(r2) instructions -------- */
         case 0xA:  /* LX */
         case 0xC:  /* INSV */
         case 0x38: {  /* EXTR.W */
            if ((archinfo->hwcaps & VEX_MIPS_ASE_DSP)) {
              UInt retVal = disDSPInstr_MIPS_WRK ( cins );
              if (0 != retVal ) {
                  goto decode_failure_dsp;
               }
               break;
            } else {
               goto decode_failure_dsp;
            }
            break;
         }
         case 0x10: {  /* ADDU.QB */
            switch(sa) {
               case  0xC:  /* SUBU_S.PH */
               case  0xD:  /* ADDU_S.PH */
               case 0x1E: {  /* MULQ_S.PH */
                  if ((archinfo->hwcaps & VEX_MIPS_ASE_DSP2P)) {
                     UInt retVal = disDSPInstr_MIPS_WRK ( cins );
                     if (0 != retVal ) {
                        goto decode_failure_dsp;
                     }
                     break;
                  } else {
                     goto decode_failure_dsp;
                  }
                  break;
               }
               default: {
                  if ((archinfo->hwcaps & VEX_MIPS_ASE_DSP)) {
                     UInt retVal = disDSPInstr_MIPS_WRK ( cins );
                     if (0 != retVal ) {
                        goto decode_failure_dsp;
                     }
                     break;
                  } else {
                     goto decode_failure_dsp;
                  }
                  break;
               }
            }
            break;
         }
         default:
            goto decode_failure;
      }
      break;  /* Special3 */
   }
    ...
}
…
decode_failure_dsp:
   vex_printf("Error occured while trying to decode MIPS32 DSP "
              "instruction.\nYour platform probably doesn't support "
              "MIPS32 DSP ASE.\n");
decode_failure:
/* All decode failures end up here. */
…
}
```

### 5.2.1 Primer modelovanja jedne instrukcije iz skupa DSP ASE

U nastavku će biti prikazana jedna instrukcija iz skupa, kao i njen model u okviru Velgrinda. Za prikaz je izabrana instrukcija `ADDQ_S.PH` koja vektorski sabira dva para označenih 16-bitnih vrednosti, saturišući rezultat ukoliko je potrebno. Na Sl. 3, prikazan je dijagram operacije koju vrši ova instrukcija. Saturacija podrazumeva postavljanje rezultata na maksimalnu ili minimalnu vrednost u slučaju prekoračenja opsega prilikom njegovog generisanja. U ovom slučaju to znači da će krajnji rezultat biti postavljen na vrednost `0x7fff` (maksimalna vrednost za 16 bita), u slučaju da je vrednost nakon sabiranja veća od te, ili na `0x8000` (minimalna vrednost za 16 bita) ukoliko je rezultat sabiranja manji od nje.




Sl. 3 - Dijagram instrukcije addq_s.ph

Pseudo-kod instrukcije `ADDQ_S.PH`.

```
ADDQ_S.PH:
tempB15..0 ←. satAdd16( GPR[rs]31..16 , GPR[rt]31..16 )
tempA15..0 ←. satAdd16( GPR[rs]15..0 , GPR[rt]15..0 )
GPR[rd]31..0 ←. tempB15..0 || tempA15..0

function satAdd16( a15..0, b15..0 )
    temp16..0 ← ( a15 || a15..0 ) + ( b15 || b15..0 )
    if ( temp16 ≠ temp15 ) then
        if ( temp16 = 0 ) then
            temp15..0 ← 0x7FFF
        else
            temp15..0 ← 0x8000
        endif
        DSPControlouflag:20 ← 1
    endif
    return temp15..0
endfunction satAdd16
```

Prilikom modelovanja instrukcije `addq_s.ph`, korišćene su operacije `Iop_Add32` (sabiranje 32-bitnih vrednosti), `Iop_And32` (operacija logičko I nad 32-bitnim vrednostima), `Iop_CmpNE32` (poređenje dve 32-bitne vrednosti u odnosu "nije jednako"), `Iop_Shr32` (32-bitno logičko pomeranje u desno), `Iop_16HLto32` (spajanje dve 16-bitne vrednosti u jednu 32-bitnu) i `IRExpr_ITE` (ternarni operator if-then-else).

U primeru su korišćene operacije koje rade nad 32-bitnim vrednostima, iako se radi o sabiranju 16-bitnih vrednosti, kako bi moglo da se detektuje prekoračenje i, po potrebi, izvrši saturacija. Ovo je urađeno poređenjem 15. i 16. bita rezultata. Ukoliko su biti različiti, i 16. bit je 0, rezultat se postavlja na maksimalnu 16-bitnu vrednost (`0x7fff`). Ukoliko su biti različiti, i 16. bit je 1, rezultat se postavlja na minimalnu 16-bitnu vrednost (`0x8000`).

```
case 0xE: {  /* ADDQ_S.PH */
   DIP("addq_s.ph r%d r%d, r%d", rd, rs, rt);
   vassert(!mode64);
   t0 = newTemp(Ity_I32);
   t1 = newTemp(Ity_I1);
   t2 = newTemp(Ity_I32);
   t3 = newTemp(Ity_I1);
   t4 = newTemp(Ity_I16);
   t5 = newTemp(Ity_I16);
   t6 = newTemp(Ity_I32);
   t7 = newTemp(Ity_I32);

   /* Add lower halves. */
   assign(t0, binop(Iop_Add32,
                    unop(Iop_16Sto32, unop(Iop_32to16, getIReg(rs))),
                    unop(Iop_16Sto32, unop(Iop_32to16, getIReg(rt)))));

   /* Bit 16 of the result. */
   assign(t6, binop(Iop_And32,
                    unop(Iop_16Uto32, unop(Iop_32HIto16, mkexpr(t0))),
                    mkU32(0x1)));
   /* Detect overflow. */
   assign(t1, binop(Iop_CmpNE32,
                    binop(Iop_Shr32,
                          binop(Iop_And32, mkexpr(t0), mkU32(0x8000)),
                          mkU8(15)),
                    mkexpr(t6)));

   putDSPControl(IRExpr_ITE(mkexpr(t1),
                            binop(Iop_Or32, getDSPControl(), mkU32(0x00100000)),
                            getDSPControl()));
   /* Saturate if needed. */
   assign(t4, IRExpr_ITE(mkexpr(t1),
                         IRExpr_ITE(binop(Iop_CmpEQ32, mkexpr(t6), mkU32(0x0)),
                                    mkU16(0x7fff),
                                    mkU16(0x8000)),
                         unop(Iop_32to16, mkexpr(t0))));

   /* Add higher halves. */
   assign(t2, binop(Iop_Add32,
                    unop(Iop_16Sto32, unop(Iop_32HIto16, getIReg(rs))),
                    unop(Iop_16Sto32, unop(Iop_32HIto16, getIReg(rt)))));

   /* Bit 16 of the result. */
   assign(t7, binop(Iop_And32,
                    unop(Iop_16Uto32, unop(Iop_32HIto16, mkexpr(t2))),
                    mkU32(0x1)));
   /* Detect overflow. */
   assign(t3, binop(Iop_CmpNE32,
                    binop(Iop_Shr32,
                          binop(Iop_And32, mkexpr(t2), (0x00008000)),
                          mkU8(15)),
                    mkexpr(t7)));

   putDSPControl(IRExpr_ITE(mkexpr(t3),
                            binop(Iop_Or32, getDSPControl(), mkU32(0x00100000)),
                            getDSPControl()));
   /* Saturate if needed. */
   assign(t5, IRExpr_ITE(mkexpr(t3),
                         IRExpr_ITE(binop(Iop_CmpEQ32, mkexpr(t7), mkU32(0x0)),
                                    mkU16(0x7fff),
                                    mkU16(0x8000)),
                         unop(Iop_32to16, mkexpr(t2))));

   putIReg(rd, binop(Iop_16HLto32, mkexpr(t5), mkexpr(t4)));
   break;
}
```

# 6. Translacija na primeru instrukcije iz instrukcijskog skupa MIPS DSP ASE

U ovom poglavlju će, na primeru instrukcije iz skupa MIPS DSP ASE, detaljnije biti prikazan proces translacije koju vrši Velgrind okruženje sa svojim alatima. Akcenat će biti stavljen na početak i kraj translacije, jer su ti delovi bili glavni zadatak ovog rada, uz osvrt na korake između i njihov uticaj na proces analize koda.

U nastavku će biti dat detaljniji prikaz koraka translacije koji se vrše za svaki pojedinačni blok koda.

## 6.1 Faze translacije

Svaki od osnovnih blokova Velgrind prevodi na zahtev. Translacija instrukcija jednog osnovnog bloka završava nekim od sledećih događaja: (a) dostignuta je granica od 60 instrukcija u bloku, (b) instrukcija koja se prevodi je instrukcija uslovnog grananja, (c) nailazak na grananje na nepoznatu adresu, ili (d) desila su se tri grananja na poznate adrese.

Translacija se vrši u 8 koraka i sve, osim instrumentacije koju obavlja alat, obavlja jezgro Velgrinda.

1. **Disasembliranje: mašinski kod → IR kod**

   Disasembliranje je prvi korak translacije. Ova faza zavisi od arhitekture na kojoj se Velgrind izvršava (i čiji se kod analizira) jer na osnovu modela instrukcije Velgrind instrukciju prevodi u neoptimizovani IR kod. Svaka instrukcija se nezavisno od susednih prevodi u jednu ili više instrukcija IR koda. U ovom koraku, generisani IR kod je organizovan u strukturu tipa stabla.

2. **Optimizacija IR koda: IR stablo → linearni IR**

Drugi korak je prva od dve optimizacije koje se rade tokom translacije. Tokom ovog koraka, kod koji je bio organizovan kao stablo se linearizuje. Iz generisanog IR koda se uklanja kod koji vrši nepotrebna dobavljanja/upis vrednosti u *guest* registre, kao i kod koji nema uticaj na krajnji rezultat. Pojednostavljuju se konstantni izrazi i vrši se jednostavno odmotavanje petlji koje se izvršavaju u jednom osnovnom bloku.

3. **Instrumentacija: linerni IR→ linearni IR**

   U ovom koraku, blok se prosleđuje odabranom alatu, koji ga proizvoljno može izmeniti. Kako bi instrumentacija bila što jednostavnija, važno je da kod u ovom koraku bude linearizovan. Za alate poput Memčeka, koji prilikom instrumentacije "boje" vrednosti meta-podacima, posebno je važno da kod bude linearizovan. Memček u ovom koraku vrši proveru definisanosti podataka na nivou svakog bita.

4. **Optimizacija: linerni IR→ linearni IR**

   Druga optimizacija je slična prethodnoj, ali je jednostavnija. Tokom ovog koraka se dodatno pojednostavljuju konstantni izrazi i uklanja se kod koji nema uticaj na krajnji rezultat. Takođe, uklanjaju se i nepotrebne provere definisanosti nastale u prethodnoj fazi. Na primer, za instrumentaciju operacije sabiranja neke promenljive i konstante, kod provere definisanosti konstante bi bio uklonjen jer su kod konstanti svi biti uvek definisani.

5. **Izgradnja stabla: linearni IR → IR stablo**

   U ovom koraku, linearizovani kod se vraća u strukturu stabla, kao priprema za narednu fazu, u kojoj se biraju instrukcije ciljne arhitekture.

6. **Odabir instrukcija: IR stablo → lista instrukcija**

   Faza odabira instrukcija je zavisna od arhitekture. U ovom koraku, svaka IR operacija se prevodi u jednu ili više instrukcija ciljne arhitekture. Tokom ove faze, još uvek su u upotrebi virtualni, *guest*, registri.

7. **Dodela registara: lista instrukcija → lista instrukcija**

   Virtualni registri se zamenjuju registrima ciljne arhitekture, a u slučaju da broj registara koji su na raspolaganju nije dovoljan primenjuje se tehnika "presipanja" registara na stek (engl. *register spill*). Ova tehnika podrazumeva da se svi registri čija se vrednost mora održavati kroz funkcijske pozive privremeno sačuvaju na steku. Iako su instrukcije zavisne od platforme, programski modul za dodelu registara nije i koristi specijalne funkcije da odredi koji su registri korišćeni tokom izvršavanja neke instrukcije.

8. **Asembliranje: lista instrukcija → mašinski kod**

   Poslednja faza translacije se sastoji od proste zamene instrukcija odgovarajućim mašinskim kodom i njegovim upisom u memoriju.

Za ovaj prikaz, izabrana je instrukcija `shrav.ph` kao primer nekih od tipičnih SIMD instrukcija ovog instrukcijskog skupa.

## 6.2 Instrukcija shrav.ph

Ova instrukcija služi za aritmetičko pomeranje u desno dve nezavisne 16-bitne vrednosti u vektorskom tipu podataka za promenljiv broj bita. Svaka od dve 16-bitne reči se nezavisno pomera u desno, uz popunjavanje svake od upražnjenih bit-pozicija znakom pomerene vrednosti. Vrednost za koju je potrebno izvršiti pomeranje se dobavlja sa četiri najmanje značajna bita iz registra prosleđenog instrukciji. Pseudo-kod ove instrukcije je prikazan u nastavku.

```
tempB15..0 ←.shift16RightArithmetic( GPR[rt]31..16, GPR[rs]3..0 )
tempA15..0 ←.shift16RightArithmetic( GPR[rt]15..0, GPR[rs]3..0 )
GPR[rd]31..0 ←.tempB15..0 || tempA15..0
function shift16RightArithmetic( a15..0, s3..0 )
    if ( s3..0 = 0 ) then
        temp15..0 ←.a15..0
    else
        sign ←.a15
        temp15..0 ← ( signs || a15..s )
    endif
    return temp15..0
endfunction shift16RightArithmetic
```

U primeru je korišćen kod malog asemblerskog programa koji izvršava samo željenu instrukciju i potrebne pomoćne instrukcije poput učitavanja vrednosti u registre ili akumulatore. U nastavku je primer tog testa za instrukciju `shrav.ph`.

```
# tiny.asm

.data
array1:  .word 0x80003286, 0x350076bd
.text
.globl __start
.ent __start
.set noreorder
.set noat
__start:

la $t5, array1
lw $t0, 0($t5)
lw $t1, 4($t5)
shrav.ph $t4, $t1, $t0

kraj:
    li $v0, 4001
    syscall
.end __start
```

Za modelovanje ove instrukcije, korišćene su operacije `Iop_And32`, `Iop_CmpEQ32`, `Iop_Sar32`, `Iop_16Sto32`, `Iop_32HIto16`, `Iop_16HLto32` i `IRExpr_ITE`, što je prikazano sledećim kodom.

```
case 0xB: {  /* SHRAV.PH */
   DIP("shrav.ph r%d, r%d, r%d", rd, rt, rs);
   vassert(!mode64);
   t0 = newTemp(Ity_I32);
   t1 = newTemp(Ity_I1);
   t2 = newTemp(Ity_I32);
   t3 = newTemp(Ity_I32);

   assign(t0, binop(Iop_And32, getIReg(rs), mkU32(0x0f)));
   assign(t1, binop(Iop_CmpEQ32, mkexpr(t0), mkU32(0x0)));
   assign(t2, binop(Iop_Sar32,
                    unop(Iop_16Sto32, unop(Iop_32to16, getIReg(rt))),
                    unop(Iop_32to8, mkexpr(t0))));
   assign(t3, binop(Iop_Sar32,
                    unop(Iop_16Sto32, unop(Iop_32HIto16, getIReg(rt))),
                    unop(Iop_32to8, mkexpr(t0))));
   putIReg(rd, binop(Iop_16HLto32,
                     IRExpr_ITE(mkexpr(t1),
                                unop(Iop_32HIto16, getIReg(rt)),
                                unop(Iop_32to16, mkexpr(t3))),
                     IRExpr_ITE(mkexpr(t1),
                                unop(Iop_32to16, getIReg(rt)),
                                unop(Iop_32to16, mkexpr(t2)))));
   break;
}
```

1. **Disasembliranje**.

Na osnovu prethodno prikazanog modela, Velgrind instrukciju prevodi u neoptimizovani IR kod. Kako su u primeru, osim instrukcije `shrav.ph`, iskorišćene i instrukcije za učitavanje 32-bitnih vrednosti u registre (`lw`), kao i pseudo-instrukcija za učitavanje početne adrese niza u registar (`la`), u kodu koji generiše Velgrind, a koji će biti prikazan u nastavku, biće prikazane i ove instrukcije.

```
----------------------- Front end -----------------------

    0x4000B0:   0x3C0D0041  lui r13, imm: 0x41
     1:       ------ IMark(0x4000B0, 4, 0) ------
     2:       PUT(52) = 0x410000:I32
     3:       PUT(128) = 0x4000B4:I32

    0x4000B4:   0x25AD00D0  addiu r13, r13, 208
     4:       ------ IMark(0x4000B4, 4, 0) ------
     5:       PUT(52) = Add32(GET:I32(52),0xD0:I32)
     6:       PUT(128) = 0x4000B8:I32

    0x4000B8:   0x8DA80000  lw r8, 0(r13)
     7:       ------ IMark(0x4000B8, 4, 0) ------
     8:       t0 = Add32(GET:I32(52),0x0:I32)
     9:       PUT(32) = LDle:I32(t0)
    10:       PUT(128) = 0x4000BC:I32

    0x4000BC:   0x8DA90004  lw r9, 4(r13)
    11:       ------ IMark(0x4000BC, 4, 0) ------
    12:       t1 = Add32(GET:I32(52),0x4:I32)
    13:       PUT(36) = LDle:I32(t1)
    14:       PUT(128) = 0x4000C0:I32

    0x4000C0:   0x7D0962D3  shrav.ph r12, r9, r8
    15:       ------ IMark(0x4000C0, 4, 0) ------
    16:       t2 = And32(GET:I32(32),0xF:I32)
    17:       t3 = CmpEQ32(t2,0x0:I32)
    18:       t4 = Sar32(16Sto32(32to16(GET:I32(36))),32to8(t2))
    19:       t5 = Sar32(16Sto32(32HIto16(GET:I32(36))),32to8(t2))
    20:       PUT(48) =
16HLto32(ITE(t3,32HIto16(GET:I32(36)),32to16(t5)),ITE(t3,32to16(GET:I32(36)),32to16(t4)))
    21:       PUT(128) = 0x4000C4:I32

    0x4000C4:   0x24020FA1  addiu r2, r0, 4001
    22:       ------ IMark(0x4000C4, 4, 0) ------
    23:       PUT(8) = Add32(0x0:I32,0xFA1:I32)
    24:       PUT(128) = 0x4000C8:I32

    0x4000C8:   0x0000000C  syscall
    25:       ------ IMark(0x4000C8, 4, 0) ------
    26:       PUT(128) = 0x4000CC:I32
    27:       PUT(128) = GET:I32(128); exit-Sys_syscall

GuestBytes 4000B0 28  41 00 0D 3C D0 00 AD 25 00 00 A8 8D 04 00 A9 8D D3 62 09 7D A1 0F 02
```

Iz prethodnog bloka se vidi da su sve mašinske instrukcije klijentskog programa prevedene u među-kod instrukcije Velgrindove interne reprezentacije. Zbog specifičnosti testnog klijentskog koda, ceo program je stao u jedan osnovni blok i završava se sistemskim pozivom za završetak programa. U realnijem slučaju, klijentski kod bi bio podeljen u više blokova, a svaki blok bi se završio skokom na adresu početka sledećeg bloka kog treba obraditi.

Instrukcija `la`, korišćena u klijentskom kodu, zapravo, je pseudo-instrukcija i tokom translacije je predstavljena dvema instrukcijama.

Radi lakše analize i referenciranja u tekstu, u generisanom kodu je, ispred svakog iskaza, dodat broj tog iskaza u datom bloku. Sedam instrukcija iz originalnog koda (računajući dve za pseudo-instrukciju `la`) prevedeno je u 27 IR iskaza formiranih u strukturu stabla.

- Iskazi 1, 4, 7, 11, 15, 22 i 25 su `IMark` iskazi. Oni ne predstvaljaju pravu operaciju već sadrže podatke o adresi na kojoj se nalazi instrukcija i njenoj dužini i označavaju okvir jedne instrukcije u listi iskaza.

- `PUT(n)` upisuje vrednost u *guest* registar sa odstojanjem `n`. Odstojanje koje se prosleđuje kao argument se odnosi na odstojanje koji neki registar ima u strukturi `VexGuestMIPS32State`. Odstojanje 52 se odnosi na registar `$13`, tj. `$t5`, a odstojanja 32 i 36 na registre `$8` (`$t0`) i `$9` (`$t1`), respektivno.
- Iskazi poput 8, 12, 16-19 privremenoj promenljivoj dodeljuju stablo IR izraza. Tu se može videti kako se jedna instrukcija razlaže na više IR izraza. `GET:I32` služi za dobavljanje celobrojne 32-bitne vrednosti iz virtualnih, *guest*, registara.
- `LDle:I32` je operacija za dobavljanje celobrojne, 32-bitne vrednosti iz memorije. Sufiks ‘le’ u nazivu označava da se operacija odnosi na little endian organizaciju memorije.

Nakon disasembliranja prve 4 instrukcije, u registru `$t5` se nalazi adresa niza iz kog se čitaju podaci, promenljive t0 i t1 sadrže adrese elemenata koje treba pročitati iz niza, a *guest* registri `$t0` i `$t1` njihove vrednosti.

Uzimajući prethodno objašnjeno u obzir, primećuje se da je instrukcija shrav.ph disasemblirna tačno onako kako je opisano prethodno prikazanim modelom.

Još je bitno naglasiti da se disasembliranje svake instrukcije završava ažuriranjem programskog brojača (PC), što je postignuto operacijom `PUT(128)`, kojom se u registar PC stavlja adresa naredne isntrukcije.

2. **Optimizacija**

Nako disasembliranja, obavlja se prva, od dve, optimizacije. Stablo IR se linearizuje i uklanjaju se nepotrebni delovi koda.

- Učitavanje adrese niza je uprošćeno i svedeno na smeštanje konstantne vrednosti u registar, izbačene su nepotrebne `GET`, `Add32` i `PUT` operacije.
- Za adresiranje elemenata niza se koriste konstante, uklonjene su viška `GET` i `Add32` operacije.
- Kompleksno tablo IR je linearizovano i pomoćnim promenljivama se dodeljuje po jedan IR izraz iz prethodno generisanog stabla. Za instrukciju shrav.ph, kompleksni iskazi 16-20, linearizovani su pomoću 16 dodela jednostavnih izraza pomoćnim promenljivama.

```
------------------------ After pre-instr IR optimisation ------------------------

IRSB {
   t0:I32   t1:I32   t2:I32   t3:I1   t4:I32   t5:I32   t6:I32   t7:I32
   t8:I32   t9:I32   t10:I32   t11:I32   t12:I32   t13:I32   t14:I32   t15:I32
   t16:I32   t17:I32   t18:I16   t19:I32   t20:I8   t21:I32   t22:I32   t23:I16
   t24:I32   t25:I8   t26:I32   t27:I16   t28:I16   t29:I32   t30:I16   t31:I16
   t32:I16   t33:I32   t34:I16   t35:I32   t36:I32

   ------ IMark(0x4000B0, 4, 0) ------
   ------ IMark(0x4000B4, 4, 0) ------
   PUT(52) = 0x4100D0:I32
   PUT(128) = 0x4000B8:I32
   ------ IMark(0x4000B8, 4, 0) ------
   t10 = LDle:I32(0x4100D0:I32)
   PUT(32) = t10
   PUT(128) = 0x4000BC:I32
   ------ IMark(0x4000BC, 4, 0) ------
   t13 = LDle:I32(0x4100D4:I32)
   PUT(36) = t13
   ------ IMark(0x4000C0, 4, 0) ------
   t14 = And32(t10,0xF:I32)
   t3 = CmpEQ32(t14,0x0:I32)
   t18 = 32to16(t13)
   t17 = 16Sto32(t18)
   t20 = 32to8(t14)
   t16 = Sar32(t17,t20)
   t23 = 32HIto16(t13)
   t22 = 16Sto32(t23)
   t25 = 32to8(t14)
   t21 = Sar32(t22,t25)
   t28 = 32HIto16(t13)
   t30 = 32to16(t21)
   t27 = ITE(t3,t28,t30)
   t32 = 32to16(t13)
   t34 = 32to16(t16)
   t31 = ITE(t3,t32,t34)
   t26 = 16HLto32(t27,t31)
   PUT(48) = t26
   ------ IMark(0x4000C4, 4, 0) ------
   PUT(8) = 0xFA1:I32
   ------ IMark(0x4000C8, 4, 0) ------
   PUT(128) = 0x4000CC:I32; exit-Sys syscall
}
```

3. **Instrumentacija**

Osnovni blok koda se u ovom koraku prosleđuje odabranom alatu, koji ga prizvoljno može menjati. Zbog velike količine podataka koja se generiše prilikom instrumentacije koda, u nastavku je prikazan samo jedan njegov deo (instrumentacija instrukcije `shrav.ph`). Instrumentacija je urađena pomoću alata Memček.

Način na koji Memček vrši instrumentaciju i proverava definisanost podataka je objašnjen u [5] i ne spade u domen ovog rada. I pored toga, treba napomenuti sledeće:

- Tokom instrumentacije, dodaje se veliki broj novih operacija i rezultujući kod je mnogo veći i kompleksniji od originalnog.
- Registri koji čuvaju meta-podatke za proveru definisanosti (*shadow* registri), nalaze se u istom bloku memorije kao i *guest* registri. U datom isečku, vidi se da se na kraju instrumentacije instrukcije `shrav.ph` dve vrednosti smeštaju u registre. Jedna operacija je smeštanje rezultata instrukcije (`PUT(48) = t26`), dok je druga smeštanje rezultata operacije za proveru definisanosti (`PUT(432) = t93`).

- Svakoj operaciji nad vrednostima iz *guest* registara prethodi odgovarajuća *shadow* operacija za proveru definisanosti meta-podataka.
- Operacije za proveru definisanosti mogu biti i jednostavnije i kompleksnije od originalne operacije

```
------ IMark(0x4000C0, 4, 0) ------
   t44 = Or32(t37,0x0:I32)
   t45 = Or32(t10,t37)
   t46 = Or32(0xF:I32,0x0:I32)
   t47 = And32(t45,t46)
   t48 = And32(t44,t47)
   t49 = t48
   t43 = t49
   t14 = And32(t10,0xF:I32)
   t51 = Or32(t43,0x0:I32)
   t52 = CmpNEZ32(t51)
   t50 = t52
   t3 = CmpEQ32(t14,0x0:I32)
   t54 = 32to16(t40)
   t53 = t54
   t18 = 32to16(t13)
   t56 = 16Sto32(t53)
   t55 = t56
   t17 = 16Sto32(t18)
   t58 = 32to8(t43)
   t57 = t58
   t20 = 32to8(t14)
   t60 = Sar32(t55,t20)
   t61 = CmpNEZ8(t57)
   t62 = 1Sto32(t61)
   t63 = Or32(t60,t62)
   t64 = t63
   t59 = t64
   t16 = Sar32(t17,t20)
   t66 = 32HIto16(t40)
   t65 = t66
   t23 = 32HIto16(t13)
   t68 = 16Sto32(t65)
   t67 = t68
   t22 = 16Sto32(t23)
   t70 = 32to8(t43)
   t69 = t70
   t25 = 32to8(t14)
   t72 = Sar32(t67,t25)
   t73 = CmpNEZ8(t69)
   t74 = 1Sto32(t73)
   t75 = Or32(t72,t74)
   t76 = t75
   t71 = t76
   t21 = Sar32(t22,t25)
   t78 = 32HIto16(t40)
   t77 = t78
   t28 = 32HIto16(t13)
   t80 = 32to16(t71)
   t79 = t80
   t30 = 32to16(t21)
   t82 = ITE(t3,t77,t79)
   t83 = 1Sto16(t50)
   t84 = Or16(t82,t83)
   t81 = t84
   t27 = ITE(t3,t28,t30)
   t86 = 32to16(t40)
   t85 = t86
   t32 = 32to16(t13)
   t88 = 32to16(t59)
   t87 = t88
   t34 = 32to16(t16)
   t90 = ITE(t3,t85,t87)
   t91 = 1Sto16(t50)
   t92 = Or16(t90,t91)
   t89 = t92
   t31 = ITE(t3,t32,t34)
   t94 = 16HLto32(t81,t89)
   t93 = t94
   t26 = 16HLto32(t27,t31)
   PUT(432) = t93
   PUT(48) = t26
```

4. **Optimizacija**

U ovom koraku optimizacije, uklanja se nepotreban kod dodat prilikom instrumentacije. U primeru instrumentacije instrukcije `shrav.ph`, 69 iskaza generisanih tokom instrumentacije, smanjeno je na 46.

```
------ IMark(0x4000C0, 4, 0) ------
   t45 = Or32(t10,t39)
   t47 = And32(t45,0xF:I32)
   t48 = And32(t39,t47)
   t14 = And32(t10,0xF:I32)
   t52 = CmpNEZ32(t48)
   t3 = CmpEQ32(t14,0x0:I32)
   t54 = 32to16(t42)
   t18 = 32to16(t13)
   t56 = 16Sto32(t54)
   t17 = 16Sto32(t18)
   t58 = 32to8(t48)
   t20 = 32to8(t14)
   t60 = Sar32(t56,t20)
   t61 = CmpNEZ8(t58)
   t62 = 1Sto32(t61)
   t63 = Or32(t60,t62)
   t16 = Sar32(t17,t20)
   t66 = 32HIto16(t42)
   t23 = 32HIto16(t13)
   t68 = 16Sto32(t66)
   t22 = 16Sto32(t23)
   t70 = 32to8(t48)
   t25 = 32to8(t14)
   t72 = Sar32(t68,t25)
   t73 = CmpNEZ8(t70)
   t74 = 1Sto32(t73)
   t75 = Or32(t72,t74)
   t21 = Sar32(t22,t25)
   t78 = 32HIto16(t42)
   t28 = 32HIto16(t13)
   t80 = 32to16(t75)
   t30 = 32to16(t21)
   t82 = ITE(t3,t78,t80)
   t83 = 1Sto16(t52)
   t84 = Or16(t82,t83)
   t27 = ITE(t3,t28,t30)
   t86 = 32to16(t42)
   t32 = 32to16(t13)
   t88 = 32to16(t63)
   t34 = 32to16(t16)
   t90 = ITE(t3,t86,t88)
   t91 = 1Sto16(t52)
   t92 = Or16(t90,t91)
   t31 = ITE(t3,t32,t34)
   t94 = 16HLto32(t84,t92)
   t26 = 16HLto32(t27,t31)
   PUT(432) = t94
   PUT(48) = t26
```

5. **Izgradnja stabla**

U ovom koraku, linearni IR se vraća u strukturu stabla, kao priprema za fazu odabira instrukcija. Izrazi koji dodeljuju vrednost pomoćnoj promenljivoj se uglavnom celi ubacuju na mesto korišćenja te promenljive. Redosled operacija za dobavljanje vrednosti iz memorije se u ovom koraku može promeniti u odnosu na originalni kod, ali se nikada ne stavljaju na mesto nakon operacija za čuvanje vrednosti u memoriji.

Nakon izgradnje stabla, IR kod instrukcije `shrav.ph` izgleda ovako:

```
------ IMark(0x4000C0, 4, 0) ------
   t48 = And32(t39,And32(Or32(t10,t39),0xF:I32))
   t14 = And32(t10,0xF:I32)
   t52 = CmpNEZ32(t48)
   t3 = CmpEQ32(t14,0x0:I32)
   t20 = 32to8(t14)
   t25 = 32to8(t14)
   t63 = Or32(Sar32(16Sto32(32to16(t42)),t20),1Sto32(CmpNEZ8(32to8(t48))))
   t16 = Sar32(16Sto32(32to16(t13)),t20)
   t84 =
Or16(ITE(t3,32HIto16(t42),32to16(Or32(Sar32(16Sto32(32HIto16(t42)),t25),1Sto32(CmpNEZ8(32to8(
t48))))),1Sto16(t52))
   t27 = ITE(t3,32HIto16(t13),32to16(Sar32(16Sto32(32HIto16(t13)),t25)))
   PUT(432) = 16HLto32(t84,Or16(ITE(t3,32to16(t42),32to16(t63)),1Sto16(t52)))
   PUT(48) = 16HLto32(t27,ITE(t3,32to16(t13),32to16(t16)))
```

6. **Odabir instrukcija**

Postojeće stablo IR se pomoću modula za odabir instrukcija menja listom instrukcija, koje još uvek koriste virtualne registre. Zbog obimnosti koda, u nastavku je prikazana samo lista instrukcija za poslednju dodelu vrednosti pomoćnoj promenljivoj iz prethodnog koda `t27 = ITE(t3,32HIto16(t13),32to16(Sar32(16Sto32(32HIto16(t13)),t25))).`

```
-- t27 = ITE(t3,32HIto16(t13),32to16(Sar32(16Sto32(32HIto16(t13)),t25)))
srl %vr141,%vr13,16
sll %vr140,%vr141,16
sra %vr140,%vr140,16
srav %vr139,%vr140,%vr25
srl %vr142,%vr13,16
li %vr144,0x0000000000000000
subu %vr143,%vr144,%vr3
and %vr146,%vr143,%vr142
nor %vr148,%vr143,%vr143
and %vr147,%vr148,%vr139
addu %vr145,%vr146,%vr147
or %vr27,%vr145,%vr145
```

**7. Dodela registara**

U ovom koraku, virtualni registri se zamenjuju registrima ciljne arhitekture. Takođe, alokator registara, prilikom zamene virtualnih registrima može da izbaci nepotrebne razmene vrednosti između dva registra.

Za prethodni blok koda, nakon dodele registara, generiše se sledeći kod:

```
srl $19,$20,16
sll $21,$19,16
sra $21,$21,16
srav $19,$21,$16
srl $16,$20,16
li $21,0x0000000000000000
subu $13,$21,$22
and $21,$13,$16
nor $16,$13,$13
and $13,$16,$19
addu $16,$21,$13
```

8. **Asembliranje**

Poslednja faza translacije je asembliranje koje podrazumeva jednostavno kodovanje odabranih instrukcija i njihov upis u memoriju.

```
sll $24,$19,31
C0 C7 13 00

sra $24,$24,31
C3 C7 18 00

or $19,$21,$24
25 98 B8 02

srl $21,$17,16
02 AC 11 00

li $24,0x0000000000000000
00 00 18 24

subu $13,$24,$22
23 68 16 03

and $24,$13,$21
24 C0 B5 01

nor $21,$13,$13
27 A8 AD 01

and $13,$21,$19
24 68 B3 02

addu $19,$24,$13
21 98 0D 03

sll $21,$18,31
C0 AF 12 00

sra $21,$21,31
C3 AF 15 00

or $24,$19,$21
25 C0 75 02
```

# 7. Testiranje

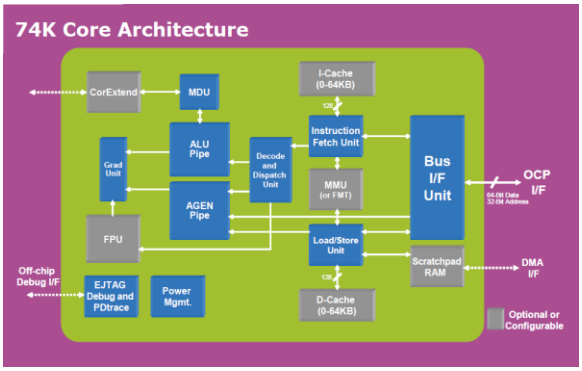
Za proveru ispravnosti programskog paketa Velgrind koji je prilagođen instrukcijskom skupu MIPS DSP ASE, korišćeni su testovi koji ispituju translaciju pojedinačnih DSP instrukcija. Pored toga, dodata funkcionalnost je testirana i dinamičkom analizom koda standardnih biblioteka otvorenog izvornog koda za audio i video obradu. Korišćene su biblioteke Pixman, WebP, WebM i zlib jer njihove realizacije koriste instrukcijski skup DSP ASE. Provera ispravnosti je izvršena na alatima Nulgrind i Memček.

## 7.1 Ciljne platforme

Kao platforme za testiranje, korišćene su MIPS Malta [6] i Sigma SMP8644 [7], obe zasnovane na procesorskoj arhitekturi 74K, kao i Sigma SMP8910 [8] platforma zasnovana na MIPS32 1004K procesorskoj arhitekturi.

- **MIPS32 74K**

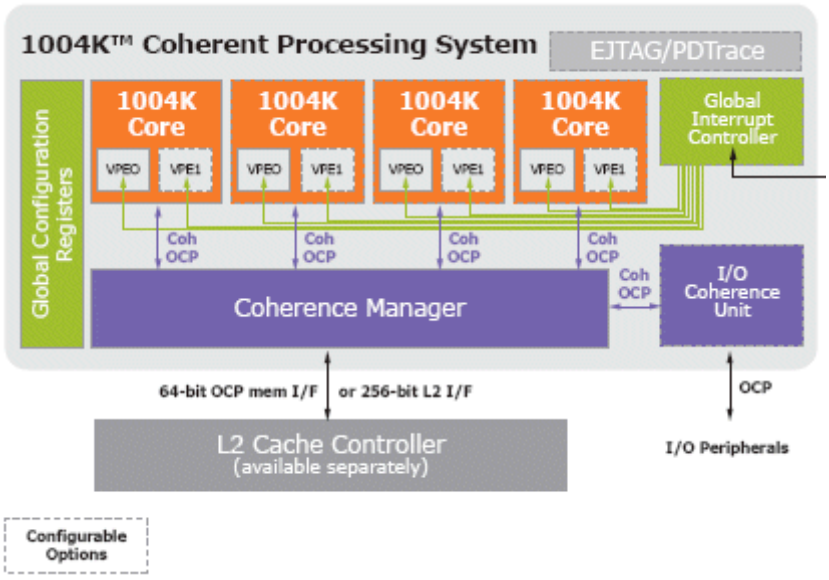
  Familija jezgara 74K je bazirana na superskalarnoj mikro-arhitekturi sa izvršavanjem instrukcija van redosleda (engl. *Out-of-order*). Arhitektura 74K poseduje 15-stepenu protočnu strukturu sa mogućnošću izvršavanja dve po dve instrukcije istovremeno i podršku za drugu reviziju instrukcijskog seta MIPS DSP ASE.




Sl. 4 - Dijagram arhitekture MIPS 74K

- **MIPS32 1004K**

  Ova arhitektura se, od ostalih sličnih sistema, izdvaja po tome što je prva arhitektura sa više jezgara (engl. *Multi-core*, MC), od kojih svako poseduje podršku za istovremeno izvršavanje više niti, što mu omogućava da prevaziđe mogućnosti višejezgarnih sistema baziranih na jednonitnim procesorima. Ovaj sistem sadrži od jednog do četiri jezgra, povezanih jedinicom zaduženom za očuvanje usklađenosti skrivenih memorija svakog od njih. Pored toga, sadrži i rukovaoca globalnim prekidima, koji prihvata do 256 prekida i prosleđuje ih jezgrima ili pak nitima unutar jezgara. Jezgra 1004K imaju 9-stepenu protočnu strukturu, koja omogućava obradu brzine i do 1.6 DMIPS/MHz po jezgru i pored celobrojne aritmetike, mogu podržavati i aritmetiku pokrednog zareza. Arhitektura MIPS32 1004K podržava prvu reviziju instrukcijskog skupa MIPS32 DSP ASE.




Sl. 5 - Dijagram arhitekture MIPS 1004K

Sva testiranja su obavljena na *big-endian* i *little-endian* verzijama platformi.

## 7.2 Rezultati

Svi alati biblioteka otvorenog izvornog koda koje su korišćene prilikom testiranja dali su bit-egzaktne rezultate. Pored provere funkcionalnosti, ispitivana je i brzina analiziranja programa sa DSP instrukcijama u ondosu na programe bez njih i dobijeni su uporedivi rezultati. Ukoliko se za analizu koristi alat Memček, ovakav kod se, u proseku, izvršava oko 20 puta sporije nego isti kod koji se izvršava bez posredstva Velgrinda, što je slučaj i sa programima koji ne koriste DSP instrukcije.

# 8. Zaključak

Prethodno opisanim prilagođenjem, omogućeno je detektovanje grešaka pomoću alata Velgrind prilikom razvoja i realizacije naprednih audio i video algoritama na arhitekturi sa podrškom za SIMD obradu. Odabirom instrukcijskog seta MIPS DSP ASE, omogućeno je otkrivanje i uklanjanje grešaka u aplikacijama na ugrađenim arhitekturama sa ograničenim resursima, kao što je MIPS, koje u poslednje vreme nude dovoljnu procesnu moć za izvršavanje i najnaprednijih audio i video algoritama u realnom vremenu.

Ispitivanje audio i video bibliotekama koje sadrže zahtevne DSP algoritme pokazalo je da je prilagođenje uspešno izvršeno, ali treba imati u vidu da ovakva analiza koda donosi usporenje od oko 20% u odnosu na DSP kod koji se izvršava samostalno, bez posredstva Velgrinda.

Dalji razvoj podrške za instrukcijski skup MIPS DSP ASE bi mogao ići u pravcu optimizacije modela DSP instrukcija i realizacija operatora Velgrind okruženja za arhitekturu MIPS u cilju generisanja što manjeg broja među-kod instrukcija i brže analize programa za digitalnu obradu signala, koji koriste DSP ASE.

# 9. Literatura

[1] Sweetman, Dominic. See MIPS Run. 2nd ed. San Francisco, Calif.: Morgan Kaufmann Publishers/Elsevier, 2007.

[2] http://www.imgtec.com/mips/mips-dsp.asp

[3] Nicholas Nethercote , Julian Seward, Valgrind: a framework for heavyweight dynamic binary instrumentation, Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation, June 10-13, 2007, San Diego, California, USA

[4] Dejan Jevtić, Petar Jovanović, Aleksandar Simeonov, Teodora Petrović: Zbornik radova 55. Konferencije za ETRAN, Banja Vrućica, 6-9. juna 2011.

[5] J. Seward and N. Nethercote. Using Valgrind to detect undefined value errors with bit-precision. In Proceedings of the USENIX'05 Annual Technical Conference, Anaheim, California, USA, April 2005.

[6] http://www.mips.com/products/development-kits/malta/

[7] http://www.sigmadesigns.com/products.php?id=33

[8] http://www.sigmadesigns.com/products.php?id=130