УНИВЕРЗИТЕТ У БЕОГРАДУ
МАТЕМАТИЧКИ ФАКУЛТЕТ

Стефана Церовина

ВИРТУЕЛНА МАШИНА ДАРТИНО-ИМПЛЕМЕНТАЦИЈА ИНТЕРПРЕТАТОРА ЗА ПЛАТФОРМУ МИПС

мастер рад

Београд, 2016.

Ментор:

др Милена Вујошевић Јаничић
Универзитет у Београду, Математички факултет

Чланови комисије:

др Саша Малков
Универзитет у Београду, Математички факултет

др Филип Марић
Универзитет у Београду, Математички факултет

Датум одбране: _____

Брату, мами и тати

Наслов мастер рада: Виртуелна машина Дартино- имплементација интерпретатора за платформу МИПС

Резиме: Интернет и напредовање технологије омогућују уграђивање сензора и механизама комуникације у све већи број свакодневних предмета. Трансформацијом огромног броја предмета у паметне уређаје, и њиховим повезивањем кроз мрежу, гради се систем који се назива IoT (скраћено од енгл. *Internet of things*). Овај термин указује на проширивање интернета у још већу мрежу која повезује све што нас окружује. Сматра се да ће IoT направити једну од највећих револуција у технологији, и да је прва фаза револуције почела. Физички свет полако постаје тип информационог система, а све то пружа могућности за развијање великог броја нових апликација, које обећавају побољшање квалитета наших живота.

Паметни уређаји представљају системе са уграђеним рачунаром, који се заснивају на микропроцесорима или микроконтролерима. Апликације за микроконтролере до сада су развијане углавном у асемблерском језику или програмском језику C, због чега је процес развоја доста спорији од развоја мобилних или веб-апликација. Дартино је пројекат компаније Гугл (енгл. *Google*) који има за циљ да омогући употребу програмског језика Дарт за програмирање микроконтролера, и тиме развој апликација приближи и олакша што већем броју програмера. Дарт је објектно-оријентисани програмски језик, иницијално развијен за писање веб-апликација, са синтаксом која је слична синтакси програмског језика C, па је лак за употребу и не захтева много учења. Дартино представља виртуелну машину за симулирање процеса, а циљ овог рада је да се омогући коришћење ове виртуелне машине на микроконтролерима са МИПС процесором, имплементацијом интерпретатора за МИПС. Мотивација за то је чињеница да су МИПС процесори трећи по заступљености на тржишту система са уграђеним рачунаром.

Кључне речи: IoT, програмски језик Дарт, виртуелна машина Дартино, МИПС

Садржај

Глава 1

Увод

Појам „Интернет ствари” (енгл. *Internet of things*), скраћено IoT, се све чешће среће, и он представља модел повезаности објеката који чине систем у оквиру којег објекти међусобно комуницирају. Реч „ствар” у оквиру појма „Интернет ствари” се односи на систем са уграђеним рачунаром (енгл. *embedded system*) који преноси и прима информације путем мреже [19]. Системи са уграђеним рачунаром су системи специјализоване намене, који обављају једну или више функција које тој намени одговарају (паметни телефони, дигитални сатови, паметне утичнице, MP3 плејери, штампачи, дигиталне камере и друго). Модерни системи са уграђеним рачунаром су углавном базирани на микроконтролерима, а ређе на микропроцесорима, зато што микроконтролере карактерише ефикасно управљање процесима у реалном времену, масовна производња, ниска цена и мала потрошња електричне енергије [21]. Сматра се да је потенцијал за развој индустрије система са уграђеним рачунаром велики, и да је будућност у изградњи велики IoT система.

Развијање апликација за системе са уграђеним рачунаром се обично своди на програмирање у асемблеру или у програмском језику C. Најчешће се користе компилатор GCC или инфраструктура LLVM коришћењем компилатора Clang, а од алата за дебаговање, обично алат GDB. Развојно окружење се најчешће налази на другом рачунару, јер микроконтролер садржи малу количину RAM и флеш меморије, па се на њему обично не може покренути ниједан регуларни оперативни систем. Због наведених услова, процес развоја апликација је доста захтеван, подложан грешкама па због тога и спор.

Пошто се веб-апликације много брже развијају од апликација за системе са уграђеним рачунаром, компанија Гугл је дошла на идеју да омогући програми-

рање система са уграђеним рачунаром у вишем програмском језику Дарт, за који постоји подршка за развој веб, серверских и мобилних апликација. Више о програмском језику Дарт речено је у глави 3.

Дарт је прилагођен специфичностима микроконтролера кроз нову библиотеку „dartino". Тако је настала виртуелна машина Дартино, са намером да се програмирање система са уграђеним рачунаром олакша и приближи што већем броју програмера. Дартино омогућава писање апликација за мале микроконтролере, на језику који доста личи на C, али је објектно-оријентисан и садржи разне погодности које процес имплементације доста олакшавају, те се апликације могу развијати брже и ефикасније. Ова виртуелна машина је детаљније описана у глави 4.

У оквиру Дартино виртуелне машине постојала је подршка за миктроконтролере са Интел и ARM архитектуром процесора. Због широке распрострањености МИПС процесора у системима са уграђеним рачунаром, у оквиру овог рада развијен је интерпретатор за платформу МИПС који је и званично интегрисан у Дартино пројекат. Имплементација интерпретатора описана је у глави 5, док је платформа МИПС детаљније описана у глави 2.

Глава 2

Архитектура МИПС

У овој глави описана је МИПС архитектура процесора. У поглављу 2.1 описане су CISC (скраћено од енгл. *Complex Instruction Set Computing*) и RISC (скраћено од енгл. *Reduced Instruction Set Computing*) архитектуре процесора, док је у поглављу 2.2 описана архитектура МИПС. Инструкције које су биле значајне у оквиру развоја интерпретатора описане су у поглављу 2.3, док су у поглављу 2.4 описани МИПС регистри. Начини адресирања су описани у поглављу 2.5. У поглављу 2.7 је описана проточна обрада на RISC и МИПС процесорима, а у поглављу 2.8 је описан појам слот закашњења при скоковима и гранањима. Структура програма у МИПС асемблерском језику описана је у поглављу 2.9, а један пример МИПС позивне конвенције дат је у поглављу 2.10.

2.1 CISC и RISC

Термин архитектура у рачунарству се користи да опише апстрактну машину која се програмира, а не стварну имплементацију те машине. Архитектура процесора у суштини дефинише скуп инструкција и регистара. Архитектура и скуп инструкција се једним именом називају ISA (скраћено од енгл. *Instruction Set Architecture*)[22].

CISC архитектуру процесора карактерише богат скуп инструкција. Главна идеја је смањивање броја инструкција по програму, при чему писање програма постаје ефикасније. Редуковањем броја инструкција по програму постиже се смањење времена извршавања. Инструкције могу бити различитих дужина, а сложеност се огледа у томе да једна инструкција може обављати више операција. На пример, инструкција може вршити учитавање вредности из меморије,

затим примену неке аритметичке операције, и записивање резултата у меморији. Овако сложене инструкције захтевају комплексност процесорског хардвера и резултујуће архитектуре. То има за последицу и теже разумевање и програмирање таквих чипова, а поред тога и већу цену. CISC процесори имају више различитих начина адресирања, од којих су неки веома комплексни. Као што је већ речено, адресе могу бити аргументи инструкције, што значи да је за разлику од RISC архитектуре, подржано индиректно адресирање. CISC процесори се углавном користе на личним рачунарима, радним станицама и серверима, а пример оваквих процесора је архитектура Интел x86 [26, 20].

RISC архитектура процесора се заснива на поједностављеном и смањеном скупу инструкција. Због једноставности инструкција, потребан је мањи број транзистора за производњу процесора, при чему процесор инструкције може брже извршавати. Међутим, редуковање скупа инструкција умањује ефикасност писања софтвера за ове процесоре. Постоје 4 начина адресирања: регистарско, PC-релативно, псеудо-директно и базно. Не постоје сложене инструкције које приступају меморији, већ се рад са меморијом своди на *load* и *store* инструкције [26, 20]. Више о начинима адресирања речено је у поглављу 2.5. Највећа предност је проточна обрада, која се лако може имплементирати, за разлику од CISC процесора код којих то није могуће. Више о проточној обради речено је у поглављу 2.7. RISC процесори се углавном користе за апликације у реалном времену. Пример RISC процесора су ARM и МИПС.

2.2 МИПС

МИПС је RISC архитектура процесора, настала средином осамдесетих година двадесетог века, као рад Џона Хенесија и његових студената на универзитету Станфорд. Истражујући RISC, показали су да се помоћу једноставног скупа инструкција, добрих компилатора и хардвера који ефикасније користи проточну обраду, могу произвести бржи процесори на мањем чипу [26].

Временом, МИПС архитектура је еволуирала на много начина. Направљена је подршка за 64-битно адресирање и операције, комплексне оперативне системе као што је Unix, као и високе перформансе при раду са бројевима у покретном зарезу. Употреба ових процесора је кроз време, у зависности од перформанси, била разнолика: радне станице, серверски системи, Сони и Нинтендо играчке конзоле, Циско рутери, ТВ сет-топ боксови, ласерски штампачи и још много

других уређаја је користило ове процесоре[26]. Приступачна цена интегрисаних кола заснованих на овој архитектури, ниска потрошња енергије и доступност алата за развој програмске подршке чине је погодном како за системе са уграђеним рачунаром, тако и за потрошачку електронику, играчке конзоле и друго.

2.3 Скуп инструкција

У МИПС асемблерском језику постоје следеће инструкције [27]:

- аритметичке: сабирање, одузимање, множење, дељење
- логичке: и, или, шифтовање у лево, шифтовање у десно
- приступ меморији: учитавање речи, записивање речи у меморију
- гранање
- скокови

На основу типова операнада, све МИПС инструкције се могу поделити у три типа [26]

R - Инструкције које као операнде очекују регистре. Представљају се следећим форматом:

OP rd, rs, rt

OP представља ознаку одређене инструкције, *rd* представља регистар за смештање резултата, док *rs, rt* означавају операнде.

Неке од инструкција које припадају R типу:

- jr - Скакање на адресу у регистру
- slt - Постави 1 у регистар за резултат уколико је вредност у првом регистру мања од вредности у другом
- srl - Логичко шифтовање у десно

I - Инструкције које имају као операнде регистар и константну вредност (енгл. *Immediate*) се представљају следећим форматом:

OP rd, rs, Imm

OP представља ознаку одређене инструкције, *rd* представља регистар за смештање резултата, *rs* представља први операнд који је регистар, док је *Imm* константа која је други операнд. Константа може имати највише 16 бита.

Неке од инструкција које припадају I типу:

- lw - Учитавање садржаја са адресе rs + Imm у регистар за резултат
- sw - Уписивање садржаја регистра rd на адресу rs + Imm
- beq - Гранање у случају да је вредност у првом регистру једнака константи
- bne - Гранање у случају да је вредност у првом регистру различита од вредности константе

J - Инструкције које се користе при скоковима. Представљају се следећим форматом:

j label

Постоје две „j" инструкције: *j* и *jal*. Инструкцијом *j* - *Jump* процесор се пребаца на извршавање инструкције која се налази на адреси *label*. Инструкција *jal* - *Jump and link* ради исто то, али смешта адресу наредне инструкције у регистар *$ra*, тако да се након завршетка функције на коју се скочило, може наставити са извршавањем. Ове инструкције имају највише места за константу-26 бита, јер су адресе велики бројеви.

Поред инструкција, постоје и псеудоинструкције, које нису праве инструкције, већ су уведене како би се се олакшало програмирање у МИПС асемблерском језику. Њих асемблер преводи у две или више правих инструкција из скупа инструкција. На пример:

- li rd, Imm- Учитавање константе *Imm* у регистар rd
- la rd, Address - Учитавање адресе *Address* у регистар

2.4 Регистри

Регистри представљају малу, веома брзу меморију, која је део процесора. МИПС процесори могу вршити операције само над садржајима регистара и специјалним константама које су део инструкције.

У МИПС архитектури, постоје 32 регистра опште намене [26]: *$0 - $31*. Два регистра имају другачије понашање од осталих:

- **$0** Увек има вредност 0, без обзира на садржај.
- **$31** Увек се користи за адресу повратка из функције на коју се скочи инструкцијом *jal*.

У наставку ће бити описани регистри опште намене [26]:

- **at** - Резервисан је за псеудоинструкције које асемблер генерише.
- **v0, v1** - Користе се за враћање резултата при повратку из неке функције. Резултат може бити целобројног типа или број записан у фиксном зарезу.
- **a0 - a3** - Користе се за прослеђивање прва 4 аргумента функцији која се позива.
- **t0 - t9** - По конвенцији која је описана у поглављу 2.10, функције могу користи ове регистре без потребе да њихов садржај сачувају пре тога. Зато се могу користити као "променљиве" при израчунавањима, само се мора водити рачуна да ће се при позиву неке друге функције садржај тих регистара изгубити. Ово су регистри које чува функција позиваоц (енгл. *caler saved registers*).
- **s0 - s7** - По конвенцији која је описана у поглављу 2.10, функција мора обезбедити да ће садржај ових регистара при уласку у функцију бити исти као и при изласку из ње. То се обезбеђује или некоришћењем, или чувањем садржаја на стеку на почетку функције, пре коришћења, и скидањем садржаја са стека пре изласка из функције. Дакле, ово су регистри које чува позвана функција (енгл. *calee saved registers*).
- **k0, k1** - Резервисано за систем прекида, који након коришћења не враћа садржај ових регистара на почетни. Како се прекид не позива из програма који се тренутно извршава, нема примене позивне конвенције. То значи да се садржај регистара које прекинути програм користи може пореметити. Због тога, систем прекида прво сачува садржаје регистара опште намене, који су важни за програм који се у том тренутку извршавао, и чији садржај планира да мења. У те сврхе се користе ови регистри.

- **gp** - Користи се у различите сврхе. У коду који не зависи од позиције (енгл. *Position Independent Code*, скраћено *PIC*), свом коду и подацима се приступа преко табеле показивача, познате као GOT (скраћено од енгл. *Global Offset Table*). Регистар *$gp* показује на ту табелу. PIC је код који се може извршавати на било којој меморијског адреси, без модификација. PIC се најчешће користи за дељене библиотеке, при чему се заједнички код библиотеке може учитати у одговарајуће локације адресних простора различитих програма који је користе.

  У регуларном коду који зависи од позиције, регистар *$gp* се користи као показивач на средину у статичкој меморији. То значи да се подацима који се налазе 32 KB лево или десно од адресе која се налази у овом регистру може приступити помоћу једне инструкције. Дакле, инструкције *load* и *store* које се користе за учитавање, односно складиштење података, се могу извршити у само једној инструкцији, а не у две као што је то иначе случај. У пракси се на ове локације смештају глобални подаци који не заузимају много меморије. Оно што је битно је да овај регистар не користе сви системи за компилацију и сва окружења за извршавање.

- **sp** - Показивач на стек. Оно што је битно је да стек расте наниже. Потребне су специјалне инструкције да би се показивач на стек повећао и смањио, тако да МИПС ажурира стек само при позиву и повратку из функције, при чему је за то одговорна функција која је позвана. *$sp* се при уласку у функцију прилагођава на најнижу тачку на стеку којој ће се приступати у функцији. Тако је омогућено да компилатор може да приступи променљивама на стеку помоћу константног помераја у односу на *$sp*.

- **fp** - Познат и као *$s8* , показивач на стек оквир. Користи се од стране функције, за праћење стања на стеку, за случај да из неког разлога компилатор или програмер не могу да израчунају померај у односу на *$sp*. То се може догодити уколико програм врши проширење стека, при чему се вредност проширења рачуна у току извршавања. Ако се дно стека не може израчунати у току превођења, не може се приступити променљивама помоћу *$sp*, па се на почетку функције *$fp* иницијализује на константну позицију која одговара стек оквиру функције. Ово је локално за функцију.

- **ra** - Ово је подразумевани регистар за смештање адресе повратка и то је подржано кроз одговарајуће инструкције скока. При уласку у функцију овај регистар обично садржи адресу повратка функције, тако да се функције углавном завршавају инструкцијом *jr $ra*, али у принципу, може се користити и неки други регистар. Због неких оптимизација које врши процесор, препоручује се коришћење регистра *$ra*. Функције које позивају друге функције морају сачувати садржај регистра *$ra*. Ово се разликује од конвенција које се користе на архитектурама x86, где инструкција позива функције адресу повратка смешта на стек.

Постоје два специјална регистра *Hi* и *Lo*, који се користе само при множењу и дељењу. Ово нису регистри опште намене, те се не користе при другим инструкцијама. Не може им се приступити директно, већ постоје специјалне инструкције *mfhi* и *mflo* за премештање садржаја ових регистара [26]. Инструкција *mfhi* је облика *mfhi rd*, и она премешта садржај регистра *Hi* у регистар *rd*, док инструкција *mflo* премешта садржај регистра *Lo*.

У оквиру МИПС архитектуре PC (скраћено од енгл. *Program Counter*) је веома специфичан јер у проточном систему процесор ради са више инструкција истовремено [26]. Свака инструкција пролази кроз 5 фаза, као што је то описано у поглављу 2.7, а PC садржи адресу инструкције која је тренутно у фази извршавања и не може му се директно приступати. Може се индиректно мењати садржај PC-а инструкцијама гранања и скока.

Постоје два формата за адресирање: коришћењем бројева *$0-$31* или коришћењем имена *$a0, $s0, $t0* и слично.

МИПС садржи и подршку за рад са покретним зарезом, али то није коришћено при развоју интерпретатора, те се због тога не описује. Више о регистрима и инструкцијама које се користе при раду са покретним зарезом може се прочитати у књизи [26].

2.5 Начини адресирања

На платформи МИПС постоје четири начина адресирања [26]:

Регистарско

Регистарско адресирање се користи у *jr* инструкцији. Пошто је величина регистра 32 бита, а било која адреса је такође 32 бита, на овај начин се

може приступити било којој адреси. У наставку је дат пример инструкције са регистарским адресирањем.

jr rs

PC-релативно

PC-релативно адресирање се користи у *beq*, *bne* и другим инструкцијама гранања. Ово су инструкције И типа. Константа која представља адресу може имати највише 16 бита, што значи да се не може приступити великом броју адреса. Ове инструкције се углавном користе за скокове на инструкције које су близу, а адреса се рачуна као PC + константа. У наставку је дат пример инструкције са PC-релативним адресирањем.

bne rs, rt, imm

Псеудо-директно

Директно адресирање би значило да се у инструкцији наводи адреса. То је немогуће зато што адреса има 32-бита, колико и цела инструкција. Псеудо-директно адресирање се користи у инструкцији *j*. За операцију се користи 6 бита, тако да за адресу остаје 26. Адресирање се изводи тако што се горња 4 бита позајме од PC-а, а на крај се дода 00. У наставку је дат пример инструкције са псеудо-директним адресирањем.

j imm

Базно

Базно адресирање подразумева рачунање адресе као збира садржаја регистра и помераја. Овај тип адресирања примењује се код инструкција *lw* и *sw*. Константа која представља померај може имати највише 16 бита.

lw rt, offset(rs)

2.6 Системски позиви

Помоћу инструкције *syscall* могу се позивати одређени системски позиви. Неопходно је у регистар *$v0* уписати кôд одговарајућег системског позива, који

Табела 2.1: Пример кодова системских функција и начина прослеђивања аргумената.

функција	код	аргументи
exit	10	
open	13	$a0 = име датотеке $a1=флегови $a2=мод
read	14	$a0 = фајл дескриптор $a1 = бафер у који се уписује $a2 = колико бајтова се чита
write	15	$a0 = фајл дескриптор $a1 = бафер који се уписује $a2 = колико бајтова се уписује
close	16	$a0 = фајл дескриптор

зависи од архитектуре. Аргументи се системском позиву прослеђују кроз регистре *$a0-$a3*, док се повратна вредност налази у регистрима *$v0* и *$v1*. Они системски позиви који могу резултовати неуспехом, статус извршавања смештају у регистар *$a3* (0 за успешно, различито од 0 за неуспешно).

У табели 2.6 приказани су кодови неких системских позива и кроз које регистре им се прослеђују аргументи.

2.7 Проточни систем

Проточна обрада (енгл. *pipeline*) настала је из чињенице да различите фазе у току извршавања инструкције користе различите ресурсе. Уколико би трајање фаза било једнако, могао би се направити систем у ком по завршетку једне фазе текуће инструкције, почиње та фаза наредне инструкције. Овакав начин обраде инструкција назван је проточна обрада [26]. Да би се омогућило једнако трајање фаза, RISC дефинише скуп инструкција у коме је количина посла која се извршава у оквиру једне инструкције максимално смањена. Да би се постигла једнакост трајања фазе декодирања инструкције, већина RISC архитектура, међу којима је и МИПС, има фиксирану величину инструкција [26].

Предуслов за ефикасну проточну обраду је коришћење кеш меморије ради убрзања приступа меморији. Кеш меморија је мала, брза, локална меморија која служи за складиштење копија меморијских података [26]. Сваки податак у

кешу има показивач на адресу у главној меморији, на којој се налази. У кешу се чувају подаци које је процесор читао из меморије у најскорије време. Уколико је кеш пун, пребришу се најстарији подаци. При читању, прво се проверава да ли се податак налази у кешу. Уколико се не налази, долази до "промашаја кеша". Промашај кеша се дешава ретко, отприлике у 10% случајева.

У RISC архитектури кеш је део процесора и везан је за проточну обраду, док је код CISC архитектуре кеш меморија део меморијског система. МИПС има одвојен инструкцијски кеш и кеш података, тако да се у исто време може читати инструкција из меморије и читати или уписивати подаци.

Проточна обрада на МИПС платформи

Извршавање сваке инструкције је подељено у 5 фаза, при чему свака фаза има фиксирано трајање. Прва, трећа и четврта фаза трају по један радни такт процесора , док друга и пета трају по пола такта, тако да укупно извршавање инструкције траје 4 такта [26]. У наставку ће бити описана свака од фаза.

1. Дохватање инструкције из инструкцијског кеша - декодирање инструкције (енгл. instruction fetch, скраћено IF)

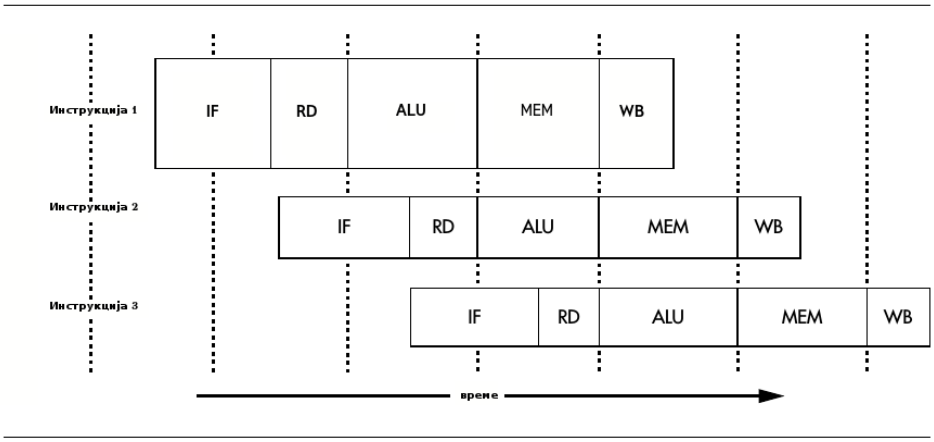
2. Читање садржаја одговарајућих регистара (операнада инструкције) (енгл. read register, скраћено RR)

3. Извршавање аритметичко/логичких операција у једном радном такту (операције у покретном зарезу, множење и дељење се раде другачије) (енгл. arithmetic/logic unit, скраћено ALU)

4. Фаза у којој се дохватају/пишу подаци у меморију. Око 75% инструкција не ради ништа у овој фази, али она постоји да не би дошло до тога да две инструкције у исто време желе да приступе кешу (енгл. memory, скраћено MEM)

5. Вредност(резултат) операције се уписује у регистар (енгл. write back, скраћено WB)

На слици 2.1 је приказано извршавање 3 инструкције на платформи МИПС. Када се заврши прва фаза прве инструкције, из меморије се дохвата друга инструкција. Са завршетком прве фазе друге инструкције, почиње прва фаза треће инструкције итд.



Слика 2.1: Проточна обрада на платформи МИПС.

Ефикасна проточна обрада има за последицу следеће [26]:

1. Све инструкције морају бити исте дужине, што ограничава комплексност инструкција. На платформи МИПС је та дужина 32 бита. Више о МИПС инструкцијама речено је у поглављу 2.3
2. Подацима се приступа само у фази 4
3. Приступ меморији омогућен само помоћу инструкција *load* и *store*
4. Аритметичко/логичке инструкције имају три операнда
5. Нема специјалних инструкција чија комплексност лоше утиче на ефикасност проточне обраде
6. Постоје 32 регистра опште намене чија величина зависи од тога да ли је архитектура 32-битна или 64-битна. Више о МИПС регистрима речено је у поглављу 2.4

2.8 Слот закашњења

Структура проточног система МИПС процесора је таква да када инструкција скока или гранања дође у фазу извршавања, инструкција након скока/гранања ће већ бити започета. Пошто ово може бити предност, дефинисано је да се инструкција која следи након скока/гранања мора извршити пре скока. Дакле, у односу на класичан асемблерски код, врши се пермутација инструкције

скока/гранања и инструкције испред ње. Позиција инструкције која следи након скока/гранања назива се слот закашњења гранања (енгл. *Branch delay slot*). Када је то могуће, углавном се у слот закашњења смешта инструкција која би иначе била пре скока/гранања. Кодом 2.8 представљено је како се може искористити слот закашњења за извршавање инструкције *move* пре инструкције скока.

```
jalr t9
move a0, s0
```

Код 1: Пример извршавања инструкције *move* у слоту закашњења

То се не може увек применити када је у питању условни скок, јер инструкција која се налази у слоту закашњења мора бити безопасна по оба исхода поређења. У случајевима када се ништа не може ставити у слот закашњења, попуњава се инструкцијом *nop* [26].

Још једна последица проточне обраде је што подаци дохваћени *load* инструкцијом стижу тек након инструкције која се налази иза *load*. Последица је да се ти подаци не могу користити одмах у наредној инструкцији. Позиција инструкције која следи одмах иза *load* се назива слот закашњења учитавања (енгл. *Load delay slot*). На модерним процесорима резултат *load* инструкције је блокиран. Ако се покуша са приступом резултату, зауставља се извршавање и чека се да подаци стигну [26].

Подразумевано асемблер води рачуна о слотовима закашњења, али како он не може увек пронаћи најбоље решење, може се навести директива *.set noreorder*, чиме се попуњавање слота закашњења препушта програмеру.

2.9 Структура програма

У Фон Најмановим архитектурама, међу којима је и МИПС, подаци и кôд се налазе у истој меморији. То захтева да се меморија подели на два сегмента, сегмент података и сегмент кода.

У сегменту података налазе се декларације имена променљивих које се користе у програму, чиме се за сваку променљиву алоцира меморија. Означава се асемблерском директивом *.data*. Декларација се састоји из имена променљиве, након чега следи „:” тип и вредност [26].

Сегмент кода се означава директивом *.text* и састоји се из инструкција. Почетна тачка за извршавање програма означава се лабелом *main:*, а крај *main* дела се означава позивом *syscall*. Овим позивом, контрола се преноси оперативном систему, који на основу садржаја регистра *$v0* зна шта треба да уради [26].

```
  .data
 hello:     .asciiz "Zdravo svete!"
 length:    .word   12
            .globl main
  .text

 main:
    li  a0, 1
    la  a1, hello
    lw  a2, length
    li  v0, 15
    syscall
end:
    li        v0, 10
    syscall
```

Код 2: Програм који исписује поздравну поруку, у МИПС асемблерском језику.

Кодом 2 је приказан пример програма у МИПС асемблерском језику. Овај програм исписује поздравну поруку на стандарадни излаз, на следећи начин:

1. Најпре се директивом *.data* означи да након те линије почиње декларација променљивих.

2. Затим се декларише ниска *„hello”*, која је типа *asciiz* (ниске овог типа се завршавају *NULL* карактером, и фиксиране су дужине). Вредност ниске се постави на *„Zdravo svete!”*.

3. Након тога се декларише променљива која представља дужину ниске *„hello”*.

4. Директива *.globl* означава да је симбол *main* доступан ван ове асемблерске датотеке.

5. Након тога се означи почетак кодног сегмента директивом *.text*.

6. Лабела *main* означава почетак прве инструкције која се извршава.

7. У регистар *$a0* учита се први аргумент системског позива *write* - 1 означава да се резултат исписује на стандардни излаз.
8. У регистар *$a1* учита се други аргумент, ниска која се исписује.
9. У регистар *$a2* учита се трећи аргумент који представља дужину ниске.
10. У регистар *$v0* учита се 15, што је кôд за системски позив *write*.
11. Након тога се контрола прослеђује оперативном систему који исписује ниску.
12. Лабела *end:* означава крај програма.
13. У регистар *$v0* учита се кôд за системски позив *exit*.
14. Након тога се позива оперативни систем да прекине извршавање програма.

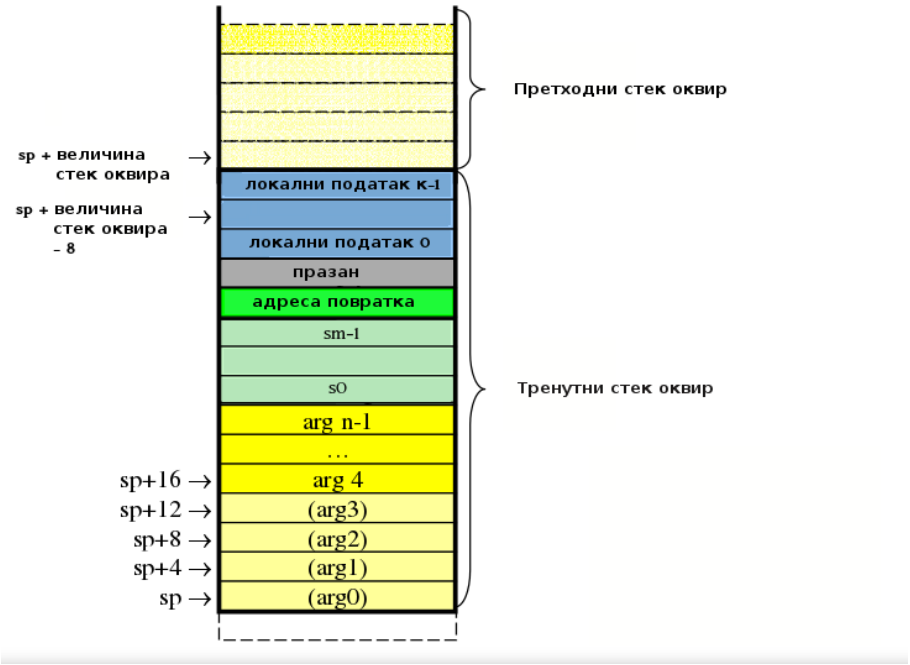
2.10 Позивна конвенција

Позивна конвенција представља правила рада са регистрима и стек оквиром приликом позива функције. Не постоји јединствена МИПС позивна конвенција која се мора примењивати, тако да ће овде бити представљена она која је коришћена при имплементацији МИПС интерпретатора у оквиру Дартино виртуелне машине.

При позиву неке функције, на стеку се прави посебан стек оквир, и то се дешава за сваки позив, односно за сваку инстанцу једне функције. Организација стек оквира дефинише однос између позиваоца и позване функције, који се односи на начин прослеђивања аргумената и резултата, и дељења регистара. Поред тога, дефинише и организацију локалног складишта позване функције унутар њеног стек оквира [24].

Стек оквир функције се дели на 5 региона, што је приказано на слици 2.2. У наставку ће бити описан сваки од њих [24].

1. Први регион представља простор за смештање аргумената, који се прослеђују функцијама које се позивају из тренутне функције (тренутна функција је она чији се стек оквир налази на врху стека). Прва четири места



Слика 2.2: Организација стек оквира.

никад не користи тренутна функција, већ функција која је позвана. Уколико је потребно проследити више од 4 аргумента, они се смештају на адресе *$sp+16*,*$sp+20*, *$sp+24* итд.

2. Други регион представља простор за смештање садржаја регистара које чува позвана функција (*$s0-$s7*), које тренутна функција жели да користи. При уласку у функцију, садржаји регистара које ће функција користити се сачувају на стеку, а пре изласка се врате на оригиналне. Циљ је да функција која је позвала тренутну функцију има исте садржаје ових регистара пре и после позива.

3. У трећем региону се смешта адресе повратка функције, односно садржај регистра *$ra*. Ова вредност се копира на стек при уласку у функцију, и копира назад у регистар *$ra* пре изласка из функције.

4. Четврти регион представља поравнање стек оквира на адресу која је дељива са 8. То је овде неопходно да би секција локалних података почела на адреси која је дељива са 8.

5. Пети регион је секција за складиштење локалних података. Функција овде мора резервисати место за све локалне променљиве, као и за све привремене регистре (*$t0-$t9*), које мора сачувати пре позивања друге функције. Ова секција такође мора бити поравната на 8.

Позивна конвенција уводи и нека додатна правила:

- Позвана функција не мора да сачува на стеку повратну вредност, већ се за то користе регистри *$v0* и *$v1*.
- Вредност показивача на стек увек мора бити дељива са 8. Ово обезбеђује да се 64-битни подаци увек могу сместити на стек без генерисања грешке поравнања адреса у току извршавања. Из овога следи да величина сваког стек оквира мора бити дељива са 8.
- Вредности регистара које чува функција позиваоц се не морају чувати при позиву функције. Позвана функција може мењати ове регистре, без чувања њихове претходне вредности.
- Прва 4 места у секцији аргумената су позната као слотови аргумената - меморијске локације које су резервисане за смештање 4 аргумента *$a0-$a3*. Функција не чува ништа на овим позицијама, јер се аргументи прослеђују кроз регистре *$a0-$3*. Међутим, позвана функција може у слотовима аргумената сачувати вредности ових регистара, уколико то жели. Свака функција мора у оквиру свог стек оквира алоцирати меморију за максималан број аргумената за било коју функцију коју позива. Уколико функција коју позива има мање од 4 аргумента, онда мора резервисати 4.

Глава 3

Програмски језик Дарт

Дарт је програмски језик вишег нивоа који развија компанија Гугл, почевши од октобра 2011. године. У наставку су наведени главни циљеви овог језика [5].

- Добра подршка за рад кроз библиотеке.
- У процесу развоја апликације наилази се на учестале проблеме и грешке. Дарт је дизајниран тако да руковање оваквих ситуација учини лакшим, коришћењем *async/await*, генератора, интерполације ниски и слично.
- Продуктивност и стабилност, које воде до делотворног решења за велике апликације.

У овој глави описан је програмски језик Дарт. У поглављу 3.1 описане су основне карактеристике овог језика. У поглављу 3.2 описано је на који начин су подржани типови, док је у поглављу 3.3 описан рад са функцијама. У поглављу 3.4 описане су специфичности при раду са класама. Подршка за асинхроност описана је у поглављу 3.5, док је рад са библиотекама је описан у поглављу 3.6. Подршка за конкуретне процесе описана је у поглављу 3.7. У поглављу 3.8 описан појам метаподатака. Разлике Дарта у односу на ЈаваСкрипт описане су у поглављу 3.9.

3.1 Основне карактеристике

Дарт је објектно-оријентисани језик, који се заснива на класама и једноструком наслеђивању. Дарт се обично користи код програмирања веб-апликација,

али се може користити и за мобилне апликације, веб-сервере, и програмирање система са уграђеним рачунаром.

Дарт је уско везан за ЈавaСкрипт. Програми написани у Дарту се могу превести у ЈавaСкрипт. При покретању у веб прегледачу, преводи се у ЈавaСкрипт помоћу *dart2js* компилатора, при чему се врши оптимизација ЈавaСкрипт кода. У неким случајевима, кôд написан у Дарту се брже извршава од одговарајућег кода написаног у ЈавaСкрипту [17, 6].

У наставку су описани неки од најважнијих концепата програмског језика Дарт [17, 6].

- Свака променљива је објекат, при чему сви објекти наслеђују класу Object.
- Навођење типова је опционо, али постоји подршка и за статичку проверу типова.
- Могу се креирати анонимне функције и затворења (енгл.*closure*).
- Нема кључних речи *public*, *private* и *protected*. Идентификатори који почињу доњом цртом су приватни.
- Постоји подршка за асинхроно програмирање и за конкурентно извршавање.
- Метаподаци се користе за пружање додатних информација о коду.

3.2 Типови

У Дарту се типови не морају наводити, али је препоручљиво. У званичном упутству стоји да би требало наводити типове у класама и методама. Тиме се смањује количина грешака, при чему не постоји провера типова у току извршавања, већ само у току превођења. Навођењем типова кôд је уједно лакши за читање и пружа бољу документацију [17, 6].

3.3 Функције

Пошто је Дарт чисто објектно-оријентисани језик, и функције су објекти, типа *Function*. То значи да се функције могу додељивати променљивама, или прослеђивати као аргументи функције [17, 6].

Функције које садрже само један израз, могу се записивати у скраћеном облику, што је представљено кодом 3.

```
bool funkcija(int arg) => izraz;
```

Код 3: Пример скраћеног облика записивања функција.

Такође, у Дарту постоје анонимне функције које се могу креирати на начин приказан кодом 4. Промењливој *func* додељује се анонимна функција која штампа поруку великим словима [17, 6].

```
var func = (msg) => print(msg.toUpperCase());
func('Hello');
```

Код 4: Пример коришћења анонимне функције која штампа елемент листе.

Могу се креирати затворења, која представљају функције која памте контекст у ком су креиране (могу приступати промењливама које су дефинисане у контексту у ком су креиране). Пример затворења дат је кодом 5. Функција *makeAdder* враћа као резултат функцију која додаје број *addBy* аргументу те функције, односно враћа затворење. Ово затворење ће имати приступ промењливој за коју је креирано. На почетку је промењлива *pom* иницијализована на 3. Након тога се креира објекат функције *makeAdder*, са аргументом *pom*, који се додели промењливој *add*. При позиву ове функције са аргументом 5, резултат ће бити 8. Након што променимо вредност промењливе *pom* на 7, позив функције за аргумент 5 ће вратити 12, јер се унутар затворења приступа тренутној вредности промењливе [17, 6].

```
Function makeAdder(num addBy) {
  return (num i) => addBy + i;
}

main() {
  var pom = 3;
  var add  = makeAdder(pom);
  add(5); // 8
  pom = 7;
  add(5); // 12
}
```

Код 5: Пример функције која враћа затворење.

3.4 Класе

Као објектно-оријентисани језик, Дарт је заснован на класама и једноструком наслеђивању. Иако свака класа може имати само једну надкласу, могуће је имплементирање више интерфејса, при чему свака класа може бити интерфејс за неку другу класу.

Класа може садржати тело друге класе (енгл. *Mixins*). Кôд неке друге класе се може искористити помоћу кључне речи *with* након које следи име класе чији се кôд копира (*Mixin*). Кодом 6 представљена је класа која користи уметнуте класе *Musical* и *Aggressive* [17, 6].

```
class Maestro extends Person
    with Musical, Aggressive {
  Maestro(String maestroName) {
    name = maestroName;
    canConduct = true;
  }
}
```

Код 6: Пример класе која користи *Mixin*.

Да би се класа могла уметнути она се креира као класа која нема конструктор. Кодом 7 представљена је класа која се може уметнути.

```
abstract class Musical {
  bool canPlayPiano = false;
  bool canCompose = false;
  bool canConduct = false;

  void entertainMe() {
    if (canPlayPiano) {
      print('Playing piano');
    } else if (canConduct) {
      print('Waving hands');
    } else {
      print('Humming to self');
    }
  }
}
```

Код 7: Пример *Mixin* класе.

Још једна важна карактеристика Дарта је да се конструктори не наслеђују. Уколико класа нема дефинисан конструктор, Дарт ће јој доделити подразумевани. Не може се дефинисати више конструктора, али се могу креирати именовани конструктори. Кодом 8 представљено је дефинисање именованог конструктора [17, 6].

```
class Point {
  num x;
  num y;

  Point(this.x, this.y);

  Point.fromJson(Map json) {
    x = json['x'];
    y = json['y'];
  }
}
```

Код 8: Пример дефинисања именованог конструктора.

3.5 Асинхроност

Дарт има више начина за подршку асинхроног програмирања, а најчешће употребљивани су *async* функције и *await* изрази. Функције које враћају *Future* или *Stream* објекат су асинхроне. Асинхроне функције враћају као резултат операцију која ће се извршити у будућности.

Future представља средство за враћање вредности некад у будућности. Када се позове функција која враћа *Future*, та функција стане са извршавањем и врати непотпуни *Future* објекат. Касније, када вредност коју треба да врати буде доступна, *Future* објекат се допуни том вредношћу или врати грешку. Вредност коју представља *Future* може се добити помоћу *async* и *await*, или помоћу *Future* API-ја [17, 6].

Да би се користио *await*, кôд мора бити унутар функције која је означена као *async*. Пример у коме се помоћу *await* чека на резултат асинхроне функције, дат је кодом 9. Функција *lookUpVersion*, је нека асинхрона функција која враћа *Future* [17, 6].

```
checkVersion() async {
  var version = await lookUpVersion();
  if (version == expectedVersion) {
    // Do something.
  } else {
    // Do something else.
  }
}
```

Код 9: Пример употребе `await`.

Future API се користио док није додата подршка за *async* и *await*, а данас се углавном користи када су нам потребне функционалности које *async* и *await* не могу пружити.

Stream представља секвенцу асинхроних догађаја, која може садржати догађаје генерисане од стране корисника, или податке прочитане из датотека. Вредности из *Stream*-а могу се добити помоћу *await-for* или *listen* из *Stream* API-ја. Пример асинхроне *for* петље којом се итерира кроз догађаје стрима дат је кодом 10. Како је овде стрим низ целобројних догађаја, у петљи се прихвата догађај стрима и додаје на суму. Када се петља заврши, функција се паузира док не стигне следећи догађај. Функција која користи *await for* петљу мора се означити са *async* [17, 6].

```
Future<int> sumStream(Stream<int> stream) async {
  var sum = 0;
  await for (var value in stream) {
    sum += value;
  }
  return sum;
}
```

Код 10: Пример употребе прихватања догађаја `Stream`-а.

3.6 Библиотеке

Свака Дарт апликација је библиотека, чак и ако нема кључну реч *library*. Кључна реч *import* користи се за укључивање одређене библиотеке, при чему постоји могућност лењог учитавања библиотеке помоћу *deferred as*. Лењо учитавање може смањити време покретања апликације, а користи се и када постоје

функционалности које се ретко користе. Пример лењог учитавања дат је кодом 11. Када нам је потребна лењо учитана библиотека, укључујемо је функцијом *loadLibrary()*, што је представљено кодом 12 [17, 6].

```
import 'package:deferred/hello.dart' deferred as hello;
```

Код 11: Пример лењог учитавања класе.

```
greet() async {
  await hello.loadLibrary();
  hello.printGreeting();
}
```

Код 12: Пример укључивања лењо учитане библиотеке.

3.7 Изолате

Дарт програм се увек извршава у једној нити. Нема конкурентног извршавања са дељењем меморије, већ је конкурентност подржана кроз изолате (енгл. *Isolates*).

Изолата је јединица конкурентног извршавања. Између изолата нема дељења меморије, а комуникација се обавља слањем порука кроз портове *ReceivePort* и *SendPort*. Свака изолата има своју хип меморију. Сваки Дарт програм се састоји из бар једне изолате, која се назива главна изолата (енгл. *main isolate*) [17, 6]. При превођењу на ЈаваСкрипт, изолате се преводе у веб-воркере (енгл. *web worker* [1]).

Пример комуникације изолата дат је кодом 13. Функција *Isolate.spawn* покреће нову изолату *echo*. Овој изолати се прослеђује порт на ком изолата која ју је креирала (главна изолата) чека одговор. У оквиру ње се отвара порт на ком се примају одговори, а затим се тај порт шаље главној изолати. Након тога се у *for* петљи помоћу *await* чека да стигну подаци. Када податак стигне, шаље се на *SendPort*. У *main* функцији се након креирања изолате, прима порт који је *echo* изолата послала, а након тога се позива функција *sendReceive*, која шаље „hello" на порт изолате *echo*, и враћа њен одговор. На крају се исписује одговор који је *echo* изолата послала.

[1] Појам који означава нешто између процеса и нити. Извршава се у позадини и тиме не утиче на одзивност система.

```
main() async {
  var receivePort = new ReceivePort();
  await Isolate.spawn(echo, receivePort.sendPort);

  var sendPort = await receivePort.first;

  var msg = await sendReceive(sendPort, "hello");
  print('received $msg');
}

echo(SendPort sendPort) async {
  var port = new ReceivePort();

  sendPort.send(port.sendPort);

  await for (var msg in port) {
    var data = msg[0];
    SendPort replyTo = msg[1];
    replyTo.send(data);
    port.close();
  }
}

Future sendReceive(SendPort port, msg) {
  ReceivePort response = new ReceivePort();
  port.send([msg, response.sendPort]);
  return response.first;
}
```

Код 13: Пример комуникације *echo* и *main* изолате.

3.8 Метаподаци

Метаподаци се користе да пруже додатне информације о коду. Означавање метаподатака почиње карактером @ иза чега следи константа у време превођења (енгл. *compile-time constant*), нпр. *deprecated*, или позив неког константног конструктора. У Дарту постоје три типа анотација (енгл. *metadata annotations*): *@deprecated*, *@override*, и *@proxy*. Анотација *@override* се користи за наглашавање намерног мењања метода надкласе. Анотација *@deprecated* се користи да означи да је неко својство застарело, док се анотација *@proxy* користи да омогући креирање објеката који имплементирају типове који нису познати у време превођења.

Пример употребе *@override* дат је кодом 14 [17, 6].

```
class A {
  @override
  void noSuchMethod(Invocation mirror) {
    // ...
  }
}
```

Код 14: Пример употребе @override.

Могу се дефинисати нове анотације метаподатака, а пример креиране анотације дат је кодом 15. Креирана је класа која садржи константи конструктор, који има два аргумента. Тиме је дефинисан метаподатак *@todo* који има два аргумента. Пример коришћења креираног метаподатка дат је кодом 16 [17, 6].

```
library todo;

class todo {
  final String who;
  final String what;

  const todo(this.who, this.what);
}
```

Код 15: Пример креирања анотације метаподатака.

```
import 'todo.dart';

@todo('x', 'make this do something')
void doSomething() {
  print('do something');
}
```

Код 16: Пример коришћења креиране анотације.

Метаподаци се могу јавити пре библиотеке, класе, конструктора, функције, поља, параметра, декларације промењиве, *import* или *export* директиве, типског параметра или *typedef*-а [17, 6].

3.9 Специфичности у односу на ЈавaСкрипт

У наставку су описане неке карактеристике Дарта, по којима се разлику од ЈавaСкрипта [17, 6].

- Дарт подржава типове.
- Дарт подржава дефинисање класа.
- У Дарту само false представља нетачно, док у ЈавaСкрипту то важи и за 0, *null*, *undefined* и "".
- Дарт подржава наслеђивање помоћу кључне речи *extends*.
- Дарт уводи изолате као подршку за конкурентно извршавање.
- У Дарту постоји *foreach* петља.
- Дарт не садржи *undefined*, већ само *null*.

Глава 4

Виртуелна машина Дартино

Дартино је пројекат компаније Гугл, који је имао за циљ повећање продуктивности при развоју апликација за системе са уграђеним рачунаром, коришћењем програмског језика Дарт. Рад на овом пројекту започет је јануара 2015. године. Даља улагања у овај пројекат заустављена су почетком септембра 2016.године, због закључка да *„Масовно коришћење вишег програмског језика у свету система са уграђеним рачунаром није реално још неко време, због велике фрагментације хардвера"* [16]. Свакако, резултати до којих се дошло биће искоришћени при развоју веб и мобилних апликација у Дарту.

У овој глави је описана Дартино виртуелна машина. У поглављу 4.1 је описано чему служи Дартино и које су компоненте виртуелне машине, док је у поглављу 4.2 представљена архитектура система. У поглављу 4.3 су представљене различите врсте процеса које су подржане у оквиру Дартина, које чине лаки процеси, изолате и влакна. У поглављу 4.4 описана је подршка за корутине, а у поглављу 4.5 описан је поступак динамичког отпремања метода. У поглављу 4.6 описане су најзначајније библиотеке у оквиру Дартина, док је у поглављу 4.7 описана употреба Дартина из командне линије и улога Дарта и Дартина при покретању програма.

4.1 Опис виртуелне машине

Виртуелна машина представља софтверску симулацију физичког рачунара. У зависности од употребе и степена сличности са физичким рачунаром, виртуелне машине се деле на виртуелне машине за симулирање система и виртуелне машине за симулирање процеса [18].

Дартино виртуелна машина представља виртуелну машину за симулирање процеса. Виртуелне машине за симулирање процеса додају слој изнад оперативног система, који служи за симулирање програмског окружења, које је независно од платформе, и у оквиру ког се може извршавати један процес. Извршавање више процеса може се постићи креирањем више инстанци виртуелне машине. Овај тип виртуелних машина обезбеђује висок ниво апстракције. Програми се пишу у програмском језику вишег нивоа, и могу се извршавати на различитим платформама. Имплементација виртуелне машине се заснива интерпретатору [18].

Као што је речено у уводу, Дартино виртуелна машина је настала са циљем да се писање апликација за микроконтролере олакша. Како би се направило боље окружење за рад, потребна је отворена и приступачна платформа, која ће већини програмера бити позната. Потребно је омогућити и статичку анализу кода, допуњавање кода, библиотеке са разним функционалностима, и тиме развијање апликација учинити ефикаснијим.

Пре Дартина је било могуће програмирати микропроцесоре у Дарту, али не и микроконтролере. Због мале количине меморије и мале брзине микроконтролера, виртуелне машине које се на њих смештају морају бити што једноставније. Виртуелна машина обично садржи компилатор изворног кода, некакву компоненту за извршавање, сакупљач отпадака, омогућује препознавање класа и објеката (објектни модел), и има подршку за дебаговање. Како би се уклопила у поменута ограничења које намећу миктроконтролери, Дартино виртуелна машина је поједностављена тиме што не садржи компилатор. Она садржи преостале четири компоненте, које су неопходне када је у питању виши програмски језик као што је Дарт. Направљен је систем у коме су компилатор и окружење за извршавање раздвојени, као што је случај код компилатора GCC или LLVM, при чему је систем за дебаговање поједностављен и смањен [23].

За програмски језик Дарт постоји Дарт виртуелна машина, која се користи за веб програмирање. У оквиру Дартина је искоришћен постојећи JIT компилатор за Дарт, који се покреће помоћу Дарт виртуелне машине. Тај компилатор може да се налази било где, обично на рачунару где се развија апликација, и он комуницира са окружењем за извршавање које се потенцијално налази на другом систему (микроконтролеру) [23]. Интерфејс за комуникацију је једноставан, јер окружење за извршавање мора бити веома једноставно, како би одговарало карактеристикама миктроконтролера.

Окружење за извршавање садржи командни API, преко ког му компилатор може затражити да уради одређене задатке. Испод тога се заправо налази стек машина, тако да се путем овог API-ја може поставити ново стање на стек окружења за извршавање, могу се правити класе, и дефинисати објекти и методе. Стек машина представља виртуелну машину код које је меморијска структура представљена као стек. То значи да се операције извршавају скидањем операнада са стека, применом операције и уписивањем резултата на стек. Компилатор класама може доделити индентификаторе, помоћу којих при примању објекта назад, може утврдити којој класи објекат припада. На који начин компилатор шаље поруке окружењу за извршавање приказано је на слици 4.1 [23].

Слика 4.1: Слање порука JIT компилатора окружењу за извршавање.

Сва комуникација се врши преко одговарајућих протокола који се обично базирају на TCP/IP протоколу, чиме је омогућено удаљено извршавање. На овај начин велики део посла се пребацује на компилатор, а окружење за извршавање се максимално упрошћава.

Помоћу протокола за комуникацију компилатора и окружења за извршавање, може се постављати и мењати структура програма [23]. Постављање структуре програма обухвата додавање нових класа и метода на стек окружења за извршавање. Мењање структуре програма обухвата:

Мењање табела метода

: Омогућена је висока интерактивност са системом, што подразумева мењање понашања програма у току извршавања, нпр. додавањем новог метода класи, у одређеном тренутку. Компилатор ће окружењу за извршавање затражити да се промени табела метода.

Мењање шема

Мењање шема се односи на мењање класе, нпр. додавањем нових поља. Оно што се дешава у тој ситуацији је да се врши пролаз кроз све инстанце класе, и оне се ажурирају и прилагођавају новом формату класе.

Дебаговање

У оквиру дебаговања, могуће је [23]:

1. Покретање дебагера
2. Постављање зауставних тачака
3. Рестартовање дебагера

Систем за дебаговање је једноставан и ограничен, јер не зна ништа о изворном коду.

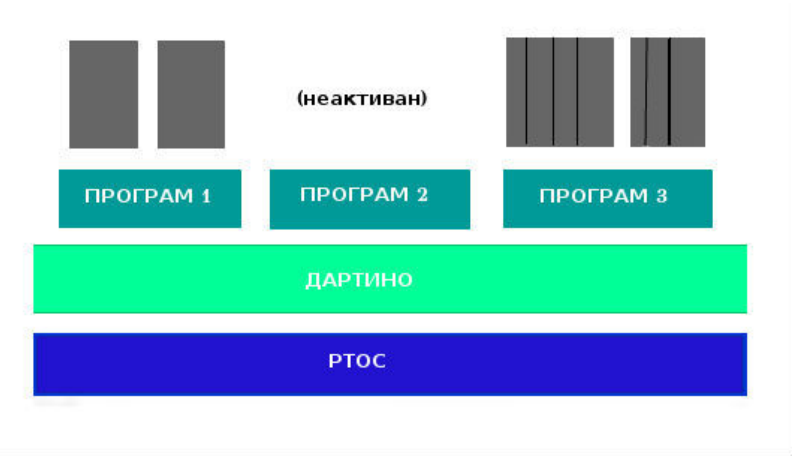
4.2 Софтверска архитектура система

Доњи слој софтверске архитектуре система са уграђеним рачунаром чини традиционални оперативни у реалном времену (нпр. Free RTOS). Изнад тога се налази Дартино окружење за извршавање, које може извршавати више програма, независно, и они не морају знати ништа једни о другима. Ово се разликује од уобичајених система који раде на оперативном систему уграђеног рачунара, и који обично не могу да извршавају више независних компоненти, на независан начин. За сваки програм можемо имати више процеса, више инстанци програма, или делова програма, који се извршавају конкурентно [23].

На слици 4.2 је приказана ситуација када имамо 3 програма. Програм2 се тренутно не извршава, за Програм1 су покренуте две инстанце, док се Програм3 извршава у 7 нити.

4.3 Процеси и врсте процеса

Дартино виртуелна машина има подршку за различите врсте процеса. Основни процеси су лаки процеси, али поред њих ту су влакна и изолате [23, 15, 4].



Слика 4.2: Приказ софтверске архитектуре система, на примеру случаја када имамо више програма који се извршају конкурентно.

Лаки процеси

За сваки процес нам је потребно мало меморије, неколико стотина бајтова. Сав кôд и подаци у меморији се деле, па меморију заузимају само подаци који представљају процес. На овај начин ради већина оперативних система, али је то новина за оперативне системе за уграђене рачунаре.

Процес се може блокирати, при чему се не заузимају ресурси, већ се главна нит враћа систему. Разлог за то је што у системима са уграђеним рачунаром обично имамо мали фиксирани број нити, па не желимо да их блокирамо.

Уколико процесор има више језгара, постоји подршка за паралелно извршавање нити које одговарају језгрима (енгл. *native thread*). На сваком језгру се процеси извршавају конкурентно, а Дартино веома добро распоређује велики број процеса на много мањи број језгара [23].

```
main() {
  for (int i = 0; i < 4000; i++) {
    Process.spawn(hello(i));
  }
}
void hello(n) {
  print("Hello from ${i}");
}
```

Код 17: Креирање паралелних процеса методом `Process.spawn`.

Кôд 17, написан у програмском језику Дарт, приказује покретање 4000 неза-

висних процеса за исписивање поздравне поруке, који ће бити распоређени на одређени број језгара и извршавати се паралелно. Сваки процес има своју хип меморију, а са осталим процесима може комуницирати само слањем порука. *Process.spawn* креира нови процес и покреће га [15].

```
var server = new ServerSocket("127.0.0.1", 0);
while(true) {
  server.spawnAccept((Socket socket){
        //izvrsava se u novom procesu
        socket.send(UTF.encode("Hello"));
        socket.close();
  })
}
```

Код 18: Манипулисање надолазећим конекцијама методом `spawnAccept`.

У коду 18, написаном у програмском језику Дарт, креира се *ServerSocket* који представља сервер, затим се помоћу метода *spawnAccept* креира нови процес. У оквиру тог процеса се чека да клијент успостави конекцију са сервером, и самим тим блокира даље извршавање док се конекција не успостави. При успостављању конекције, клијент (*socket*) шаље серверу (*server*) поздравну поруку [23]. Издвајање комуникације у нови процес је потребно вршити увек када се врши неко комплексније израчунавање пре слања поруке.

Комуникација између процеса се обавља слањем порука, помоћу канала и портова. Канал представља један ред порука. Порт представља могућност слања поруке каналу, и без приступа одређеном порту, не може се слати порука каналу који чека на другој страни порта [23].

```
final channel = new Channel();
final port = new Port(channel);
Process.spawn(() {
        int i=0;
        while(i < 50) {
                port.send(i++);
        }
});
while(true) {
        print(channel.receive());
}
```

Код 19: Комуникација процеса слањем порука на одређени канал.

Кôд 19, написан у програмском језику Дарт, приказује један процес који шаље поруке, и функцију *print* која блокира извршавање све док не прими следећу поруку и испише је.

Процеси деле само објекте који се не мењају, и то на следећи начин [23]:

1. Сваки непроменљиви објекат се може слати као порука, без копирања
2. Више процеса могу користити(читати) исти објекат истовремено
3. Нема потребе за експлицитним примитивама за синхронизацију, јер мењање објеката није омогућено

Изолате

Изолате су независни воркери, свака изолата има своју хип меморију, и за разлику од нити нема дељења меморије између изолата. Изолате комуницирају само преко порука. Могу се паузирати, наставити и убити [15]. Детаљније су описане у поглављу 3.7.

Влакна

Влакна (енгл. *Fibers*) представљају нити извршавања у оквиру једног процеса, које не деле непроменљиву меморију. Међутим, могу делити екстерну меморију (нпр. помоћу класе ForeignMemory [1]), али се у том случају мора ручно водити рачуна о синхронизацији. Извршавају се конкурентно, тако да нема паралелизма. Омогућују да више независних нити једног процеса могу чекати на исту ствар(блокирати) независно једна од друге. На овај начин се могу имплементирати више одвојених контрола тока извршавања програма [4].

Кôд 20 приказује употребу два влакна у програмском језику Дарт. У оквиру једног се чека на промену вредности у сензору за температуру, док се у оквиру другог чека на промену вредности у сензору за влажност ваздуха [23]. Функција *publishOnChange* блокира извршавање док се у сензору за који је позвана не деси промена вредности. На овај начин је омогућено да се у оквиру једног процеса чека на више ствари, и нема паралелизма, већ се уколико дође до блокирања у једном влакну, прелази на извршавање следећег који је спреман за извршавање.

[1] https://dartino.github.io/api/dart-dartino.ffi/ForeignMemory-class.html.

```
void publishOnChange(Socket socket, String property, Channel input){
            int last = 0;
            while(true){
              int current = input.receive();
              if(current != last)
                socket.send(UTF.encode('("$property": $current)'));
              last = current;
            }
}

Fiber.fork(() => publishOnChange(server, "temperature", tempSensor));
Fiber.fork(() => publishOnChange(server,"humidity", humSensor));
```

Код 20: Пример употребе два влакна која чекају на функцију `publishOnChange`, при чему не зависе једно од другог.

4.4 Корутине

Дартино има уграђену подршку за корутине. Корутине су објекти налик на функције, које могу вратити више вредности. Када корутина врати вредност, зауставља се на тренутној позицији, а њен стек извршавања се чува за касније покретање. Када врати последњу вредност, сматра се завршеном, и више се не може покретати [4].

Кодом 21 дат је пример употребе корутина у програмском језику Дарт,. Функција *Expect.equals* пореди две вредности које су јој прослеђене као аргументи, и у случају неједнакости враћа грешку. При првом позиву корутине *co(1)*, резултат ће бити 4. Овај резултат је враћен позивом *Coroutine.yield(4)*. Након позива yield, корутина ће бити суспендована. Другим позивом корутине *co(2)*, њено извршавање ће се наставити тамо где је стала. Позив *Expect.equals(2, 2)* ће вратити тачно, и извршавање корутине ће се завршити. Пошто није наведена повратна вредност, резултат ће бити *null*. Након тога се корутина сматра завршеном и не може се наставити.

4.5 Динамичко отпремање метода

Динамичко отпремање метода (енгл. *Dynamic dispatch*) се односи на позивање полиморфних метода класа, при чему се који метод треба позвати разрешава у фази извршавања, а не у фази превођења. Прави се табела отпремања, која се имплементира као низ, у који се сваки метод, сваке класе, смешта на

```
var co = new Coroutine((x) {
    Expect.equals(1, x);
    Expect.equals(2, Coroutine.yield(4));
});

Expect.equals(4, co(1));
Expect.isTrue(co.isSuspended);
Expect.isNull(co(2));
Expect.isTrue(co.isDone);
```

Код 21: Употреба корутина.

своју позицију. Позиција метода у табели се рачуна помоћу редног броја класе и позиције метода. У току извршавања, на основу ове табеле се одлучују који метод треба позвати. Оно што је неопходно при добијању одређеног метода из табеле, је провера да ли је резултат заправо оно што нам треба. Уколико није, значи да тражени метод одређене класе не постоји. Загарантовано је константно време отпремања, и табела отпремања се израчунава пре извршавања, па представља део апликације [23].

```
class A {
  metod1();
}
class B extends A {
  metod2();
}
class C extends B {
  metod1();
}
class D extends A {
  metod3();
}
```

Код 22: Пример хијерархије класа помоћу ког се илуструје генерисање табеле отпремања метода.

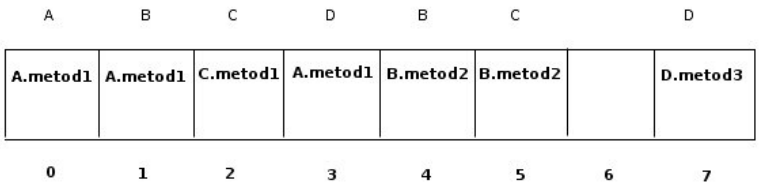
У коду 22 је представљено неколико класа којима се преклапају методи. Процес генерисања табеле отпремања је следећи: Класама редом придружимо бројеве: 0, 1, 2 и 3. Методима придружимо померај на мало компликованији начин: методу *metod1*, који је дефинисан у највећем броју класа, придружимо померај 0, и он се смешта на почетак табеле. Методу *metod2* придружимо померај 3, и методу *metod3* придружимо 4. Померају морају бити јединствени

бројеви, како не би дошло до преклапања метода [23].

У наставку је пример рачунања позиције метода у табели отпремања:

- Позиција метода *metod1* за класу A: 0 (редни број класе) + 0 (померај метода) = 0
- Позиција метода *metod1* за класу B: 1 (редни број класе) + 0 (померај метода) = 1
- Позиција метода *metod1* за класу C: 2 (редни број класе) + 0 (померај метода) = 2
- Позиција метода *metod1* за класу D: 3 (редни број класе) + 0 (померај метода) = 3
- Позиција метода *metod2* за класу B: 1 (редни број класе) + 3 (померај метода) = 4
- Позиција метода *metod2* за класу C: 2 (редни број класе) + 3 (померај метода) = 5
- Позиција метода *metod3* за класу D: 3 (редни број класе) + 4 (померај метода) = 7

Пример табеле отпремања за ове класе приказан је на слици 4.3. Приметимо да се могу појавити празне позиције у табели.



Слика 4.3: Табела отпремања за класе у примеру 22.

4.6 Најзначајније библиотеке

Ради прилагођавања програмског језика Дарт специфичностима микроконтролера, развијена је нова библиотека *„dartino”*. Она се састоји из више мањих библиотека, а неке од најзначајних су описане у наставку [8].

file - Интерфејс за рад са датотекама. Подржано само када се Дартино извршава на POSIX платформи.

ffi - „foreign function interface" библиотека која омогућава да се из Дарт кода позивају функције дефинисане у неком другом програмском језику, нпр. функције програмског језика C.

http - Имплементација HTTP (скраћено од енгл. *HyperText Transfer Protocol*) клијента.

mbedtls - TLS (скраћено од енгл. *Transport Layer Security*) подршка, базирана на mbedtls[2] [12]. Ово се може користити на исти начин као нормални сокет (и да се прослеђује HTTP пакету).

mqtt - MQTT клијентска библиотека за MQTT (скраћено од енгл. *MQ Telemetry Transport*) протокол, IoT протокол за размену порука заснован на објављивању/претплати [13].

os - Приступ оперативном систему. Подржано када се Дартино извршава на POSIX платформи.

socket - Дартино имплементација TCP (скраћено од енгл. *Transmission Control Protocol*) и UDP (скраћено од енгл. *User Datagram Protocol*) сокета.

stm32 - Подршка за STM32[3] плоче.

4.7 Покретање Дарт програма унутар виртуелне машине

Примарни начин за комуникацију са Дартино виртуелном машином је преко команде *dartino*. Ова команда комуницира са dart процесом, који се извршава у позадини и не захтева интеракцију са корисником (енгл. *persistent process*). Дарт процес омогућава компилацију, тако што преводи Дарт код у Дартино бајткод, док Дартино виртуелна машина (*dartino-vm*) извршава и дебагује програме тако што Дартино бајткод преводи у извршни код.

[2]Имплементација TLS и SSL сигурносних протокола и одговарајућих алгоритама криптовања.

[3]Фамилија микроконтролера базираних на 32-битним ARM процесорима.

Нешто више о извршним фајловима које се користе при покретању програма [7]:

dartino Ово је мали извршни фајл који преко сокета прослеђује стандардне улазно/излазне сигнале процесу dart. Служи за покретање JIT компилатора који је написан у Дарту и извршава се у оквиру Дарт виртуелне машине

dartino-vm Виртуелна машина која покреће програм из бајткода.

dart dart је процес који се у позадини извршава и не захтева интеракцију са корисником. Овај процес се састоји од главне нити и неколико воркера. Главна нит ослушкује надолазеће конекције из *dartino* команде, и може да користи воркере да обави тражени задатак. Основна улога овог процеса је покретање Дарт компилатора. Овај процес се извршава на Дарт виртуелној машини.

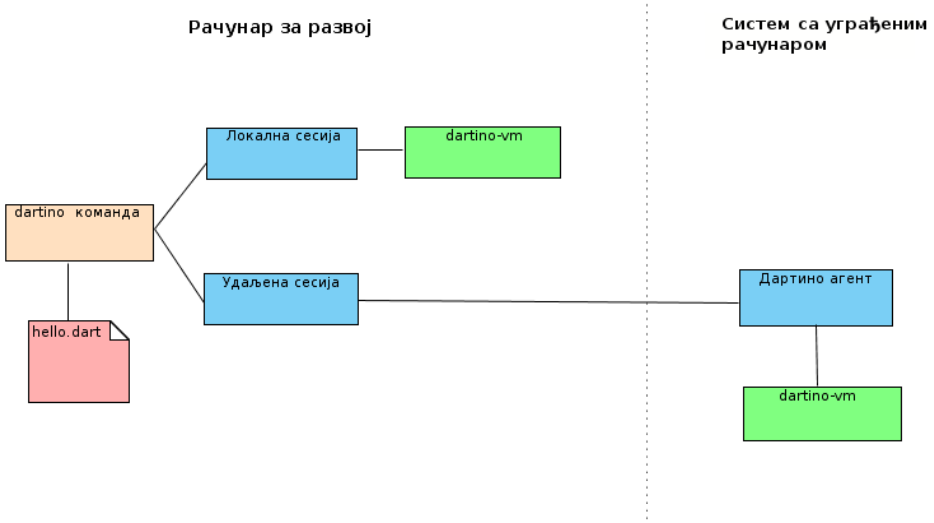
```
main(){

int a = 5, b = 6;
int c = a + b;

print('Zbir je ${c}');
}

load literal 5          // 5
load literal 6          // 6
load local 1            // a
load local 1            // b
invoke unfold add 27649 // a + b
load const @0           // 'Zbir je ${c}'
load local 1            // c
invoke unfold method 69632
invoke unfold add 27649
invoke static 1
pop
drop 3
return null
method end 72
```

Код 23: Пример програма у прог. језику Дарт који сабира два броја и исписује резултат, и одговарајућег бајткода.

Шта се заправо дешава када се покрене команда *dartino run hello.dart*? Дартино је подразумевано повезан на локалну сесију, која је повезана на локалну виртуелну машину која ради на нашем рачунару. *dartino* преводи кôд до бајткода, помоћу Дарт компилатора, а онда га прослеђује *dartino-vm* која га извршава. Дартино виртуелна машина враћа резултат назад Дартину, и он га приказује.

Кодом 23 представљен је пример програма који сабира две променљиве које имају целобројне вредности и исписује резултат, и одговарајући бајткод који се добија као резултат JIT компилације у оквиру Дарт виртуелне машине.

Дартино подржава више сесија. Свака сесија може бити придружена различитој Дартино виртуелној машини, што омогућује кориснику да тестира више различитих уређаја истовремено. Тренутно је руковање сесијама експлицитно [23, 7].



Слика 4.4: Процес извршавања програма.

Покретање програма у оквиру одређене сесије је омогућено на следећи начин [6]:

./dartino create session my_session

Креирање нове сесије „my_session".

Након тога морамо да покренемо дартино виртуелну машину:

./dartino-vm

Након покретања дартино виртуелне машине се исписује порука облика „Waiting

for compiler on 127.0.0.1:61745", при чему број 61745 представља рандом генерисани порт.

Да би се прикачили на дати сокет, потребно је да отворимо нови терминал и у њему покренемо dartino команду са наредним параметрима:

./dartino attach tcp_socket 127.0.0.1:61745 in session my_session

Затим у оквиру те сесије можемо да покренемо жељени програм на следећи начин:

./dartino run hello.dart in session my_session

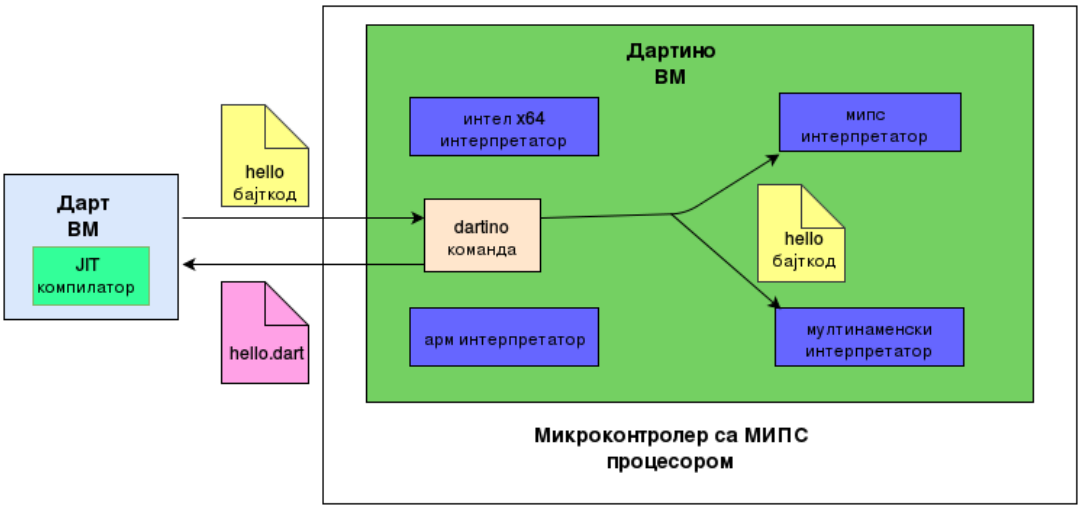
Сесија се прекида помоћу следеће команде:

./dartino x-end session my_session

Глава 5

Имплементација интерпретатора за платформу МИПС

Као што је описано у поглављу 4.7 Дартино виртуелна машина представља окружење за извршавање програма написаних у програмском језику Дарт. Компилација Дарт кода у Дартино бајткод није део Дартино виртуелне машине, већ се користи Дарт виртуелна машина која садржи JIT компилатор. Дарт вм преводи улазни код у бајткод, који затим прослеђује Дартино вм која га преводи у извршни код, што је приказано на слици 5.1. У оквиру овог рада урађена је подршка за превођење бајткода у извршни код за платформу МИПС, и у овој глави је описан поступак имплементације.



Слика 5.1: Процес превођења Дарт кода у извршни код.

У оквиру Дартино вм постојао је мултинаменски интерпретатор, који је не-

зависан од платформе, написан у програмском језику C++. Такође постојали су интерпретатори за платформе Интел и ARM. Мултинаменски интерпретатор је био доста спорији од интерпретатора за специфичне архитектуре, али да би се Дартино вм користила на МИПС платформи, морао се користити тај интерпретатор. Како конфигурација мултинаменског интерпретатора за МИПС није постојала, најпре је требало то урадити, и поступак за то је описан у оквиру овог рада. Идеја је била развити интерпретатор за МИПС, који ће имати боље перформансе од мултинаменског. Конфигурисање мултинаменског интерпретатора било је важно како би се у току развоја МИПС интерпретатор могао тестирати, и такође, да би се могле поредити перформансе. Конфигурасање је у овом раду описано како би се могли репродуковати извршени тестови и поређења, пошто процес није једноставан, а није ни саставни део пројекта, зато што је развој мултинаменског интерпретатора заустављен у раној фази, због чега се у оквиру тренутне верзије Дартино виртуелне машине не може користити.

У овој глави је описана имплементација интерпретатора за платформу МИПС, кроз делове кода и одређене тест примере који су развијани у току имплементације. У поглављу 5.1 описан је начин за конфигурисање мултинаменског интерпретатора за платформу МИПС, док је у поглављу 5.2 описан поступак имлементације МИПС интерпретатора. У поглављу 5.3 описан је имплементирани систем за дебаговање. У поглављу 5.4 описани су тест примери који су довели до уочавања грешака у окружењу, док су у поглављу 5.5 приказани резултати тестирања на постојећем скупу тестова. У поглављу 5.6 приказан је резултат поређења перформанси МИПС интерпретатора са мултинаменским и ARM интерпретатором.

5.1 Конфигурација мултинаменског интерпретатора

У почетку развоја, постојао је мултинаменски интерпретатор, који не зависи од архитектуре [10]. Било је потребно конфигурисати Дартино за платформу МИПС, и на тај начин обезбедити превођење за МИПС платформу.

У наставку ће бити описан процес превођења Дартино виртуелне машине, и шта све обухвата конфигурисање за МИПС.

Превођење се врши помоћу *ninja* система за превођење. Ово је мали, веома брз систем за превођење, који се обично не пише ручно. Како је једини циљ при

имплементацији овог система била брзина, он је читљив, али се препоручује коришћење у комбинацији са другим мета системима за превођење [14]. У оквиру Дартино пројекта коришћен је *gyp* мета систем [9]. Превођење се врши командом *ninja* која у оквиру текућег директоријума тражи датотеку *build.ninja*. Ова датотека се генерише на основу одговарајућих *.gyp* и *.gypi* датотека.

У наставку су описане датотеке које су од значаја приликом конфигурације:

- **default_targets.gypi** - Датотека у којој се дефинишу подржане архитектуре.
- **common.gypi** - Датотека у којој се дефинише конфигурација која је потребна да би се извршило превођење.
- **cc_wrapper.py** - Датотека у којој се покреће одговарајући GCC компилатор.
- **cxx_wrapper.py** - Датотека у којој се покреће одговарајући G++ компилатор.
- **build.ninja** - Датотека која се генерише на основу *.gyp* и *.gypi* датотека. Она се извршава командом *ninja*.

У датотеци *default_targets.gypi* дефинишу се подржане архитектуре. За МИПС је потребно додати *DebugXMIPS*, при чему X-означава се врши унакрсна компилација (енгл. *cross compilation*). Овде се за сваку од архитектура увлачи конфигурација из датотеке *common.gypi*. У датотеци *common.gypi* дефинише се у којим датотекама се врши покретање одређене врсте компилатора GCC и G++. То су датотеке *cc_wrapper.py* и *cxx_wrapper.py*. Поред тога, у *common.gypi* су дефинисани флегови који се прослеђују компилатору GCC, који су независни од архитектуре. Такође, овде се дефинише и врста оперативног система. Затим се за сваку од архитектура дефинишу одговарајући флегови, као и путање до библиотека које се користе у фази линковања. За МИПС је потребно навести *'-mips32r2'* који означава циљну архитектуру и *'-EL'* што указује да је запис података у меморији такав да је на нижој адреси нижи бит меморијске речи (енгл. *little endian*). Поред тога, потребно је навести путање до библиотека које се налазе у оквиру одређеног ланца алата за МИПС (енгл. *mips toolchain*). Такође, за сваку од архитектура, дефинишу се „лажни" макрои који се користе у датотекама *cc_wrapper.py* и *cxx_wrapper.py*, како би се на основу

њих изабрао одређени скуп алата који се користи при превођењу. Ови макрои су „лажни" јер се прослеђују заједно са флеговима који се користе у фази линковања, али се у оквиру поменутих датотека избаце из тог скупа, пре него што дођу до линкера. За платформу МИПС, у оквиру датотека *cc_ wrapper.py* и *cxx_ wrapper.py* треба додати да се покрећу GCC и G++ из одређеног ланца алата за МИПС.

На овај начин је омогућено покретање команде *ninja* која врши превођење и креира одговарајуће излазне (енгл. *out*) директоријуме за сваку од подржаних архитектура. Унутар сваког од излазних директоријума, генерише се датотека *build.ninja* помоћу које се врши превођење за жељену архитектуру. Превођење за МИПС врши се помоћу команде *ninja -C out/DebugXMIPS*.

Одговарајуће *build.ninja* датотеке генеришу се на основу више различитих *.gyp* и *.gypi* датотека, али су овде наведене само оне које су значајне при конфигурацији за жељену платформу .

Како би се могао покренути програм написан у Дарту, на платформи МИПС, потребно је обезбедити да се МИПС извршне датотеке покрећу помоћу QEMU (скраћено од енгл. *Quick Emulator*) емулатора у корисничком режиму[1]. Да би се то вршило аутоматски, непходно је линукс команди *update-binfmts* проследити вредности које одговарају МИПС ELF (скраћено од енгл. *format of Executable and Linking Format* [2]) датотекама, и путању до скрипте која покреће QEMU емулатор. Верзија 2.1.43 линукс кернела, и све након ње, садрже *binfmt_ misc* модул. Овај модул омогућује да системски администратор може регистровати који се интерпретатор треба користити за различите бинарне формате, на основу магичних вредности и вредности маске (енгл. *magic and mask*), и да при регистровању датотеке која има те вредности, позове одговарајући интерпретатор. Команда *update-binfmts* управља базом ових интерпретатора. Команда са одговарајућим аргументима је представљена кодом 24, док је скрипта која покреће QEMU представљена кодом 25.

На овај начин је омогућено и тестирање мултинаменског интерпретатора на платформи МИПС, помоћу постојећег скупа тестова. Конфигурисање овог интерпретатора за МИПС било је потребно како би се током развоја могла проверавати исправност интерпретатора који је специфичан за МИПС, а поред

[1] У овом режиму QEMU може покретати процесе преведене за једну врсту процесора на другој врсти процесора.

[2] Бинарни формат извршних датотека на Линукс оперативном систему.

```
sudo update-binfmts --install qemu-mipsel /<skripta>
--magic '\x7fELF\x01\x01\x01\x00\x00\x00\x00
        \x00\x00\x00\x00\x00\x02\x00\x08\x00'
--mask '\xff\xff\xff\xff\xff\xff\xff\x00\xfe\xff
        \xff\xff\xff\xff\xff\xfe\xff\xff\xff'
--credentials yes --package qemu-user-static
```

Код 24: Команда за аутоматско покретање МИПС извршних датотека помоћу QEMU-а. <skripta> представља путању до скрипте 25.

```
#!/bin/bash
qemu=/<qemu>/mipsel-linux-user/qemu-mipsel
exec $qemu $*
```

Код 25: Скрипта у којој се покреће QEMU емулатор. <qemu> представља путању до директоријума у ком се налази QEMU емулатор.

тога, и да би се поредиле перформансе.

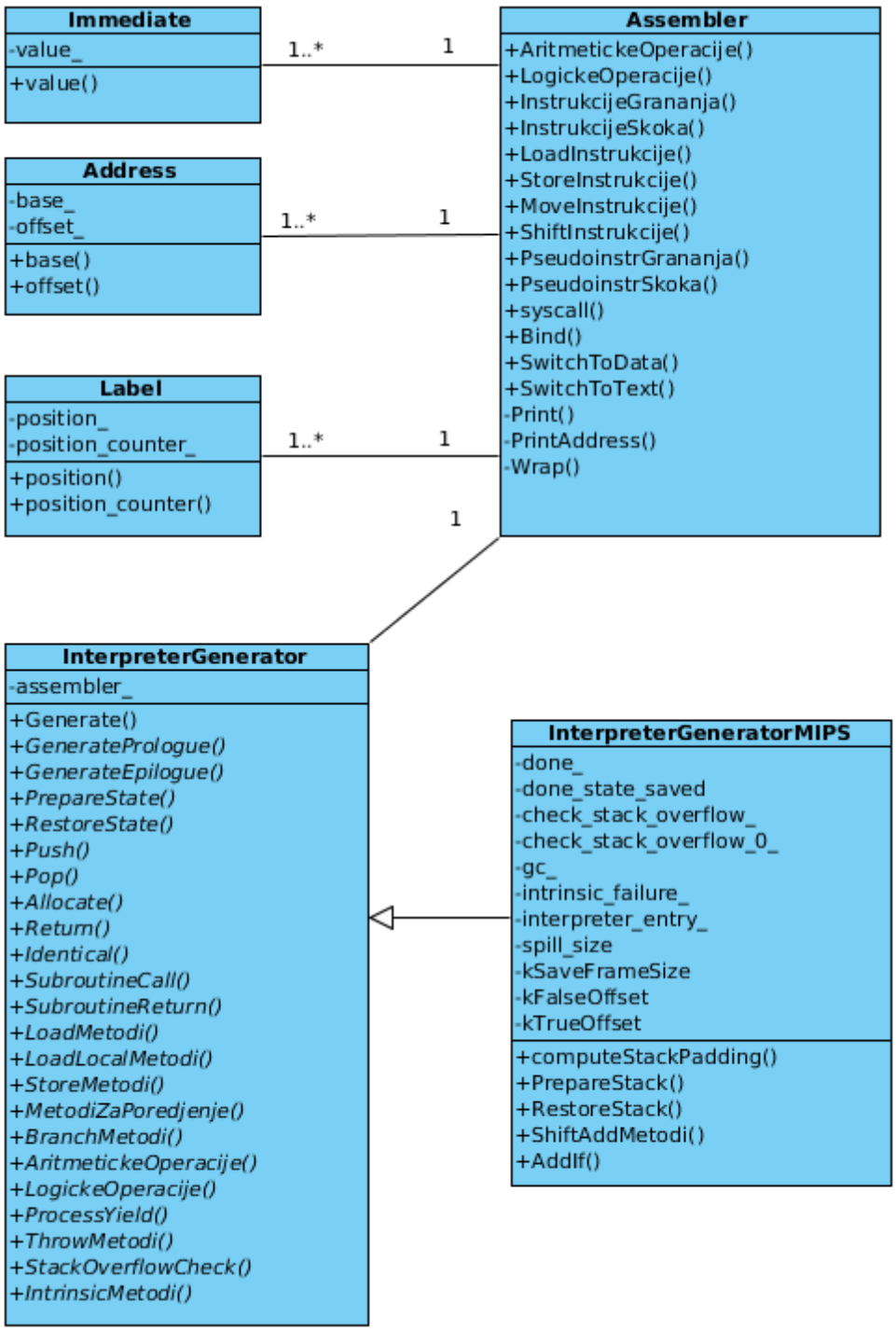
5.2 Имплементација у програмском језику C++

Имплементација је рађена на рачунару са Интел x86-64 процесором, унакрсном компилацијом, при чему је циљна архитектура МИПС32Р2 (енгл. *mips32r2*, скраћено од mips32 release 2). Имплементирани интерпретатор генерише асемблерску датотеку чијим превођењем добијамо програм који извршава бајткод на МИПС архитектури.

Имплементација је рађена у програмском језику C++. Како би се омогућило генерисање МИПС асемблерског кода, направљене су датотеке *assembler_ mips.h*, *assembler_ mips.cc* и *assembler_ mips_ linux.cc* које описују МИПС асемблерски језик [1, 2, 3]. У датотеци *assembler_ mips.h* дефинисана је енумерација регистра која обухвата МИПС регистре. Више о МИПС регистрима речено је у поглављу 2.4.

На слици 5.2 приказан је дијаграм класа које су имплементиране, и које ће у овом поглављу бити описане. Како класе садрже велики број метода, на дијаграму су приказане групе метода, као и неки најназначајнији методи који нису сврстани ни у једну групу.

Направљене су класе *Label*, *Immediate* и *Address* које су представљене кодом 26. Лабела представља линију у оквиру асемблерског фајла на коју се може скочити. Класа *Label* која садржи два поља: *position* - представља редни број



Слика 5.2: Дијаграм имплементираних класа.

лабеле и *position_ counter* - заједничка променљива за све инстанце класе *Label*, на основу које се додељује позиција лабеле. Инстанца класе *Immediate* пред-

ставља константу вредност целобројног типа, док инстанца класе *Address* представља меморијску адресу, која је на платформи МИПС32Р2 32-битна. Адреса се рачуна као збир адресе у базном регистру и помераја. Базни регистар је регистар опште намене који садржи 32-битну адресу.

```
class Label {
 public:
  Label() : position_(-1) {}

  int position() {
    if (position_ == -1) position_ = position_counter_++;
    return position_;
  }

 private:
  int position_;
  static int position_counter_;
};
class Immediate {
 public:
  explicit Immediate(int32_t value) : value_(value) {}
  int32_t value() const { return value_; }

 private:
  const int32_t value_;
};
class Address {
 public:
  Address(Register base, int32_t offset)
      : base_(base), offset_(offset) {}

  Register base() const { return base_; }
  int32_t offset() const { return offset_; }

 private:
  const Register base_;
  const int32_t offset_;
};
```

Код 26: Класе помоћу којих се генеришу лабеле, константе и адресе у МИПС асемблерском језику.

Дефинисани су макрои за МИПС инструкције са одговарајућим бројем ар-

гумената. Помоћу ових макроа, у оквиру класе *Assembler* дефинисане су све МИПС инструкције које су потребне при интерпретацији.

```
#define INSTRUCTION_3(name, format, t0, t1, t2)  \
  void name(t0 a0, t1 a1, t2 a2) {               \
    Print(format, Wrap(a0), Wrap(a1), Wrap(a2)); \
  }
```

Код 27: Макро за генерисање инструкције која има 3 аргумента.

У коду 27 представљен је пример макроа за генерисање МИПС асемблерских инструкција које имају 3 аргумента. Аналогно томе, постоје макрои за генерисање инструкција са 1, 2 и 4 аргумента. Инструкције које имају 4 аргумента су заправо инструкције гранања, при чему први аргумент представља услов гранања. Тај први аргумент означава која инструкција гранања је у питању, док су остала 3 аргумента стварни аргументи инструкције. У оквиру класе *Assembler* имплементирани су методи *SwitchToText()* и *SwitchToData()* који постављају директиве за сегмент података, односно сегмент кода. Поред тога, класа *Assembler* садржи и методе који представљају „псеудоинструкције" генерисане ради прегледнијег кода. Нпр. метод *B* представља „псеудоинструкцију" која врши гранање и након тога попуњава слот закашњења инструкцијом *nop*. У оквиру ове класе имплементиран је и метод *Print* за уписивање асемблерских инструкција у фајл. Дефинисан је начин записивања регистара, лабела и адреса у МИПС асемблеру.

У коду 28 представљен је део метода *Print* који се односи на начин записивања лабеле унутар асемблерског фајла. Пре позиције додаје се префикс „.L" који означава да је у питању лабела. Такође, представљен је и део метода који се односи на начин записивања регистра унутар асемблерског фајла. Регистар се означава префиксом „$", након чега следи ознака регистра. Поред тога, представљен је и метод за записивање адресе унутар асемблерског фајла. Адреса се записује као померај иза ког следи у загради запис базног регистра. На овај начин је омогућено генерисање МИПС асемблерског кода.

У оквиру интерпретатора имплементиране су класе *InterpreterGenerator* и *InterpreterGeneratorMIPS*. Класа *InterpreterGenerator* је апстрактна. У оквиру ње је имплементиран метод *Generate* који је одговоран за генерисање асемблерског кода. Класа као једини приватни податак чува показивач на инстанцу

```
case 'l': {
  Label* label = va_arg(arguments, Label*);
  printf(".L%d", label->position());
  break;
}

case 'r': {
  Register reg = static_cast<Register>(va_arg(arguments, int));
  printf("$%s", ToString(reg));
  break;
 }

void Assembler::PrintAddress(const Address* address) {
  printf("\%d($%s)", address->offset(), ToString(address->base()));
}
```

Код 28: Пример записивања лабеле у МИПС асемблерском језику.

класе *Assembler*. Помоћу овог показивача, приступа се методима који су дефинисани у класи *Assembler* (инструкцијама, псеудоинструкцијама и осталим методима). Како би се олакшао приступ овим методима, дефинисан је макро `__` који представља показивач на инстанцу класе *Assembler*. То је представљено кодом 29.

```
#define __ assembler()->
```

Код 29: Макро за приступање методима класе *Assembler*, при чему метод assembler() инстанцу те класе.

Дефинише се макро *V(name, branching, format, size, stack_diff, print)* који се користи за генерисање виртуелних апстрактних метода који имају потпис *void Do##name()*, користећи макро *BYTECODES_DO(V)* који је дефинисан у датотеци *bytecodes.h*. На пример, један од тих метода је *DoLoadLiteralNull* који поставља на стек *Null* објекат. Слично томе дефинише се и макро *V(name)*, помоћу ког се генеришу уграђени (енгл. *intrinsic*) методи. Они представљају функције којима се може приступити из C кода, и које компилатор директно имплементира у коду, без стварног позивања ових функција.

У оквиру метода *Generate* позивају се виртуелни апстрактни методи *GeneratePrologue* и *GenerateEpilogue*. Ови методи се односе на МИПС позивну конвенцију, која је описана у поглављу 2.10. Функција *GeneratePrologue* служи

за генерисање асемблерског кода који ће се извршити приликом позива сваке функције. Мора се сачувати садржај регистара *$s0-$s7* на стеку, и такође садржај регистра *$ra*, у ком се налази адреса повратка функције. По повратку из неке функције, позива се функција *GenerateEpilogue* у којој се скида са стека садржај регистара *$s0-s7* и регистра *$ra*.

Поред тога, у оквиру метода *Generate*, помоћу макроа *V(name, branching, format, size, stack_diff, print)* позивају се већ наведени методи облика *void Do##name()*. Пре позива сваког од метода, позива се метод који генерише одговарајућу лабелу у оквиру асемблерске датотеке, како би се на део кода који генерише тај метод могло скочити. Такође, у оквиру метода *Generate*, у сектору података се декларишу лабеле на које се може скочити.

Класа *InterpreterGeneratorMIPS* наслеђује класу *InterpreterGenerator* и имплементира све њене виртуелне методе. Од приватних података, ова класа садржи лабеле помоћу којих се означавају делови кода на које се може скакати из различитих метода. Неке од тих лабела су *check_stack_overflow*, помоћу које се скаче на део кода који врши проверу прекорачења на стеку, и *done_* лабела која се користи за скакање на део кода који сачува садржаје регистара на стеку.

Дефинише се макро *INVOKE_BUILTIN(kind)*, помоћу ког се декларишу методи који имају исти потпис, облика *DoInvoke##kind()* и облика *DoInvoke##kind##Unfold()*. Ови методи позивају методе облика *Invoke##kind()* прослеђујући им лабелу на којој се врши опоравак у случају прекорачења. Методе облика *Invoke##kind()* чине методи који имплементирају поређење (*DoInvokeEq*, *DoInvokeEq*, *DoInvokeLt*, *DoInvokeLe*, *DoInvokeGt*, *DoInvokeGe* и *DoInvokeCompare*), методи који имплементирају основне аритметичке операције (*DoInvokeAdd*, *DoInvokeSub*, *DoInvokeMul*, *DoInvokeDiv*), и методи који имплементирају битовске операторе (*DoInvokeBitNot*, *DoInvokeAnd*, *DoInvokeOr*, *DoInvokeXor*, *DoInvokeShl* и *DoInvokeShr*).

Кодом 30 представљен је метод *InvokeAdd*, који позива метод (*DoInvokeAdd*) прослеђујући му лабелу на део кода који врши опоравак од прекорачења. Овај метод сабира два броја, при чему се у случају прекорачења скаче на део кода који врши опоравак.

Поред тога, ова класа имплементира и све остале виртуелне апстрактне методе своје надкласе. Међу тим методама се налазе методи облика *LoadLocalN*, који постављају *N*-ту реч са стека као аргумент функције. Слично томе, метод *StoreLocal* поставља вредност на стек. Методе за рад са стеком чине *Push* за постављање садржаја на стек, *Pop* за скидање садржаја са стека, *Drop* и

```
void InterpreterGeneratorMIPS::InvokeAdd(const char* fallback) {
  Label no_overflow;
  LoadLocal(A0, 1);
  __ andi(T0, A0, Immediate(Smi::kTagMask));
  __ B(NEQ, T0, ZR, fallback);
  LoadLocal(A1, 0);
  __ andi(T1, A1, Immediate(Smi::kTagMask));
  __ B(NEQ, T1, ZR, fallback);

  __ xor_(T1, A0, A1);
  __ b(LT, T1, ZR, &no_overflow);
  __ addu(T0, A0, A1);  // Delay-slot.
  __ xor_(T1, T0, A0);
  __ b(LT, T1, ZR, fallback);
  __ Bind(&no_overflow);
  __ move(A0, T0);  // Delay-slot.
  DropNAndSetTop(1, A0);
  Dispatch(kInvokeAddLength);
}
```

Код 30: Функција у МИПС интерпретатору која сабира две целобројне вредности.

DropNAndSetTop за одбацивање N речи са стека. Такође, имплементиране су методе за рад са показивачем на стек оквир: учитавање показивача на стек оквир у регистар (*LoadFramePointer*), постављање показивача на стек оквир (*StoreFramePointer*) и чување показивача на стеку (*PushFramePointer*). Имплементиран је и метод за повратак из функције (*Return*), затим методи који се односе на гранање (*DoBranchBack*, *DoBranchBackIfTrue*, *DoBranchBackIfFalse* и други), методи за избацивање изузетака (*DoThrow* и *DoThrowAfterSaveState*), за рад са корутинама, за поређење објеката, и многи други.

При скоку на функцију која се налази ван интерпретатора, мора се позвати метод за поравнање стека. Овај метод је представљен кодом 31. На основу позивне конвенције, описане у поглављу 2.10, неопходно је резервисати 4 места на стеку за аргументе функције (уколико функција коју позивамо позива неку другу функцију). Осим тога, неопходно је сачувати садржај регистра *$gp*. Стек увек мора бити поравнат на 8 (адреса стека мора бити дељива са 8), па се због тога сачува 6 речи уместо 5.

По повратку из неке спољне функције, позива се метод *RestoreStack*, која ради инверзну операцију метода *PrepareStack*. Овај метод је представљен кодом

```
void InterpreterGeneratorMIPS::PrepareStack() {
  __ addiu(SP, SP, Immediate(-6 * kWordSize));
  __ sw(GP, Address(SP, 5 * kWordSize));
}
```

Код 31: Пример метода за поравнање стека, која се позива пре скока на неку спољну функцију.

32.

```
void InterpreterGeneratorMIPS::RestoreStack() {
  __ lw(GP, Address(SP, 5 * kWordSize));
  __ addiu(SP, SP, Immediate(6 * kWordSize));
}
```

Код 32: Пример метода за поравнање стека, која се позива по повратку из неке спољне функцију.

Након дефиниције класе, позива се макро *GENERATE(, Interpret)*. Овај макро је дефинисан у датотеци *generator.h*, и он је приказан кодом 33 .Помоћу њега се декларише функција чије се име добија надовезивањем аргумената макроа на реч *Generate*. Како овај позив макроа нема први аргумент, на реч *Generate* ће се надовезати само други аргумент, што је *Interpret*. На овај начин декларише се функција која позива метод *Generate* класе *InterpreterGeneratorMIPS*, помоћу које се генерише асемблерски код.

```
#define GENERATE(p, n)                                          \
  static void Generate##p##n(Assembler* assembler);             \
  static const Generator kRegister##p##n(Generate##p##n, #p #n); \
  static void Generate##p##n(Assembler* assembler)
```

Код 33: Макро помоћу ког се генерише асемблерски код.

Први програм који је успешно извршен помоћу МИПС интерпретатора био је програм који декларише променљиву и иницијализује је на вредност 42, а затим је исписује на стандардни излаз. Овај програм је представљен кодом 34.

```
main() {
  var number = 42;
  print(${number});
}
```

Код 34: Програм за исписивање броја 42 у програмском језику Дарт.

Како би се омогућило дебаговање интерпретатора, направљен је механизам за дебаговање у МИПС асемблеру који садржи две опције: могућност исписивања редног броја метода класе *InterpreterGeneratorMIPS* који се тренутно извршава и могућност исписивања садржаја задатог регистра. Ти механизми су у процесу развоја били део интерпретатора. Када дође до грешке и прекида извршавања програма, овај механизам нам омогућава да знамо у којој функцији је дошло до грешке. Овај механизам је детаљније описан у поглављу 5.3.

Други програм који је успешно извршен помоћу МИПС интерпретатора је програм који исписује поздравну поруку. Овај програм је представљен кодом 35.

```
main() {
  print("Hello world");
}
```

Код 35: Програм за исписивање поруке „Hello world” у програмском језику Дарт.

Први програм из подржаног скупа тестова, који је покренут помоћу МИПС интерпретатора, је програм који исписује поздравну поруку и информације о машини на којој се извршава. Овај програм представљен је кодом 36.

```
main() {
  SystemInformation si = sys.info();
  String nodeInformation =
      si.nodeName.isEmpty ? '' : ' running on ${si.nodeName}';
  print('Hello from ${si.operatingSystemName}$nodeInformation.');
}
```

Код 36: Програм који исписује „Hello” и информације о машини на којој се извршава.

Заједно са развојем покретани су тестови који проверавају исправност интерпретатора. Више о тестирању речено је у поглављу 5.4, док је о перформансама речено у поглављу 5.6. Након што је потврђено да имплементирани интерпретатор задовољава све потребне услове, интегрисан је у Дартино пројекат [11].

5.3 Систем за дебаговање

Како се у току превођења Дартина, од интерпретатора генерише асемблерска датотека, која се затим користи за генерисање програма који извршава бајткод на МИПС архитектури, било је неопходно да се систем за дебаговање угради у асемблерски код.

У коду 37 представљен је макро у оквиру ког се налазе инструкције за чување садржаја регистра на стеку. У регистар се упише садржај са врха стека, а затим се врх стека помери за дужину једне речи, односно садржај се скине са стека.

У коду 38 представљене су инструкције за скидање садржаја регистра на стеку. На стеку се алоцира меморија за једну реч (помери се врх стека), а затим се садржај регистра упише на врх стека.

```
#define push_asm(reg)                                  \
  assembler()->subi(SP, SP, Immediate(1*kWordSize));\
  assembler()->sw(reg, Address(SP, 0));
```

Код 37: Макро за чување садржаја регистра са стека.

```
#define pop_asm(reg)                                   \
  assembler()->lw(reg, Address(SP, 0));                \
  assembler()->addi(SP, SP, Immediate(1*kWordSize));
```

Код 38: Макро за скидање садржаја регистра са стека.

У коду 39 приказан је макро који се користи да би се исписао садржај регистра. Неопходно је сачувати садржај регистара који се користе у макроу, како систем за дебаговање не би утицао на извршавање остатка програма. У *$a0* се смешта адреса ниске „print_reg”, коју генерише функција *GenerateDebugStrings* док се у *$a1* сачува садржај регистра који је аргумент макроа. Након тога се скочи на функцију *printf*, при чему се у *$a0* и *$a1* налазе аргументи функције. Сличан макро генерисан је за записивање редног броја функције интерпретатора која се тренутно извршава.

У коду 40 приказана је функција која се користи при генерисању ниски које се користе при дебаговању: „string_редни_број_ниске” и „print_reg”. Ниска „string_редни_број_ниске” представља лабелу која се записује у сектору података, иза које следи „.asciiz”, што представља тип податка, и на крају сам

```
#define PrintRegister(reg)                          \
  push_asm(GP);                                     \
  push_asm(V0);                                     \
  push_asm(A0);                                     \
  push_asm(A1);                                     \
  push_asm(T9);                                     \
  push_asm(RA);                                     \
  assembler()->la(A0, "print_reg");                 \
  assembler()->move(A1, reg);                       \
  assembler()->lw(T9, "%call16(printf)($gp)");      \
  assembler()->jalr(T9);                            \
  assembler()->nop();                               \
  pop_asm(RA);                                      \
  pop_asm(T9);                                      \
  pop_asm(A1);                                      \
  pop_asm(A0);                                      \
  pop_asm(V0);                                      \
  pop_asm(GP);
```

Код 39: Макро за исписивање садржаја регистра.

```
void InterpreterGeneratorMIPS::GenerateDebugStrings() {
  int i;
  char *str = (char *)malloc(10);
  printf("\n\t.data\n");
  for(i=1;i<=255;i++) {
    sprintf(str, "string_%d", i);
    printf("%s: .asciiz \"%d   \"\n", str, i);
  }
  printf("print_reg: .asciiz \"register_value: \%\%x\\n\"\n");
  free(str);
}
```

Код 40: Функција која генерише ниске које се користе при дебаговању, у сектору података у асемблерској датотеци.

податак, односно број. Ова ниска се користи у интерпретатору при исписивању редног броја функције која се тренутно извршава. Ниска „print_reg" користи се при исписивању садржаја регистра, што је приказано кодом 39. Ова ниска такође представља лабелу у сектору података, иза које следи тип „.asciiz", и на крају вредност податка, односно садржај регистра који се исписује. Више о запису у сектору података у МИПС асемблерском језику описано је у поглављу 2.9.

5.4 Тестирање

Упоредо са имплементацијом интерпретатора, писани су мали тест примери. Примери неких тестова дати су у опису имплементације, и то су примери којима је тестиран интерпретатор. Овде ће бити наведени још неки од тестова који су помогли у решавању највећих грешака у окружењу интерпретатора.

Један од проблема који се јавио у току имплементације је функција *signalfd*, која у QEMU емулатору за МИПС није била имплементирана. То је установљено тест примером који је приказан кодом 41.

```
int fd = signalfd(-1, &signal_mask, SFD_CLOEXEC);
  if (fd == -1) {
    FATAL1("signalfd failed: %s", strerror(errno));
  }
```

Код 41: Позив функције `signalfd`, који је производио грешку при извршавању.

Након тога је направљен тест у програмском језику C, у ком се такође користи функција *signalfd*, и преведен за МИПС, како би се утврдило да ли разлог због ког не пролази тест има везе са интерпретатором. При извршавању тог теста јављала се грешка неподржане функције. Када је подршка за функцију додата у оквиру QEMU емулатора, тест је прошао.

Приликом тестирања рада са сокетима јавила се грешка при позиву функције *Socket.connect*, односно функције *sys.socket* која се налази у оквиру ње. Пример позива те функције дат је кодом 42.

```
fd = sys.socket(sys.AF_INET, sys.SOCK_STREAM, 0);
```

Код 42: Позив функције `sys.socket`, који је производио грешку при извршавању.

Утврђено је да унутар Дартина вредност макроа *SOCK_STREAM* није прилагођена МИПС архитектури, која користи различите вредности неких системских макроа у односу на остале архитектуре.

Направљено је и неколико тест примера у којима је уочен проблем при позиву функције *sys.setsockopt*, којој су прослеђиване погрешне вредности макроа *SOL_SOCKET* и *SO_REUSEADDR*. Пример позива те функције представљен

је кодом 43.

```
int _setReuseaddr(int fd) {
 int result =
  sys.setsockopt(fd, sys.SOL_SOCKET, sys.SO_REUSEADDR, FOREIGN_ONE);
  return result;
}
```

Код 43: Позив функције `sys.setsockopt`, који је производио грешку при извршавању.

Сличан проблем уочен је при прављењу теста у ком се генерише нова датотека. Проблем је био у макроима који се користе при системском позиву open: *O_CREAT*, *O_APPEND* и *O_NONBLOCK*.

Кодом 44 представљен је део Дартино датотеке *„system_linux.dart”*, док је кодом 45 представљен део датотеке *„system_posix.dart”* (које се налазе у оквиру библиотеке за приступ оперативном систему) у којима су исправљене вредности одговарајућих макроа, тако да зависе од архитектуре. Исправне вредности макроа за МИПС платформу добијене су читањем одговарајућих датотека у оквиру ланца алата за МИПС.

```
  static final bool isMips = sys.info().machine == "mips";
  int get AF_INET6 => 10;
  int get O_CREAT => isMips ? 256 : 64;
  int get O_TRUNC => 512;
  int get O_APPEND => isMips ? 8 : 1024;
  int get O_NONBLOCK => isMips ? 128 : 2048;
  int get O_CLOEXEC => 524288;

  int get FIONREAD => isMips ? 0x467f : 0x541b;
  int get SOL_SOCKET => isMips ? 65535 : 1;
  int get SO_REUSEADDR => isMips ? 4 : 2;
```

Код 44: Део датотеке ,,system_linux_dart’’ у ком су одговарајући макрои, тако да им вредности зависе од архитектуре.

```
  int get SOCK_STREAM => isMips ? 2 : 1;
  int get SOCK_DGRAM => isMips ? 1 : 2;
```

Код 45: Део датотеке „system_posix_dart” у ком су одговарајући макрои, тако да им вредности зависе од архитектуре.

5.5 Резултати

На крају имплементације, покренут је скуп тестова који је део Дартино пројекта. Тестови се покрећу помоћу пајтон (енгл. *python*) скрипте *tools/test.py*, а пример покретања тестова на платформи МИПС представљен је кодом 46. Неопходно је навести на којој се платформи покрећу тестови. Пошто је за неке тестове потребно мало више времена од подразумеваног, потребно је навести временско ограничење (енгл. *timeout*) од 1200 секунди. Временско ограничење представља максимално време извршавања појединачног теста, након чега се извршавање насилно прекида. Разлог зашто се неки тестови извршавају дуже него на ARM платформи је што на МИПС32Р2 не постоје неке инструкције које постоје на ARM архитектури, па је било неопходно имплементирати их помоћу већег броја инструкција. Поређење са ARM архитектуром је веома важно, јер су ARM процесори веома заступљени на тржишту система са уграђеним рачунаром [25]. Убрзање би се постигло уколико би нам референтна платформа била МИПС64Р6, јер је уведен нови, богатији, скуп инструкција.

```
tools/test.py -axmips -t1200
```

Код 46: Команда за покретање тестова на платформи МИПС.

Постојећим скуповима тестова тестиране су следеће карактеристике језика:

1. Корутине
2. Влакна
3. Изолате
4. Коришћење функција неких других програмских језика
5. Конекција на ХТТПС сервер коришћењем ТЛС протокола
6. Рад са уграђеним библиотекама

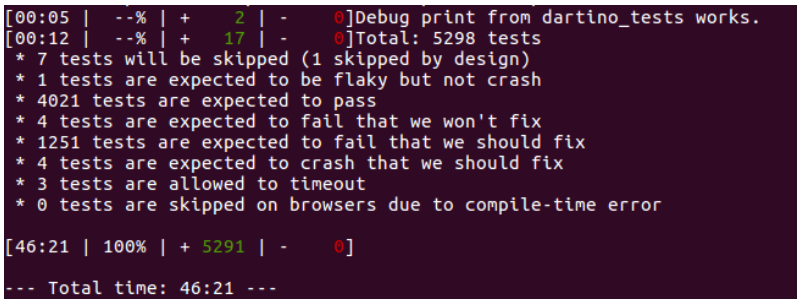
Кроз ове тестове тестира се исправност аритметичких и логичких операција, у једноставним случајевима као и у случајевима када може доћи до прекорачења. Поред тога, тестира се и трајање извршавања таквих операција, али и трајање других тестова. Затим, тестира се исправност поређења различитих објеката, претварање бројева у ниску, креирање изолата и чекање резултата, комуникација између процеса помоћу канала и портова. Такође, тестира се креирање влакана помоћу којих се у оквиру процеса чека на више ствари. Тестира се и коришћење неких спољних функција, дефинисаних у C датотеци, помоћу Дартино *ffi* библиотеке.

У овом скупу се налази 5291 тест. За 4021 тест очекивано је успешно извршавање, док је за 4 теста очекивано да резултат буде *failure*. Поред тога, за 1251 тест се очекује неуспешно ивршавање, јер још увек нису подржане све функционалности које ти тестови покривају (нпр. асинхроност). Грешка при истеклом временском ограничењу се може јавити за 3 теста, док постоје 4 теста код којих долази до прекида у извршавању (енгл. *crash*). То је резултат тестирања који се сматра успешним, и у том случају ће се на крају извршавања исписати **+5291 | -0**.

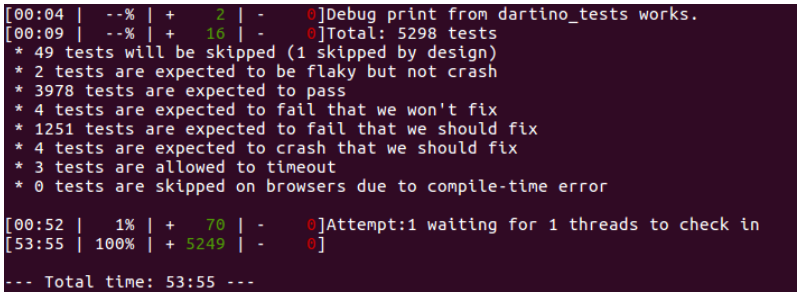
Резултат покретања постојећег скупа тестова, помоћу интерпретатора за МИПС, је приказан на слици 5.3. Тестови се у просеку извршавају за 45 минута, при навођењу временског ограничења од 1200 секунди. Сви тестови се извршавају успешно, а поред броја тестова који су се успешно извршили, исписује се и већ описани очекивани резултат. На плаформи ARM се тестови извршавају за око 50 минута, при временском ограничењу од 600 секунди. То је приказано на слици 5.4. Тестови се на платформи Интел x86 извршавају за 20 минута, без потребног навођења временског ограничења. То је представљено на слици 5.5. Интел x86 архитектура процесора припада CISC архитектурама, које имају много богатији скуп инструкција. Тестови за Интел се покрећу на рачунару са Интел x86-64 процесором, док се тестови за ARM и МИПС покрећу помоћу QEMU емулатора, те је и очекивано да се тестови на платформи Интел извршавају брже.

5.6 Поређење перформанси

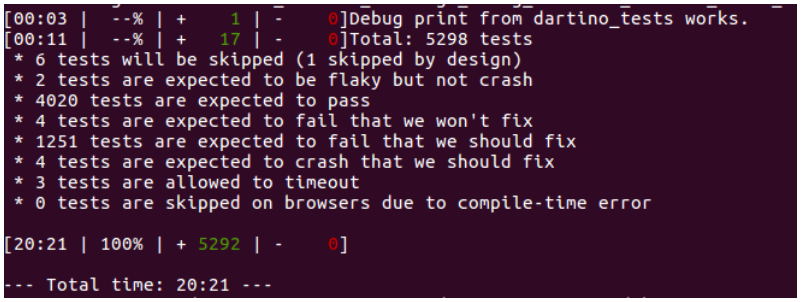
У склопу Дартино виртуелне машине, постојао је мултинаменски интерпретатор написан у програмском језику C++, који је било потребно конфигурисати



Слика 5.3: Резултати тестирања на платформи МИПС.



Слика 5.4: Резултати тестирања на платформи ARM.



Слика 5.5: Резултати тестирања на платформи x86-64.

за МИПС, на начин који је описан у поглављу 5.1. Помоћу њега су се могли извршавати програми написани у Дарту, који су одговарали функционалностима које су подржане у Дартину, у том тренутку. Међутим, компанија Гугл је крајем марта 2016.године престала са унапређивањем мултинаменског интерпретатора, а крајем априла се више није могао користити, јер није подржавао функционалности које су подржавали остали интерпретатори.

Идеја поређења перформанси се заснивала на поређењу перформанси мултинаменског интерпретатора и развијеног МИПС интерпретатора, на примеру неке апликације написане у Дарту. Од тренутка када је мутинаменски интер-

Табела 5.1: Време извршавања сортирања бројева у опсегу од 0 до 100, алгоритмом *qsort*, изражено у секундама.

бр.ел.	МИПС	мултинаменски	АРМ	МИПС у односу на мултинаменски	МИПС у односу на АРМ
10 000	6.17	1.27	7.23	+4.90	-1.06
20 000	12.49	12.27	14.51	+0.22	-2.02
40 000	25.44	24.65	29.44	+0.79	-4.79

претатор избачен, виртуелна машина је доста унапређена. На пример, измењена је библиотека *ffi* која омогућава да се из Дарт програма позивају функције из C програма, и обрнуто. Поред тога, додата је *mbedtls* библиотека, описана у поглављу 4.6. Такође, имплементиран је део библиотеке за рад са изолатама. Систем је доста проширен, међутим, са повећањем система изгубило се на перформансама. На пример, програм који се на тадашњој верзији Дартина могао извршити за 4 секунде, на тренутној верзији се извршава за 29 секунди. Због чињенице да мултинаменски интерпретатор није унапређиван, и чињенице да је на верзији на којој тај интерпретатор ради, систем доста бржи од тренутног, поређење перформанси на старијој верзији система и садашњој не даје праву слику.

Због једноставности мултинаменског интерпретатора, није било могуће направити неку комплексну апликацију у програмском језику Дарт, па су примера ради, перформансе упоређене на алгоритму сортирања *qsort*, што је приказано табелом 5.6. На примеру сортирања низа од 10 000 бројева, који су у опсегу од 0 до 100, МИПС интерпретатор је за 4.90 секунди спорији од мултинаменског, док је при сортирању 20 000 бројева, спорији за 0.22 секунде. При сортирању 40 000 бројева, МИПС интерпретатор је спорији за 0.79 секунди. У поређењу са АРМ интерпретатором, при сортирању 10 000 бројева, МИПС интерпретатор је бржи за 1.06 секунди, за 20 000 бројева је бржи за 2.02 секунде, док је при сортирању 40 000 бројева бржи за 4.79 секунди.

Поређење перформанси на скупу постојећих тестова такође не даје најбољу слику, јер је на старијој верзији Дартина, скуп тестова био мањи. На тој верзији је постојало 4847 тестова, који се успешно извршавају за око 24 минута. Тренутни скуп тестова се помоћу МИПС интерпретатора успешно извршава за око 45 минута, што је представљено на слици 5.3.

Иако се на овај начин показује да је МИПС интерпретатор спорији, успех се огледа у томе што се у оквиру Дартино виртуелне машине може користити

само МИПС интерпретатор, а подршка за мултинаменски је избачена. Такође, МИПС интерпретатор има више могућности од мултинаменског и јачу подршку, а поред тога, брзина МИПС интерпретатора не одудара од брзине АРМ интерпретатора.

Глава 6

Закључак

IoT представља идеју о проширивању интернета у огромну мрежу објеката који нас окружују. Ову мрежу чине системи са уграђеним рачунаром, засновани на једноставним микроконтролерима, и комплекснијим микропроцесорима. Апликације за микроконтролере се углавном развијају у асемблерском језику или у програмском језику C, због чега је развој доста успорен и не привлачи велики број програмера. Дартино виртуелна машина омогућује програмирање микроконтролера у програмском језику Дарт. Дарт је објектно-оријентисани програмски језик, који се заснива на класама, и подржава асинхроно програмирање и конкурентно извршавање. Користи се за развој веб-апликација, сервера и мобилних апликација, при чему синтакса језика доста подсећа на C.

Како је платформа МИПС веома заступљена у системима са уграђеним рачунаром, у раду је описана имплементација интерпретатора за платформу МИПС, у оквиру Дартино виртуелне машине. Помоћу овог интерпретатора, користећи Дартино на уређајима са МИПС процесором, могу се извршавати сви програми написани у Дарту, који садрже функционалности које су тренутном верзијом машине подржане. У оквиру виртуелне машине, постојао је мултинаменски интерпретатор написан у програмском језику C++, који се могао конфигурисати за платформу МИПС. Идеја је била направити интерпретатор који има боље перформансе од мултинаменског. Међутим, у раној фази развоја мултинаменски интерпретатор је избачен, чиме је онемогућено коришћење Дартино виртуелне машине на платформи МИПС, а самим тим и поређење перформанси. Имплементацијом описаног интерпретатора омогућено је коришћење Дартина на платформи МИПС, и у томе се огледа важност ове

имплементације. Поред тога, перформансе развијеног интерпретатора не одударају од перформанси интерпретатора за ARM.

Иако је у овом тренутку развој Дартино виртуелне машине заустављен, и многе карактеристике Дарта нису подржане, њеним развојем је потврђено да се може направити виртуелна машина за виши програмски језик као што је Дарт, која одговара перформансама микроконтролера.

Библиографија

[1] assembler_mips.cc. https://github.com/dartino/sdk/blob/master/src/vm/assembler_mips.cc.

[2] assembler_mips.h. https://github.com/dartino/sdk/blob/master/src/vm/assembler_mips.h.

[3] assembler_mips_linux.cc. https://github.com/dartino/sdk/blob/master/src/vm/assembler_mips_linux.cc.

[4] Coroutines and fibers. https://github.com/dartino/sdk/wiki/Coroutines-and-Fibers.

[5] Dart programming language. https://www.dartlang.org/.

[6] Dart programming language specification, 4th edition. https://www.dartlang.org/guides/language/language-tour.

[7] Dartino command line application. https://github.com/dartino/sdk/wiki/Dartino-command-line-application.

[8] Dartino libraries. https://dartino.github.io/api/.

[9] Gyp. https://gyp.gsrc.io/.

[10] interpreter.cc. https://github.com/dartino/sdk/blob/master/src/vm/interpreter.cc.

[11] interpreter_mips.cc. https://github.com/dartino/sdk/blob/master/src/vm/interpreter_mips.cc.

[12] mbed tls. https://tls.mbed.org/.

[13] Mqtt. http://mqtt.org/.

[14] Ninja. `https://ninja-build.org/`.

[15] Procesees and isolates. `https://github.com/dartino/sdk/wiki/Processes-and-Isolates`.

[16] An update on dartino. `https://github.com/dartino/sdk/blob/master/README.md`.

[17] *Dart Programming Language Specification, 4th edition.* ECMA International, 2015.

[18] Iain D. Craig. *Virtual machines.* Springer, 2005.

[19] Lizhe Wang Alexey Vinel Feng Xia, Laurence T.Yang. Internet of things. *International Journal of Communication Systems*, 2012.

[20] Aws Yousif Al-Taie Hasan Krad. A new trend for cisc and risc architectures. *Asian Journal of Information Technology*, 2007.

[21] Steve Heath. *Embedded systems design, second edition.* Newnes, An imprint of Elsevier Science, 2002.

[22] David A. Petterson John L.Hennessy. *Computer Architecure, A Quantitative approach, 4th edition.* Morgan Kaufmann Publishers, 2007.

[23] Kasper Lund. The internet of programmable things. GOTO Conference 2015, `https://www.youtube.com/watch?v=Hx2iGEAvZRk`, `http://gotocon.com/dl/goto-cph-2015/slides/KasperLund_InternetOfProgrammableThings.pdf`.

[24] The Santa Cruz Operation. *System V Application Binary Interface, MIPS RISC Processor, Suplement 3rd edition.* 1996.

[25] David Seal. *ARM Architecture Reference Manual, second edition.* Addison-Wesley Professional, 2001.

[26] Dominic Sweetman. *See MIPS Run.* Morgan Kaufman Publishers, 2007.

[27] MIPS Technologies. *MIPS 32 Architecture For Programmers, second edition.* MIPS Technologies, 2009.