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:

Кључне речи: МИПС, систем са уграђеним рачунаром, интерпретатор,

Садржај

Глава 1

Увод

Појам „Интернет ствари” (енг. *Internet of things*), скраћено ИоТ (енг. *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 (енг. *embedded system*) који преноси и прима информације путем мреже [15]. Системи са уграђеним рачунаром су системи специјалне намене, који обављају једну или више функција које тој намени одговарају (паметни телефони, дигитални сатови, паметне утичнице, МП3 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 [16]. Сматра се да је потенцијал за развој индустрије система са уграђеним рачунаром велики, и да је будућност у изградњи велики ИоТ система.

Развијање апликација за системе са уграђеним рачунаром се обично своди на програмирање у асемблеру или у програмском језику Ц (енг. *C*). Најчешће се користе компилатори ГЦЦ (енг. *GCC*) или ЛЛВМ (енг. *LLVM*), а од алата за дебаговање, обично алат ГДБ (енг. *GDB*). Развојно окружење се најчешће налази на другом рачунару, јер микроконтролер садржи малу количину РАМ и флеш меморије, па се на њему обично не може покренути ниједан регуларни оперативни систем, а и микролинукс често захтева више РАМ меморије од оне којом микроконтролер располаже. 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 (енг. *Google*) је дошла на идеју да омогући програмирање 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 Ц, 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 Арм 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

МИПС

У овој глави описана је МИПС архитектура процесора. У поглављу 2.1 описане су ЦИСК (енг. *CISC - Complex Instruction Set Computing*) и РИСК (енг. *RISC - Reduced Instruction Set Computing*) архитектуре процесора, док је у поглављу 2.2 описана архитектура МИПС. У поглављу 2.3 је описана проточна обрада на РИСК и МИПС процесорима, а у поглављу 2.4 је описан појам слот закашњења при скоковима и гранањима. У поглављу 2.5 су описани МИПС регистри. Скуп инструкција је описан у поглављу 2.6, а начини адресирања су описани у поглављу 2.7. Структура програма у МИПС асемблерском језику описана је у поглављу 2.8, а један пример МИПС позивне конвенције дат је у поглављу 2.9.

2.1 ЦИСК и РИСК

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 ИСА (енг. *ISA - Instruction Set Architecture*)[17].

ЦИСК 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. ЦИСК процесори се углавном користе на личним рачунарима, радним станицама и серверима, а пример таквог процесора је Интел x86 [19, 12].

РИСК 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 [19, 12]. Више о начинима адресирања речено је у поглављу 2.7. Највећа предност је проточна обрада, која се лако може имплементирати, за разлику од ЦИСК процесора код којих то није могуће. Више о проточној обради речено је у поглављу 2.3. РИСК процесори се углавном користе за апликације у реалном времену. Пример РИСК процесора су АРМ и МИПС .

2.2 МИПС

МИПС је РИСК архитектура процесора, настала средином осамдесетих, као рад Џона Хенесија и његових студената на универзитету Станфорд. Истражујући РИСК, показали су да се помоћу једноставног скупа инструкција, добрих компилатора и хардвера који ефикасније користи проточну обраду, могу произвести бржи процесори на мањем чипу [19].

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 друго [19]. 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 елек-

тронику, играчке конзоле и друго.

2.3 Проточни систем

Проточна обрада (енг. *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 [10]. Да би се омогућило једнако трајање фаза, РИСК 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 РИСК архитектура, међу којима је и МИПС, има фиксирану величину инструкција [19].

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 [19]. 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 РИСК архитектури кеш је део процесора и везан је за проточну обраду, док је код ЦИСК 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 [19]. У наставку ће бити описана свака од фаза.

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

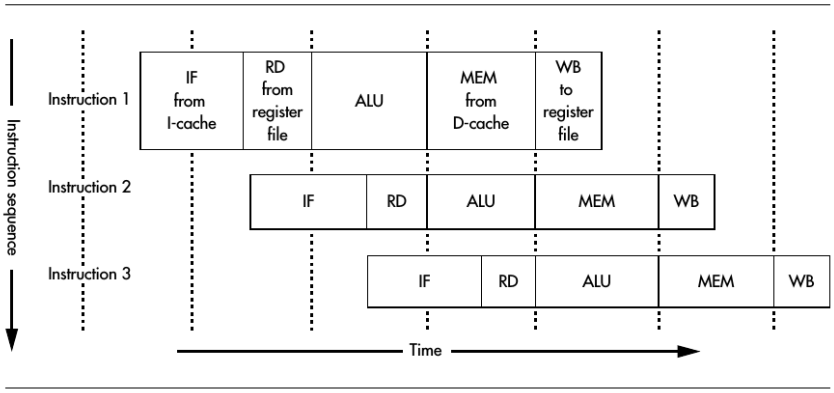
2. Читање садржаја одговарајућих регистара (операнада инструкције) (енг. read register, скраћено RR)

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

4. Фаза у којој се дохватају/пишу подаци у меморију. Углавном 3 од 4 инструкције не раде ништа у овој фази, али постоји да не би дошло до тога да две инструкције у исто време желе да приступе кешу (енг. memory, скраћено MEM)

5. Вредност(резултат) операције се уписује у регистар (енг. 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 [19]:

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 2.6

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 2.5

2.4 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. Позиција инструкције која следи након скока/гранања назива се слот закашњења гранања (енг. *Branch delay slot*). Када је то могуће, углавном се у слот закашњења смешта инструкција која би иначе била пре скока/гранања. То се не може применити када је у питању условни скок. У случајевима када се ништа не може ставити у слот закашњења, попуњава се инструкцијом *nop* [19].

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 (енг. *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 [19].

2.5 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 [19]: *$0 - $31*. Два регистра имају другачије понашање од осталих:

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

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 [19].

У наставку ће бити описани регистри опште намене [19]:

- **at** - Резервисан је за псеудоинструкције генерисане од стране асемблера. Псеудоинструкције 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.
- **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 2.9, 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.
- **s0 - s7** - По конвенцији која је описана у поглављу 2.9, 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.
- **k0, k1** - Резервисано за систем прекида, који након коришћења не враћа садржај на почетни.
- **gp** - Користи се у различите сврхе. У коду који не зависи од позиције (енг. *Program Independent Code*, скраћено *PIC*), свом коду и подацима се

приступа преко табеле показивача, познате као ГОТ (енг. *Global Offset Table*). Регистар *$gp* показује на ту табелу. У регуларном коду који зависи од позиције, може се користи као показивач на средину у статичкој меморији. То значи да се подацима који се налазе 32 бита лево и десно од овог регистра може приступити помоћу једне инструкције, тако што се овај регистар користи као базни. У пракси се на ове локације смештају глобални подаци који не заузимају много меморије.

- **sp** - Показивач на стек. 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. Одговорност је на функцији која је позвана. *$sp* се при уласку у функцију прилагођава на најнижу тачку на стеку којојој ће се приступати у функцији. Тако је омогућено да компилатор може да приступи променљивама на стеку помоћу константног помераја у односу на *$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** - При уласку у функцију овај регистар обично садржи адресу повратка функције, тако да се функције углавном завршавају инструкцијом *jr $ra*. У принципу, може се користити било који регистар, али неке оптимизације које врши процесор ће радити боље уколико се користи ra (нпр. предвиђање гранања (енг. *branch prediction*)). Функције које позивају друге функције морају сачувати садржај регистра *$ra*.

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

2.6 Скуп инструкција

У МИПС асемблерском језику постоје следеће инструкције [21]:

- 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 [6]

Р - 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 Р 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

И - Инструкције које имају као операнде регистар и константну вредност 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 И типу:

- addi - Сабирање вредности у регистру и константе
- slti - Постави 1 у регистар за резултат уколико је вредност у првом регистру мањи од константе
- beq - Гранање у случају да је вредност у првом регистру једнака константи

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.

2.7 Начини адресирања

На платформи МИПС постоје четири начина адресирања [13]:

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)

2.8 Структура програма

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

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 [8].

Кодни сегмент 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 [8].

Кодом 1 је приказан пример програма у МИПС асемблерском језику. Овај програм исписује поздравну поруку на стандарадни излаз, на следећи начин:

```
  .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 1: Програм који исписује поздравну поруку, у МИПС асемблерском језику

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 код 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 код за системски позив *exit*.

14. Након тога се позива оперативни систем да прекине извршавање програма.

2.9 Позивна конвенција

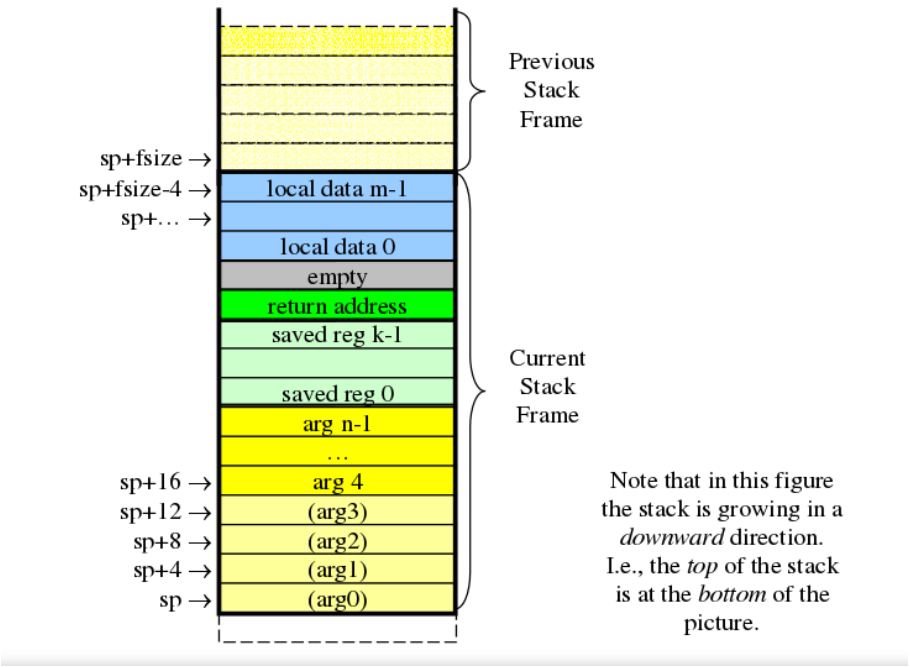
Позивна конвенција представља правила рада са регистрима и стек оквиром приликом позива функције. Не постоји јединствена МИПС позивна конвенција која се мора примењивати, тако да ће овде бити представљена она која је коришћена при имплементацији МИПС интерпретатора у оквиру Дартино виртуелне машине.

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 [7].

Стек оквир функције се дели на 5 региона, што је приказано на слици 2.2. У наставку ће бити описан сваки од њих [7].

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 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 представња простор за смештање садржаја регистара чувања (*$s0*-*$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



Слика 2.2: Организација стек оквира

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 конценција уводи и нека додатна правила:

- 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 аргумената се не морају чувати при позиву функције. 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

У овој глави описане су карактеристике програмског језика Дарт. У поглављу 3.2 описано је на који начин су подржани типови, док су у поглављу 3.3 описане специфичности при раду са класама. Рад са библиотекама је описан у поглављу 3.4. На који начин је подржана асинхроност описано је у поглављу 3.5, а подршка за конкуретне процесе описана је у поглављу 3.6. У поглављу 3.7 описан је рад са функцијама, док је у поглављу 3.8 описан појам метаподатака. Разлике Дарта у односу на јаваскрипт описане су у поглављу 3.9.

3.1 Уопштено

Дарт је објектно-оријентисани језик, заснован на класама и једноструком наслеђивању. Опционо се може превести у јаваскрипт. При покретању у веб прегледачу, преводи се у јаваскрипт помоћу dart2js компилатора, при чему се врши оптимизација јаваскрипт кода. У неким случајевима, код написан у Дарту се брже извршава од ручно написаног кода у јаваскрипту [20, 2].

Дарт је скалабилан језик који се може користити како за веб апликације, тако и за мобилне апликације, веб сервере, и програмирање система са уграђеним рачунаром.

Важни концепти

У наставку су описани неки од најважнијих концепата програмског језика Дарт [20, 2].

- Све што може бити променљива је објекат, при чему сви објекти наслеђују класу Object.
- Навођење типова је опционо, при чему постоји подршка за статичку проверу која не утиче на компилацију програма.
- Могу се креирати анонимне функције и затворења.
- Нема кључних речи public, private и protected. Идентификатори који почињу доњом цртом су приватни.
- Постоји подршка за асинхроно програмирање.
- Дарт уводи Изолате, као подршку 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 код је уједно лакши за читање и пружа бољу документацију [20, 2].

3.3 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 (енг. *Mixins*). Код неке друге класе се може искористити помоћу кључне речи *with* након које следи име класе чији се код копира (*Mixin*). Кодом 2 представљена је класа која користи *Mixins* Musical, Aggressive i Demented [20, 2].

Mixin се креира као класа која нема конструктор. Кодом 3 представљена је класа која је *Mixin*.

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

Код 2: Пример класе која користи *Mixin*

```
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 3: Пример *Mixin* класе

Још једна важна карактеристика Дарта је да се конструктори не наслеђују, па уколико класа нема дефинисан конструктор, Дарт ће јој доделити подразумевани. Не може се дефинисати више конструктора, али се могу креирати именовани конструктори. Кодом 4 представљено је дефинисање именованог конструктора *Point.fromJson* [20, 2]:

3.4 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

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

  Point(this.x, this.y);

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

Код 4: Пример дефинисања именованог конструктора

5. Када нам је потребна лењо учитана библиотека, укључујемо је функцијом *loadLibrary()*, што је представљено кодом 6 [20, 2].

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

Код 5: Пример лењог учитавања класе

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

Код 6: Пример укључивања лењо учитане библиотеке

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. Оне врате као резултат операцију која се извршава одређени период, и не чекају да се изврши.

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* АПИ-ја [20, 2].

Да би се користио *await*, код мора бити унутар функције која је означена као *async*. Пример у коме се помоћу *await* чека на резултат асинхроне функције, дат је кодом 7. Функција lookUpVersion, је нека аинхрона функција која враћа *Future* [20, 2].

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

Код 7: Пример употребе `await`

Future АПИ 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* АПИ-ја. Пример асинхроне for петље којом се итерира кроз догађаје стрима дат је кодом 8. Како је овде стрим низ целобројних догађаја, у петљи се прихвата догађај *Stream*-а 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* [20, 2].

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

Код 8: Пример употребе прихватања догађаја `Stream`-а

3.6 Изолате

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

Изолата је јединица конкурентног извршавања. Између изолата нема дељења меморије, а комуникација се обавља слањем порука. Свака изолата има своју хип меморију. Сваки Дарт програм се састоји из бар једне изолате, која се назива главна изолата (main isolate) [20, 2]. При превођењу на јаваскрипт, изолате се преводе у веб воркере. Пример употребе изолата дат је у поглављу 18.

3.7 Функције

Пошто је Дарт чисто објектно-оријентисани језик, и функције су објекти, типа *Function*. То значи да се функције могу додељивати промењивама, или прослеђивати као аргументи функције [20, 2].

Функције које садрже само један израз, могу се записивати у скраћеном облику, што је представљено кодом 9.

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

Код 9: Пример скраћеног облика записивања функција

Такође, у Дарту постоје анонимне функције које се могу креирати на начин приказан кодом 10. Методом forEach позива се анонимна функција за сваки елемент листе [20, 2].

```
var list = ['jabuke', 'pomorandze', 'banane'];
list.forEach((i) {
  print(list.indexOf(i).toString() + ': ' + i);
});
```

Код 10: Пример коришћења анонимне функције која штампа елемент листе

Могу се креирати затворења, која представљају функције која памте контекст у ком су креиране (могу приступати промењивама које су дефинисане у контексту у ком су креиране). Пример затворења дат је кодом 11. Функција makeAdder враћа као резултат функцију која додаје број addBy аргументу те функције, односно затворење. Ово затворење ће имати приступ промењивој

за коју је креирано. На почетку је променљива pom иницијализована на 3. Након тога се креира објекат функције makeAdder, са аргументом pom, који се додели променљивој add1. При позиву ове функције са аргументом 5, резултат ће бити 8. Након што променимо вредност променљиве pom на 7, позив функције за аргумент 5 ће вратити 12, јер се унутар затворења приступа тренутној вредности променљиве [20, 2].

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

main() {
  var pom = 3;
  var add  = makeAdder(pom);
  add1(5); // 8
  pom = 7;
  add(5); // 12
}
```

Код 11: Пример функције која враћа затворење

3.8 Метаподаци

Метаподаци се користе да пруже додатне информације о коду. Означавање метаподатака почиње карактером @ иза чега следи константа у време превођења (енг. *compile-time constant*), нпр. *deprecated*, или позив неког константног конструктора. У Дарту постоје три типа анотација (енг. *metadata annotations*): *@deprecated*, *@override*, и *@proxy* Анотација *@override* се користи за наглашавање намерног мењања метода надкласе. Пример употребе *@override* дат је кодом 12 [20, 2].

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

Код 12: Пример употребе @override

Могу се дефинисати нове анотације метаподатака, а пример креиране анотације дат је кодом 13. 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 14 [20, 2].

```
library todo;

class todo {
  final String who;
  final String what;

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

Код 13: Пример креирања анотације метаподатака

```
import 'todo.dart';

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

Код 14: Пример коришћења креиране анотације

Метаподаци се могу јавити пре библиотеке, класе, конструктора, функције, поља, параметра, декларације променљиве, *import* или *export* директиве, типског параметра или *typedef*-а [20, 2].

3.9 Специфичности у односу на јаваскрипт

У наставку су описане неке карактеристике Дарта, по којима се разлику од јаваскрипта [20, 2].

- Дарт подржава типове
- У Дарту само false представља нетачно, док у јаваскрипту то важи и за 0, *null*, *undefined* и „”
- Дарт подржава два метода за рад са ДОМ (енг. *DOM, скраћено од Document Object Model*) елементима: *query* и *queryAll*

- 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

Дартино

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 4.6 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 [14].

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 [14].

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 ГЦЦ-а или ЛЛВМ-а, при чему је систем за дебаговање поједностављен и смањен [18].

За програмски језик Дарт постоји Дарт виртуелна машина, која се користи за веб програмирање. У оквиру Дартина је искоришћен постојећи ЈИТ 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) [18]. Интерфејс за комуникацију је једноставан, јер окружење за извршавање мора бити веома једноставно, како би одговарало карактеристикама миктроконтролера.

Окружење за извршавање садржи командни АПИ, преко ког му компилатор може затражити да уради одређене задатке. Испод тога се заправо налази стек машина, тако да се путем овог АПИ-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

[18].

Слика 4.1: Слање порука ЈИТ 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 [18]. 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 [18]:

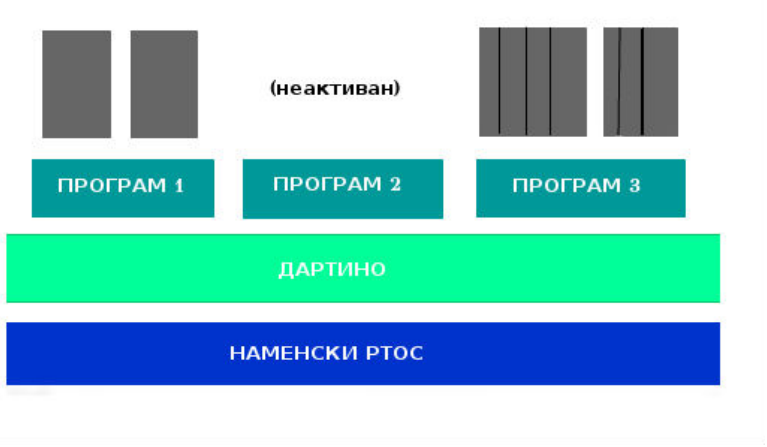
1. Покретање дебагера
2. Постављање зауставних тачака

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. Фри РТОС). 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 [18].



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

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 [18, 11, 1].

Лаки процеси

За сваки процес нам је потребно мало меморије, неколико стотина бајтова. Сав код 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 са уграђеним рачунаром.

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 (енг. *native thread*). На сваком језгру се процеси извршавају конкурентно, а Дартино веома добро распоређује велики број процеса на много мањи број језгара [18].

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

Код 15: Креирање паралелних процеса методом `Process.spawn`

Код 15 приказује покретање 4000 независних 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 [11].

У коду 16 се креира 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*)

```
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 16: Манипулисање надолазећим конекцијама методом `spawnAccept`

шаље серверу (*server*) поздравну поруку [18]. 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 [18].

```
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 17: Комуникација процеса слањем порука на одређени канал

Код 17 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 [18]:

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 (енг. *Isolates*) су независни воркери (енг. *Worker* [1]), 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 [11].

```
main() {
  Expect.equals(1597, fib(16));
  Expect.equals(4181, fib(18));
}

fib(n) {
  if (n <= 1) return 1;
  var n1 = Isolate.spawn(() => fib(n - 1));
  var n2 = Isolate.spawn(() => fib(n - 2));
  return n1.join() + n2.join();
}
```

Код 18: Употреба изолата

Код 18 приказује пример употребе изолата за рачунање елемената Фибоначијевог низа, рекурзивно. Покрећу се две независне изолате и на крају се спаја резултат.

Влакна

Влакна (енг. *Fibers*) представљају нити извршавања у оквиру једног процеса, које не деле непроменљиву меморију. Међутим, могу делити екстерну меморију (нпр. помоћу класе ForeignMemory [2]), 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 [1].

Код 19 приказује употребу два влакна. 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 [18]. Функција *publishOnChange* блокира извршавање док се у сензору за који је позвана не деси промена вредности.

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

[2] 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 19: Пример употребе два влакна која чекају на функцију `publishOnChange`, при чему не зависе једно од другог

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.

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 [1].

Кодом 20 дат је пример употребе корутина. 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.

```
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 20: Употреба корутина

4.5 Динамичко отпремање метода

Динамичко отпремање метода (енг. *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 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 [18].

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

Код 21: Пример хијерархије класа помоћу ког се илуструје генерисање табеле отпремања метода

У коду 21 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 [18].

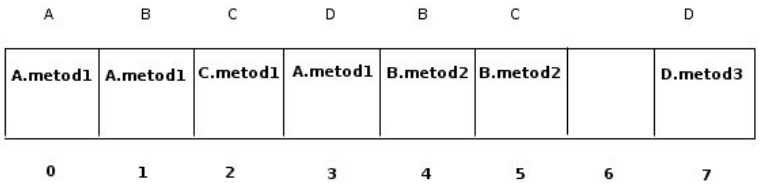
У наставку је пример рачунања позиције метода у табели отпремања:

- 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.

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 [4].



Слика 4.3: Табела отпремања за класе у примеру 21

file - Интерфејс за рад са датотекама. Подржано само када се Дартино извршава на ПОСИКС 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 Ц.

http - Имплементација ХТТП (енг. *HTTP - HyperText Transfer Protocol*) клијента.

mbedtls - ТЛС (енг. *TLS - Transport Layer Security*) подршка, базирана на mbedtls[3] [5]. Ово се може користити на исти начин као нормални сокет (и да се прослеђује ХТТП пакету).

mqtt - MQTT клијентска библиотека за MQTT (скраћено од MQ Telemetry Transport) протокол, ИоТ протокол за размену порука заснован на објављивању/претплати [9].

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

socket - Дартино имплементација ТЦП (енг. *TCP - Transmission Control Protocol*) и УДП (енг. *UDP - User Datagram Protocol*) сокета

stm32 - Подршка за STM32[4] плоче [**?**].

[3] Имплементација ТЛС и ССЛ сигурносних протокола и одговарајућих алгоритама криптовања

[4] Фамилија микроконтролера базираних на 32-битним ARM процесорима

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 (енг. *persistent process*). Дарт процес омогућава компилацију, док Дартино виртуелна машина (**dartino-vm**) извршава и дебагује програме .

Нешто више о извршним фајловима које се користе при покретању програма [3]:

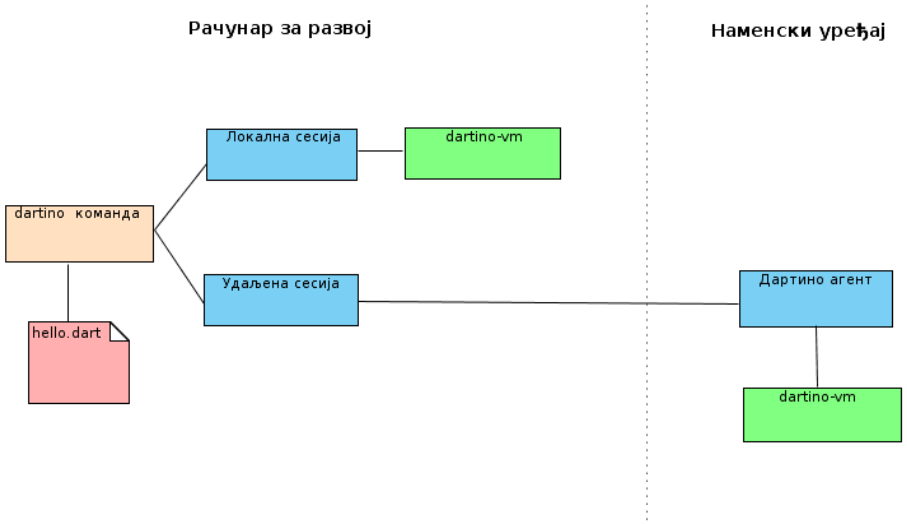
dartino Ово је мали извршни фајл који преко сокета прослеђује стандардне улазно/излазне сигнале процесу dart. Служи за покретање ЈИТ компилатора који је написан у Дарту и извршава се у оквиру Дарт виртуелне машине

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.

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 код 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.

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 [18, 3].



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

У овој глави је описан поступак имплементације интерпретатора за платформу МИПС, кроз делове кода и тест примере за које су одређене целине имплементиране. У поглављу 5.2 је описан стек Дартино виртуелне машине, док је у поглављу 5.3 описан имплементирани систем за дебаговање. У поглављу 5.4 описани су још неки значајни тест примери, резултати тестирања на постојећем скупу тестова и апликација у језику Дарт, имплементирана за потребе упоређивања перформанси интерпретатора.

5.1 Имплементација у програмском језику Ц++

Имплементација је рађена на рачунару са Интел x86 процесором, кроскомпилацијом, при чему је циљна архитектура МИПС32Р2 (енг. *mips32r2*, скраћено од mips32 release 2). У почетку развоја, постојао је мултинаменски интерпретатор, који не зависи од архитектуре. Требало је само конфигурисати Дартино за платформу МИПС, и тиме би превођење за МИПС било омогућено. Конфигурисање подразумева додавање опције „mips" у списку подржаних архитектура и додавање опција које се користе приликом превођења. Да би се при превођењу добио одговарајући извршни фајл, било је непходно ГЦЦ-у проследити аргументе који указују на МИПС архитектуру: *-mips32r2*, *-EL*, и линкеру проследити *-EL*, што указује на литл ендиан (енг. *little endian*[1]). Да би

[1]Начин записа података у меморији тако да је на нижој адреси нижи бит меморијске речи.

се у фази линковања користиле МИПС библиотеке, постављене су путање до њих. МИПС извршне датотеке се покрећу помоћу КЕМУ (енг. *QEMU - Quick Emulator*) емулатора у корисничком режиму[2]. Да би се то вршило аутоматски, непходно је команди **update-binfmts** проследити вредности које одговарају мипс ЕЛФ (енг. *ELF - format of Executable and Linking Format* [3]) датотекама, и путању до скрипте која покреће КЕМУ емулатор. На овај начин ће се при покретању извршних датотека чије се меџик (енг. *magic*) и маск (енг. *mask*) вредности поклапају са овим, покретање извршити помоћу КЕМУ емулатора за МИПС. Команда са одговарајућим аргументима је представљена кодом 22, док је скрипта која покреће КЕМУ представљена кодом 23.

```
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 22: Команда за аутоматско покретање МИПС извршних датотека помоћу КЕМУ-а. <skripta> представља путању до скрипте 23:

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

Код 23: Скрипта у којој се покреће КЕМУ емулатор

Где <qemu> представља путању до директоријума у ком се налази qemu емулатор.

Интерпретатор за платформу МИПС је урађен по угледу на постојећи интерпретатор за АРМ процесоре.

За почетак су направљене датотеке *assembler_ mips.h* и *assembler_ mips.cc* које описују МИПС асемблерски језик. Најпре је најпре дефинисана енумерација регистар која обухвата МИПС регистре. Више о МИПС регистрима речено је у поглављу 2.5. Направљене су класе *Label*, *Constant* и *Address*.

[2]У овом режиму КЕМУ може покретати процесе преведене за једну врсту процесора на другој врсти процесора

[3]Бинарни формат извршних датотека на Линукс оперативном систему

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

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

 private:
  int position_;
  static int position_counter_;
};
```

Код 24: Класа помоћу које се генеришу лабеле у МИПС асемблерском језику

Лабела представља линију у оквиру асемблерског фајла на коју се може скочити. У коду 24 је представљена класа *Label* која садржи два поља: *position* - представља редни број лабеле и *position_ counter* - заједничка променљива за све инстанце класе *Label*, на основу које се додељује позиција лабеле.

```
class Immediate {
 public:
  explicit Immediate(int32_t value) : value_(value) {}
  int32_t value() const { return value_; }

 private:
  const int32_t value_;
};
```

Код 25: Класа помоћу које се генеришу константе у МИПС асемблерском језику

У коду 25 је представљена класа *Immediate*. Инстанца те класе представља константу вредност целобројног типа.

У коду 26 је представљена класа *Address*. Инстанца те класе представља меморијску адресу, која је на платформи МИПС32P2 32-битна. Адреса се рачуна као збир адресе у базном регистру и помераја. Базни регистар је регистар опште намене који садржи 32-битну адресу.

Дефинисани су макрои за МИПС инструкције са одговарајућим бројем аргумената, помоћу којих ће се по потреби увести одговарајуће инструкције које

```
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: Класа помоћу које се представљају меморијске у МИПС асемблерском језику

се користе 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.

Након тога је имплементирана функција *Print* за уписивање асемблерских инструкција у фајл. Дефинисан је начин записивања регистара, лабела и адреса у МИПС асемблеру.

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

Код 28: Пример записивања лабеле у МИПС асемблерском језику

У коду 28 представљен је део функције *Print* који се односи начин записивања лабеле унутар асемблерског фајла. Пре позиције додаје се префикс „.L”

који означава да је у питању лабела.

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

Код 29: Пример записивања регистра у МИПС асемблерском језику

У коду 29 представљен је део функције *Print* који се односи начин записивања регистра унутар асемблерског фајла. Регистар се означава префиксом „$”, након чега следи ознака регистра.

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

Код 30: Пример записивања адресе у МИПС асемблерском језику

У коду 30 представљен је део функције *Print* који се односи начин записивања 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. Следећи корак је генерисање интерпретатора. Интерпретатор генерише асемблерску датотеку чијим превођењем добијамо програм који извршава бајткод на МИПС архитектури. Већина функција у интерпретатору представља имплементацију одговарајућих Дартино процедура.

Најпре су имплементиране специфичности у вези са МИПС позивном конвенцијом, описаном у поглављу 2.9. 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.

При скоку на функцију која се налази ван интерпретатора, мора се позвати функција за поравнање стека. Ова функција је представљена кодом 31. На основу позивне конвенције, описане у поглављу 2.9, 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.

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

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

По повратку из неке спољне функције, позива се функција *RestoreStack*, која ради инверзну операцију функције *PrepareStack*. Ова функција је представљена кодом 32.

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

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

Након тога настављена је постепена имплементација интерпретатора. Први циљ је био превођење програма који декларише променљиву и иницијализује је на вредност 42, а затим је исписује на стандардни излаз. Овај програм је представљен кодом 33.

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

Код 33: Програм за исписивање броја 42 у програмском језику Дарт

Направљени су механизми за дебаговање у МИПС и ARM асемблеру који садрже две опције: могућност исписивања редног броја функције интерпретатора која се тренутно извршава и могућност исписивања садржаја задатог регистра. Ти механизми су у процесу развоја били део интерпретатора. Коришћени су

како би се извршавањем програма на ARM архитектури добила информација о функцијама које је неопходно имплементирати да би се тај програм извршио на МИПС архитектури. Такође, 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.

Након програма који исписује број 42, имплементиран је остатак функција који је неопходан за исписивање поруке „Здраво свете". Пример Дарт програма који исписује поруку представљен је кодом 34.

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

Код 34: Програм за исписивање поруке „Hello world" у програмском језику Дарт

Затим су имплементиране функције неопходне за превођење програма са основним аритметичким операцијама, рад са низовима, битовски оператори и слично. Кодом 35 представљена је функција која сабира два броја, при чему се проверава да ли је дошло до прекорачења. Уколико је било прекорачења, непходно је скочити на део кода који врши опоравак.

Након тога имплементиране су функције које су неопходне да би се исправно извршио програм који исписује поздравну поруку и информације о машини на којој се програм извршава, и тај програм представљен је кодом 36.

На крају су имплементиране преостале функције, како би се могао покренути постојећи скуп тестова.

5.2 Дартино стек

У оквиру Дартино виртуелне машине се ради са локалним стеком, и за те потребе је искоришћен регистар S2. Све операције са стеком у оквиру интерпретатора се раде над тим стеком, при чему стек процесора користи Дарт виртуелна машина, односно компајлер. Функције која врши скидање садржаја са стека је представљена кодом 37. У регистар се упише садржај са врха стека, а затим се врх стека помери за дужину једне речи, односно садржај се скине

```
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 35: Функција у МИПС интерпретатору која сабира две целобројне вредности

```
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

са стека. Функција која поставља садржај на стек је представљена кодом 38. На стеку се алоцира меморија за једну реч (помери се врх стека), а затим се садржај регистра упише на врх стека.

```
void InterpreterGeneratorMIPS::Pop(Register reg) {
  __ lw(reg, Address(S2, 0));
  __ addiu(S2, S2, Immediate(1 * kWordSize));
}
```

Код 37: Функција за скидање садржаја регистра са локалног Дартино стека.

```
void InterpreterGeneratorMIPS::Push(Register reg) {
  __ addiu(S2, S2, Immediate(-1 * kWordSize));
  __ sw(reg, Address(S2, 0));
}
```

Код 38: Функција за чување садржаја регистра на локалном Дартино стеку.

5.3 Систем за дебаговање

С обзиром да се функције у интерпретатору извршавају у току превођења кода, а не у току извршавања, било је неопходно да се систем за дебаговање угради у асемблерски код.

У коду 39 представљен је макро у оквиру ког се налазе инструкције за чување садржаја регистра на стеку. Чување садржаја на стеку је имплементирано као у функцији која је представљена кодом 38, при чему је једина разлика то што се ради са процесорским стеком.

У коду 40 представљене су инструкције за скидање садржаја регистра на стеку. Макро је имплеметиран исто као функција у коду 37.

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

Код 39: 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 40: Макро за скидање садржаја регистра са стека.

У коду 41 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.

```
#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 41: 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 42: Функција која генерише ниске које се користе при дебаговању, у сектору података у асемблерској датотеци

У коду 42 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 податак, односно број. Ова ниска се користи при исписивању редног броја функције у интерпретатору која се тренутно извршава. Ниска „print_reg” користи се се исписивању садржаја регистра, што је приказано кодом 41. Ова ниска такође

представља лабелу у сектору података, иза које следи тип „.asciiz”, и на крају вредност података, односно садржај регистра који се исписује. Више о запису у сектору података у МИПС асемблерском језику описано је у поглављу 2.8.

5.4 Тестирање и резултати

Упоредо са имплементацијом интерпретатора, писани су мали тест примери. Примери неких тестова дати су у опису имплементације. Овде ће бити наведени још неки од тестова који су помогли у решавању највећих багова.

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 43.

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

Код 43: Позив функције `signalfd`, који је производио грешку при извршавању

Након тога је направљен тест у програмском језику Ц, 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 КЕМУ 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 44.

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

Код 44: 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 45.

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

Код 45: 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 46 представљен је део Дартино датотеке *„system_linux.dart”*, док је кодом 47 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 МИПС тулчејна (енг. *toolchain*).

```
  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 46: 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 47: Део датотеке „system_posix_dart” у ком су одговарајући макрои, тако да им вредности зависе од архитектуре

На крају имплементације, покренут је скуп тестова који је део Дартино пројекта. Тестови се покрећу помоћу пајтон (енг. *python*) скрипте *tools/test.py*, а пример покретања тестова на платформи МИПС представљен је кодом 48. 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 (енг. *timeout*) од 1200 секунди. Разлог зашто се неки тестови извршавају дуже него на ARM платформи је што на МИПС32P2 не постоје неке инструкције које постоје на ARM архитектури, па је било неопходно имплементирати их помоћу већег броја инструкција. Убрзање би се постигло уколико би нам референтна платформа била МИПС64P6, јер је уведен нови, богатији, скуп инструкција.

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

Код 48: 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

Резултат тестирања на платформи МИПС је приказан на слици 5.1. Тестови се у просеку извршавају за 45 минута, при навођењу временског ограничења од 1200 секунди. На плаформи ARM се тестови извршавају за око 50 минута, при временском ограничењу од 600 секунди. То је приказано на слици 5.2. Тестови

се на платформи Интел x86 извршавају за 20 минута, без потребног временског ограничења. То је представљено на слици 5.3.

```
[00:05 |  --% | +    2 | -    0]Debug print from dartino_tests works.
[00:12 |  --% | +   17 | -    0]Total: 5298 tests
 * 7 tests will be skipped (1 skipped by design)
 * 1 tests are expected to be flaky but not crash
 * 4021 tests are expected to pass
 * 4 tests are expected to fail that we won't fix
 * 1251 tests are expected to fail that we should fix
 * 4 tests are expected to crash that we should fix
 * 3 tests are allowed to timeout
 * 0 tests are skipped on browsers due to compile-time error

[46:21 | 100% | + 5291 | -    0]

--- Total time: 46:21 ---
```

Слика 5.1: Резултати тестирања на платформи МИПС

```
[00:04 |  --% | +    2 | -    0]Debug print from dartino_tests works.
[00:09 |  --% | +   16 | -    0]Total: 5298 tests
 * 49 tests will be skipped (1 skipped by design)
 * 2 tests are expected to be flaky but not crash
 * 3978 tests are expected to pass
 * 4 tests are expected to fail that we won't fix
 * 1251 tests are expected to fail that we should fix
 * 4 tests are expected to crash that we should fix
 * 3 tests are allowed to timeout
 * 0 tests are skipped on browsers due to compile-time error

[00:52 |   1% | +   70 | -    0]Attempt:1 waiting for 1 threads to check in
[53:55 | 100% | + 5249 | -    0]

--- Total time: 53:55 ---
```

Слика 5.2: Резултати тестирања на платформи ARM

```
[00:03 |  --% | +    1 | -    0]Debug print from dartino_tests works.
[00:11 |  --% | +   17 | -    0]Total: 5298 tests
 * 6 tests will be skipped (1 skipped by design)
 * 2 tests are expected to be flaky but not crash
 * 4020 tests are expected to pass
 * 4 tests are expected to fail that we won't fix
 * 1251 tests are expected to fail that we should fix
 * 4 tests are expected to crash that we should fix
 * 3 tests are allowed to timeout
 * 0 tests are skipped on browsers due to compile-time error

[20:21 | 100% | + 5292 | -    0]

--- Total time: 20:21 ---
```

Слика 5.3: Резултати тестирања на платформи x86-64

5.5 Пример апликације у програмском језику Дарт

Глава 6

Закључак

Библиографија

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

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

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

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

[5] mbed tls. https://en.wikipedia.org/wiki/Mbed_TLS.

[6] Mips assembly details. https://en.wikibooks.org/wiki/MIPS_Assembly/MIPS_Details.

[7] Mips calling convention summary. http://acm.sjtu.edu.cn/w/images/d/db/MIPSCallingConventionsSummary.pdf.

[8] Mips quick tutorial. http://logos.cs.uic.edu/366/notes/mips%20quick%20tutorial.html.

[9] Mqtt. https://en.wikipedia.org/wiki/MQTThttps://www.youtube.com/watch?v=Hx2iGEAvZRk.

[10] Pipelining. http://cs.stanford.edu/people/eroberts/courses/soco/projects/risc/pipelining/.

[11] Procesees and izolates. https://github.com/dartino/sdk/wiki/Processes-and-Isolates.

[12] Risc vs cisc. https://cs.stanford.edu/people/eroberts/courses/soco/projects/risc/risccisc/.

[13] Summary of addressing modes in mips. `https://www.cs.umd.edu/class/sum2003/cmsc311/Notes/Mips/addr.html`.

[14] Virtuelna mašina. `https://en.wikipedia.org/wiki/Virtual_machine`.

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

[16] Steve Heath. *Embedded systems design, second edition.*

[17] David A. Petterson John L.Hennessy. *Computer Architecure, A Quantitative approach, 4th edition.*

[18] 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`.

[19] Dominic Sweetman. *See MIPS Run.* Morgan Kaufman publishers, 2007.

[20] Dominic Sweetman. *Dart Programming Language Specification, 4th edition.* ECMA International, 2015.

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