УНИВЕРЗИТЕТ У БЕОГРАДУ
МАТЕМАТИЧКИ ФАКУЛТЕТ

Сретен Ковачевић

V8 - ИМПЛЕМЕНТАЦИЈА НЕОПТИМИЗУЈУЋЕГ WEBASSEMBLY КОМПИЛАТОРА ЗА MIPS АРХИТЕКТУРУ

мастер рад

Београд, 2018.

Ментор:

др Филип Марић, ванредни професор
Универзитет у Београду, Математички факултет

Чланови комисије:

др Милена Вујошевић-Јаничић, ванредни професор
Универзитет у Београду, Математички факултет

др Милан Банковић, доцент
Универзитет у Београду, Математички факултет

Датум одбране: _____

Наслов мастер рада: v8 - имплементација неоптимизујућег WebAssembly компилатора за MIPS архитектуру

Резиме:

Кључне речи:

Садржај

Глава 1

Увод

Глава 2

Архитектура *MIPS*

У овој глави је описана архитектура *MIPS* процесора. У поглављу 2.1 су описане архитектура процесора *CISC* (скраћено од енгл. *Complex Instruction Set Computing*) и *RISC* (скраћено од енгл. *Reduced Instruction Set Computing*) и њихове разлике, а у поглављу 2.2 *MIPS* архитектура. У поглављима 2.3 и 2.4 описане су инструкције и регистри *MIPS* архитектуре. У поглављу 2.5 описан је механизам проточне обраде и њена имплементација на *MIPS* архитектури, а у поглављлу 2.6 је представљен слот закашњења. Поглавље 2.7 описује на који начин *MIPS* архитектура води рачуна о редоследу бајтова у меморији.

2.1 *CISC* и *RISC* архитектура

Архитектура у рачунарству представља спој организације (начин комуникације међу различитим деловима рачунара), хардвера (примена конкретних логичких кола) и скуп инструкција и регистара *ISA* (скраћено од енгл. *Instruction Set Architecture*) [7]. Да би разумели разлику између *CISC* (скраћено од енгл. *Complex Instruction Set Computing*) и *RISC* (скраћено од енгл. *Reduced Instruction Set Computing*) архитектуре довољно је да посматрамо последњу ставку.

Процесори дизајнирани по *CISC* архитектури карактеришу се великим бројем инструкција које су на раслопагању програмеру. Овим дизајном се смањује број наредби у програму, по цену броја циклуса по инструкцији. Смањењем броја инструкција по програму постиже се мања потрошња меморије [1]. Инструкције често имају различиту дужину записа, а могу и обављати неколико независних задатака. Оваква архитектура захтева сложен хардвер, што може довести до потешкоћа у разумевању и програмирању оваквих чипова. Пружају

могућност великог броја различитих начина адресирања. Овакви процесори се користе на личним рачунарима, радним станицама и серверима. Типичан пример је *Intel x86* серија процесора.

RISC архитектура користи високо оптимизован скуп инструкција. Мотив је супротан у односу на мотив *CISC* архитектуре. Смањује се број циклуса по инструкцији, али се зато добија мањи број инструкција, те је потребан већи број инструкција по програму. Јединствена карактеристика *RISC* архитектуре је проточна обрада (енгл. *Pipelining*). Проточна обрада се постиже преклапањем извршавања већег броја инструкција. Захваљујући томе постижу се боље перформансе у поређењу са *CISC* процесорима [1]. Како је број инструкција које су подржане мали, имплементација самог чипа је знатно једноставнија и јефтинија. Подржана су четири начина адресирања: регистарско, *PC*-релативно, псеудо-директно и базно. Процесори *RISC* архитектуре се користе у наменским уређајима (енгл. *embedded*) и примери су *ARM* и *MIPS*.

2.2 *MIPS*

MIPS је представник *RISC* архитектуре, настао средином осамдесетих година двадесетог века на Станфорд универзитету. Група студената, предвођена Џоном Хенесијем, је истраживала рад *RISC* процесора и открила да се бољом применом проточне обраде, која је до тад била недовољно искоришћена, може доћи до бржих процесора на мањем чипу. *MIPS* је током година успео да се одржи на тржишту, али и да сачува епитет једне од најједноставнијих архитектура [8].

Процесори из *MIPS* породице успели су да пронађу примену у наменским уређајима. Могу се наћи у мобилним уређајима, мрежним уређајима, сет-топ боксовима, паметним телевизорима. Све претходно наведене примене захтевају покретање апликација које захтевају интензивна израчунавања (процесирање слика и видеа, анализа података, интеракција међу субјектима, итд.).

2.3 Инструкције

У *MIPS* асемблерском језику све инструкције су једнаке дужине 32 бита. Могу се поделити у следеће групе [9]:

- аритметичке: сабирање, одузимање, множење, дељење
- логичке: и, или, шифтовање
- приступ меморији: учитавање, записивање у меморију
- гранања и скокови
- контролне

На основу типова операнада, инструкције се могу поделити у три типа [8]:

R - Инструкције које као операнд очекују регистре. Представљају се у следећем формату:

$$OP\ rd, rs, rt$$

OP представља ознаку инструкције, rd је регистар за смештање резултата, док су регистри rs и rt операнди. Неки од примера ових инструкција су:

- *jr* - скакање на адресу смештену у регистру
- *slt* - поставља 1 у регистар уколико је први аргумент мањи од другог
- *addu* - смештање у регистар збира аргумената, посматрајући аргументе као неозначене целе бројеве

I - Инструкције које као операнде имају регистар и константну вредност, у облику специјалне вредности која је уписана у инструкцију (енгл. *Immediate*), представљају се форматом:

$$OP\ rd, rs, Imm$$

OP представља ознаку инструкције, rd је регистар за смештање резултата, rs први операнд (регистар), а Imm представља константу која је други операнд. Константа може имати највише 16 бита. Примери ових инструкција су:

- *lw* - учитавање вредности са адресе $rs + Imm$ у одредишни регистар
- *sw* - смештање вредности из одредишног регистра на адресу $rs + Imm$
- *beq* - гранање уколико је вредност у регистру једнака константи
- *addiu* - смештање у регистар збира вредности из регистра и константе

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

j label

Постоје две инструкције овог типа, а то су *j* и *jal*. Код прве инструкције (енгл. *Jump*) се ток извршавања пребацује на позицију *label*, исто се дешава и са другом инструкцијом (енгл. *Jump and link*), али се и адреса наредне инструкције уписује у *$ra* регистар. Ове инструкције прихватају највеће константе, које су дужине 26 бита, што је оправдано великим бројевима којим се представљају адресе.

2.4 Регистри

Регистри представљају малу, веома ефикасну меморију која се налази у процесору. *MIPS* инструкције могу као аргументе да примају једино регистре и специјалне константе.У *MIPS* архитектури постоје 32 регистра опште намене, од којих два имају другачије понашање [8]:

$0 - Увек враћа нулу, без обзира шта се у њега уписује

ra - Користи се за смештање адресе повратка из функције приликом коришћења *jal* инструкције Сви остали регистри могу се равноправно користити у инструкцијама (чак се и регистар $0 може користити, али ће ре резултат у том случају бити занемарен).

У наставку ће бити описани регистри, као и њихово препоручено коришћење:

at - Резервисан за псеудоинструкције које генерише асемблер.

v0, v1 - Користе се за смештање целобројних повратних вредности функција. Уколико се резултат не може сместити у два регистра, компилатор ће резултат сместити у меморију, а адресу у ове регистре.

a0 - a3 - Користе се за смештање прва четири целобројна аргумента при позиву функција. Остали се смештају на стек.

t0 - t9 - Привремени регистри, није потребно рестаурирати вредност након коришћења.

s0 - s7 - Садржај ових регистара мора остати непромењен након сваке функције, што се постиже привременим чувањем њиховог садржаја на стеку уколико се користе. Дужност да сачува њихову вредност има позвана функција (енгл. *callee saved registers*).

k0, k1 - Резервисани за системе прекида оперативног система, иначе се ретко користе.

gp - Има два начина примене. Уколико се ради о коду који не зависи од позиције (енгл. *Position Independent Code* скраћено *PIC*), овај регистар показује на табелу показивача (енгл. *Global Offset Table*). Уколико је у питању регуларан код, показује на средину у статичкој меморији. На тај начин се помоћу једне инструкције може приступити било ком податку који је 32KB лево или десно од њега. Овај регистар не користе сви системи за компилацију и сва окружења за извршавање.

sp - Показивач на стек. Стање стека је потребно експлицитно ажурирати, те се инструкције за одржавање показивача на стек углавном генеришу на почетку и на крају функција. Како стек расте надоле, на почетку функције се поставља на најнижу тачку до које ће стек расти.

fp - Показивач на стек оквир. Користи се од стране функције, за праћење стања на стеку. Уколико се при превођењу не може одредити на коју вредност да се постави **sp** регистар, променљивим на стеку се приступа помоћу овог регистра.

ra - Подразумевани регистар за смештање адресе повратка из функције. Овакво понашање је подржано кроз одговарајуће инструкције скока. Ово је разлика у односу на *x86* архитектуру, где се адреса повратка смешта на стек. Функције се углавном завршавају наредбом *jr $ra*. Иако се може користити и било који други регистар, то се не препоручује због оптимизација које врши процесор у случају коришћења овог регистра. Функције које позивају друге функције морају сачувати његову вредност.

2.5 Проточна обрада

Проточна обрада (енгл. *pipelining*) почива на чињеници да различите фазе извршавања користе различите ресурсе. Уколико имамо систем у ком је свака фаза једнаке дужине, добили би систем код ког би на крају завршетка фазе за једну инструкцију у ту фазу ушла следећа инструкција [8]. Да би овакав систем био могућ, процесори *RISC* архитектуре бирају минималан скуп инструкција које имају приближно исто време извршавања у свакој фази. Такође, инструкције су исте дужине како би се осигурало да је фаза декодирања идентична у свакој фази. Оваква конфигурација може се видети и у *MIPS* архитектури [8].

Како би проточна обрада била ефикасна користи се кеш меморија, чиме се убрзавају приступи меморији. Кеш меморија је мала, веома брза, локална меморија у којој се налази копија података из меморије [8]. У кешу се чувају подаци које је процесор најскорије користио, док се најстарији подаци преписују (уколико је кеш попуњен). Када процесор у кешу не пронађе потребне податке („промашај кеша", дешава се у 10% случајева), тада се приступа меморији.

Код *RISC* архитектуре, кеш је уско повезан за процесор и активно се користи за имплементацију проточне обраде, док се код *CISC* архитектуре кеш посматра као део меморијског система. *MIPS* има одвојен кеш података и инструкцијски кеш, што омогућава симултано читање инструкција и уписивање или читање података.

MIPS инструкције су подељене у пет фаза, и трајање сваке фазе је фиксирано. Прва, трећа и четврта фаза трају по један такт процесора, док друга и пета захтевају пола такта за своје извршење [8]. У наставку је описана свака од фаза.

1. Дохватање инструкције из инструкцијског кеша и њено декодирање
2. Читање садржаја наведених регистара
3. Извршавање аритметичко/логичких операција у једном такту (операције у покретном зарезу, множење и дељење су сложеније и раде се другачије)
4. Дохватање и уписивање у меморију. У 75% случајева инструкције не раде ништа у овој фази, али она постоји да не би више инструкција чекало на приступ кешу података
5. Резултат операције се уписује у одредишни регистар

2.6 Слот закашњења

Слот закашњења (енгл. *Delay slot*) је најосетнији ефекат проточне обраде из угла програмера. Због структуре проточне обраде на *MIPS* архитектури (која је описана у поглављу 2.5), у тренутку када наредбе гранања или скока дођу до фазе извршавања, рад на наредној инструкцији ће већ бити започет, иако је ток извршавања потенцијално промењен. Започета инструкција се извршава без обзира на исход наредбе промене тока извршавања и на тај начин се започети посао не одбацује.

Како би се постигло да се у слоту закашњења не појави више од једне инструкције, наредбе гранања имају посебно понашање при ком се већ после пола такта у фази извршавања аритметичко/логиких операција зна где ће се извршавање наставити. Како и друга фаза траје пола такта, овим смо обезбедили да само једна инструкција може доспети до прве фазе, за чије обављање је потребан један такт. Оваква конфигурација пружа могућност програмеру или компилатору да промени редослед инструкција у програму и тако неко израчунавање смести у слот закашњења [8].

Упркос уштеди коју слот закашњења може да донесе, он представља и потенцијални ризик. Посебно треба истаћи условна гранања, у којима нека операција не треба да буде извршена у оба случаја. Некад је безбедније (или једино исправно) оставити у слоту закашњења инструкцију *nop*.

Још једна последица проточне обраде је и слот закашњења учитавања (енгл. *load delay slot*). Подаци дохваћени *load* инструкцијом постају расположиви тек након инструкције која следи иза ње. Стога се њен резултат не може користити у следећој инструкцији. Модерни процесори имају механизам блокирања резултата *load* инструкције. Уколико резултат проба да се искористи у следећој инструкцији, процесор ће зауставити извршавање док резултат не буде спреман. На ранијим верзијама такав код је имао недефинисано понашање [8].

2.7 Редослед бајтова

Глава 3

WebAssembly

WebAssembly (скраћено *Wasm*) је безбедан и преносив код ниског нивоа. Дизајниран је са идејом да обезбеди ефикасно извршавање и компактну репрезентацију. Главни циљ му је да омогући функционисање апликација високог нивоа на Вебу, али и да не прави никакве претпоставке о окружењу нити уводи нове функционалности, што би га чинило погодним за коришћење у оквиру других окружења [5]. У овој глави ће бити описан *WebAssembly* и његове карактеристике. У поглављу 3.1 ће бити приказан развој језика и заступљеност, а у поглављу 3.2 ће бити приказан дизајн *WebAssembly*-ја, као и карактеристике које са њим долазе. У поглављу 3.3 је описана његова структура, док је у поглављу 3.4 је дат приказ семантичких фаза. У поглављу 3.5 дат је преглед могућности које пружа *JavaScript* за коришћење *WebAssembly*-ја.

3.1 Развој и подршка

WebAssembly пројекат је започео као *W3C CG* (скраћено од енгл. *World Wide Web Consortium Community Group*), 29. априла 2015. године [14]. У групу су се временом укључили произвођачи 4 најпопуларнија прегледача, *Mozilla*, *Microsoft*, *Google* и *Apple*. Технологије које су делимично биле узор и сматрају се претечом су *asm.js* и *Google Native Client*. За мање од годину дана, ове компаније су имале спреман прототип имплементације подршке за *WebAssembly* у својим прегледачима [4], док су се прве верзије прегледача са подршком на тржишту нашле у току 2017. године.

C/C++ апликације могу врло лако превести на *WebAssembly*, а затим и покретати унутар прегледача. Тако је могуће користити разне библиотеке већ

Слика 3.1: Приказ видео игре

написане у овим језицима, као што су стандардна библиотека, *OpenGL* и *SDL* за графичко програмирање, *pthread* за вишенитно програмирање. Ове библиотеке користе инфраструктуру ниског нивоа које обезбеђује *WebAssembly*, док користе интерфејс других сервиса. Тако се, на пример, *OpenGL* извршава помоћу *WebGL*, док се уместо системских позива за добијање времена и датума ти позиви прослеђују самом прегледачу и користе се његови унутрашњи механизми [6].

Подршка целе заједнице дала је одличан замајац даљем развоју пројекта и његовом широком прихватању. Убрзо су се појавили и први примери који су за циљ имали да прикажу моћ новог кода. Један такав пример је и мала демонстрација у виду игре на *Unity* платформи, користећи *WebGL* библиотеку [13]. Изглед игре приказан је на слици 3.1. Уколико прегледач подржава *WebAssembly*, игра се покреће помоћу њега, док ће, у супротном, бити покренута *asm.js* верзија игре.

Брзо се отишло и корак даље, те су се тако појавили покушаји да се врло сложене и захтевне апликације покрену у прегледачу користећи *WebAssembly*. Пројекат који је добио највећу медијску пажњу и покушај да се популарна игра *Doom 3*. У тренутку њеног појављивања, 2004. године, била је једна од најзахтевнијих игара и представљала је велики залогај за рачунаре тог времена, док се њена графика и за данашње стандарде сматра врло квалитетном. Уз све то, и чињеницу да је у питању игра отвореног кода, постала је идеалан кандидат да се заиста тестирају могућности које пружа *WebAssembly*. Иако и даље у развоју, игра се успешно извршава и тренутно показује да је верзија која се покреће кроз прегледач користећи *WebAssembly* за 40-60% спорија од игре

која се извршава директно на машини. Иако резултат сам по себи не делује импресивно, треба додати да је то и до 3 пута брже у односу на верзију која користи само *Javascript* [3].

3.2 Дизајн

Пред *WebAssembly* су постављени одређени циљеви у погледу дизајна. Два основна циља су брза, безбедена и преносива семантика и ефикасна и преносива репрезентација. Оба циља са својим испуњењем језику пружају карактеристике неопходне да се постигне жељени ниво перформанси.

Из угла семантике *WebAssembly* је [5]:

- Брз - извршава се ефикасношћу која тежи ефикасности језика нижег нивоа (енгл. *native*), користећи предности савременог хардвера
- Безбедан - код се валидира и извршава у меморијском сефу, заштићеном окружењу које спречава угрожавање података или упаде
- Добро дефинисан - потпуно и прецизно дефинише исправне програме и њихово понашање на начин који је лако разумети, како формално тако и неформално
- Хардверски независтан - може се превести на свим архитектурама, персоналним и преносивим рачунарима и наменским уређајима
- Језички независтан - не фаворизује одређени језик вишег нивоа, стил програмирања или објектни модел
- Платформски независтан - може бити уграђен у прегледач, представљати самосталну виртуелну машину или представљати део неког већег окружења
- Отворен - програми могу комуницирати са окружењем користећи једноставне методе

Посматрајући репрезентацију *WebAssembly*-ја, можемо закључити да је [5]:

- Компактан - бинарни формат се брзо преноси захваљујући запису који је краћи и од обичног текста и од кода на језику ниског нивоа

- Модуларан - програм се може поделити на мање целине које се могу слати, кеширати и користити независно
- Ефикасан - може се декодирати, валидирати и компилирати у једном пролазу, како са *JIT* (скраћено од енгл. *Just-in-time*) тако и са *AOT* (скраћено од енгл. *Ahead-of-time*) компилацијом
- Проточан (енгл. *streamable*) - омогућује да декодирање, валидација и компилација почну пре него што су сви подаци на располагању
- Погодан за паралелизацију (енгл. *parallelizable*) - допушта да декодирање, валидација и компилација буду издељени у више независних паралелних задатака
- Преносив - не прави претпоставке о архитектури које нису широко распрострањене међу модерним хардвером

3.3 Структура

WebAssembly кодира језик ниског нивоа, који је налик на асемблер. Његову структуру чине [5]:

- Вредности
- Инструкције
- Замке (енгл. *Traps*)
- Функције
- Табеле
- Линеарна меморија
- Модули
- Уграђивач (енгл. *Embedder*)

У наставку ће бити описан сваки од елемената структуре.

Вредности

У оквиру *WebAssembly*-ја постоје четири типа вредности. То су целобројне вредности и бројеви у покретном зарезу (имплементирани по стандарду *IEEE 754-2008)*, оба у 32-битној и 64-битној варијанти. 32-битне целобројне вредности се користе и за репрезентацију истинитосних вредности (енгл. *boolean*) и меморијских адреса. На располагању су све уобичајене операције над овим типовима, као и конверзије међу њима. Не постоји разлика између означених и неозначених целих бројева, већ се на основу конкретне операције одлучује како ће се број посматрати [5].

Инструкције

Рачунски модел је заснован на принципима стек машине. Код се састоји од низа инструкција које се редом извршавају. Инструкције врше промене над подацима који се налазе на имплицитном стеку операнада и могу се поделити у две основне категорије. Једноставне инструкције са стека узимају аргумент и резултат смештају назад на стек. Контролне инструкције мењају ток извршавања програма. Програм је добро структуиран, односно подељен у блокове, петље и условне кодове и наредбе гранања могу да гађају само неке од ових структура [5].

Замке

Неке инструкције, под одговарајућим условима, могу изазвати замку (енгл. *trap*), које прекидају извршавање програма. *WebAssembly* не поседује механизам за обраду замки, већ се оне прослеђују окружењу, где се хватају и обрађују на одговарајући начин [5].

Функције

Код је подељен у одвојене функције. Свака функција прима низ вредности као параметре и враћа низ вредности као резултат[1]. Функције се могу међусобно позивати, укључујући и рекурзивне позиве. Функције могу декларисати локалне променљиве које се могу користити попут виртуелних регистара [5]. Пример функције која враћа факторијел броја дат је у оквиру кода 3.1.

[1]У тренутној имплементацији може се вратити само један резултат.

Табеле

Табела представља низ вредности неког типа. На тај начин се допушта програму да помоћу индекса индиректно приступи елементу. Тренутно, једини подржани тип је референца на функцију. Захваљујући томе, програм може позивати функције користећи само индекс табеле. Ово опонаша показиваче на функције [5].

Линеарна меморија

Линеарна меморија је непрекидан, променљив низ сирових бајтова. Таква меморија има иницијалну величину, али се може динамички проширити. Програм може учитати или уписати вредност у меморију на адресу било ког бајта (укључујући и непоравнату). Уколико се покуша приступ ван тренутних граница меморије, замка ће бити активирана [5].

Модули

WebAssembly у свом бинарном запису узима облик модула. Модул садржи дефиниције функција, табела, и линеарне меморије. Такође, може садржати и глобалне променљиве и константе. Дефиниције могу бити увезене тако што ћемо навести модул из ког увозимо и име дефиниције коју увозимо заједно са одговарајућим типом. Опционо, неке дефиниције могу бити извезене под једним или више различитих имена. Осим дефиниција, могу се додати иницијализациони подаци за меморије и табеле. Могу садржати и почетне функције, чије извршавање се одвија аутоматски [5]. У коду 3.1 дат је пример једноставног модула који извози функцију.

Уграђивач

Имплементација *WebAssembly*-ја је углавном уграђена (енгл. *embedded*) у окружење домаћина. То окружење одређује како ће модули бити учитани, како су увози доступни и како се приступа извезеним дефиницијама. Детаљи зависе од окружења и нису одређени структуром самог језика [5]. Конкретни примери уграђивача биће приказани касније.

```
(module
  (func $fact (param $0 i32) (result i32)
    get_local $0
    i32.const 0
    i32.eq
    (if (result i32)
      (then
        i32.const 1
      )
      (else
        get_local $0
        get_local $0
        i32.const 1
        i32.sub
        call $fact
        i32.mul
      )
    )
  )
  (export "factorial" (func $fact))
)
```

Код 3.1: Пример модула који извози функцију за рачунање факторијела

3.4 Семантичке фазе

Семантика је подељена у три фазе. За сваки део језика постоји одговарајућа фаза, а оне су [5]:

Декодирање - *WebAssembly* модули се шаљу у бинарном облику. Модул представљен кодом 3.1 у бинарном формату заузима 62 бајта и његов хексадекадни запис[2] дат кодом 3.2. Декодирање је процес који форматира и конвертује бинарни облик у интерну репрезентацију модула. Интерна репрезентација може бити у облику апстрактне синтаксе, али и конкретан машински код.

Валидација - Декодирани модул мора бити валидан. Ова фаза проверава услове добре дефинисаности како би се осигурало да је модул исправан и безбедан. Прецизније, врши се провера типова функција, као и низ инструкција које јој припадају како би се утврдило да је стек операнада конзистентно коришћен.

Извршавање - Уколико су прве две фазе успешно окончане, модул се може

[2]Хексадекадни запис добијен коришћењем скупа алата *WABT* (скраћено од енгл. *WebAssembly Binary Toolkit*) [12].

```
00 61 73 6d 01 00 00 00 01 06 01 60 01 7f 01 7f
03 02 01 00 07 0d 01 09 66 61 63 74 6f 72 69 61
6c 00 00 0a 19 01 17 00 20 00 41 00 46 04 7f 41
01 05 20 00 20 00 41 01 6b 10 00 6c 0b 0b
```

Код 3.2: Хексадекадни запис модула

извршити. Сама фаза извршавања се састоји од две подфазе:

- Инстанцирање (енгл. *Instantiation*) - Инстанца модула је његова динамичка репрезентација, са сопственим стањима и стеком извршавања. У овој фази се извршава тело модула, све увезене дефиниције, иницијализују се глобалне променљиве, меморије, табеле и активира се почетна функција (уколико је дефинисана). Враћа примерке извоза модула.
- Позивање (енгл. *Invocation*) - Једном када је фаза инстанцирања завршена, њен резултат се може користити да се позивају извезене функције из претходно инстанцираног модула. Функцијама се прослеђују одговарајући аргументи, а као резултат се добија резултат њиховог извршавања.

Фазе инстанцирања и позивања су операције које одређује окружење домаћина.

3.5 JavaScript API

Као што је речено у секцији 3.2, *WebAssembly* може бити део неког већег система. Данас се *WebAssembly* модули углавном користе у већ постојећим *Javascript* апликацијама. Иако постоји идеја да се у будућности *WebAssembly* модули учитавају попут било ког *Javascript* модула, данас то није случај и стога је потребан посебан интерфејс.

Тај интерфејс је дужан да обезбеди 3 основна корака [11]:

1. Смештање бајт-кода у типизирани *ArrayBuffer*
2. Превођење бајт-кода у *WebAssembly.Module* објекат
3. Инстанцирање објекта типа *WebAssembly.Module* уз увожење и извожење потребних елемената.

У наставку ће бити приказан пример који показује употребу оваквог интерфејса за увоз и извоз функција. Нека је дат *WebAssembly* модул 3.3, зовимо га у наставку *simple.wasm* и *Javascript* код 3.4.

```
(module
  (func $i (import "imports" "i") (param i32))
  (func (export "e")
    i32.const 42
    call $i))
```

Код 3.3: *simple.wasm*

```
function instantiate(bytes, imports) {
  return WebAssembly.compile(bytes)
  .then(m => new WebAssembly.Instance(m, imports));
}

var importObject =
  { imports: { i: arg => console.log(arg) } };

bytes = readbuffer("simple.wasm")
instantiate(bytes, importObject)
.then(instance => instance.exports.e())
```

Код 3.4: *Javascript* код који користи модул из *simple.wasm*

simple.wasm увози функцију са два нивоа приступних имена (*imports.i*), те и променљива *importObject* мора да одражава такву структуру. Функција *fetch* добавља код модула из захтева пристиглог са мреже, након чега резултат преводимо помоћу *WebAssembly.compile()* метода и инстанцирамо модул прослеђујући бајт-код и увозни објекат *importObject* помоћу конструктора *WebAssembly.Instance*. Тада, из добијене инстанце можемо позвати изведену функцију. Резултат овог програма је исписивање броја 42 на стандардни излаз. На овај начин је у потпуности омогућена комуникација између *Javascript* и *WebAssembly* кода.

Још један битан део *WebAssembly* структуре програма који треба да буде доступан из *Javascript* кода је линеарна меморија. За *Javascript*, линеарна меморија је *ArrayBuffer* који је променљиве величине. Нова меморија може бити инстанцирана коришћењем *WebAssembly.Memory()* конструктора, који може при-

мати до 2 аргумента. Први је иницијална величина меморије, а други максимална дозвољена. Оба аргумента представљају број *WebAssembly* страница, од којих је свака величине 64 килобајта. Овај објекат садржи елемент *buffer* који је типа *ArrayBuffer*.

Осим креирања, могуће је приступити сваком појединачном бајту користећи оператор []. Над *WebAssembly.Memory* објектом дозвољена је и операција проширивања меморије помоћу метода *grow*. Као и код функција, меморија може бити и увезена и извезена из модула. На овај начин, добијамо могућност да у сваком тренутку проверимо или променимо стање меморије, али и да поставимо иницијално стање меморије.

Глава 4

V8

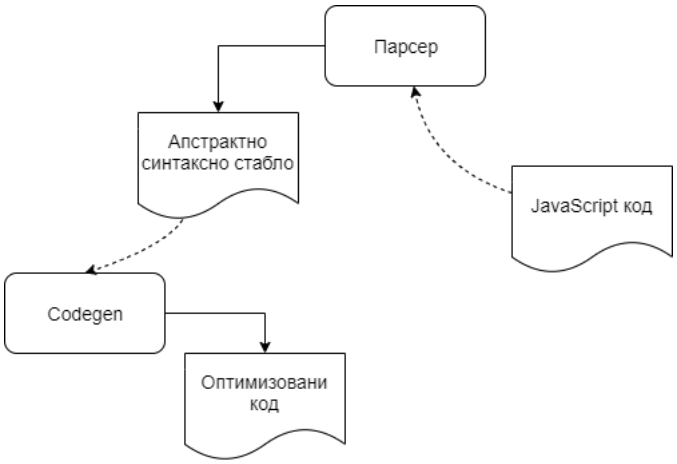
V8 је *JavaScript* мотор (енгл. *engine*) отвореног кода, написан у језику *C++*, развијен од стране компаније *Google*. Осим у оквиру њиховог пројекта *Chromium*, који се користи као основа за многе веб прегледаче, међу којима је најпопуларнији *Google Chrome*, *V8* је саставни део и *Node.js* пројекта. У поглављу 4.1 биће речи о развоју *V8 JavaScript* мотора, док ће у поглављу 4.2 бити представљене његове карактеристике. Оптимизујући компилатор *TurboFan* биће приказан у поглављу 4.3. У поглављу 4.4 биће приказана подршка за *WebAssembly* унутар *V8*.

4.1 Развој

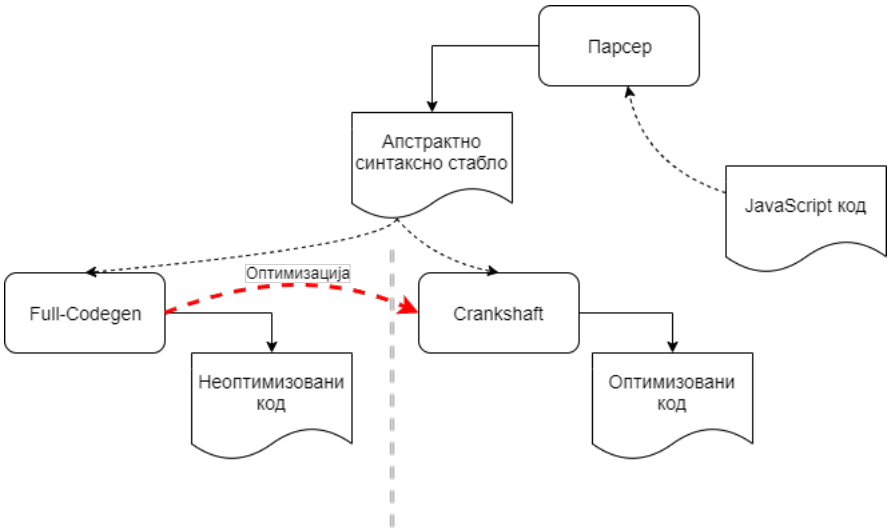
Прва званична верзија *V8 JavaScript* мотора појавила се, заједно са првом верзијом *Google Chrome* веб прегледача, у септембру 2008. године. Креатор овог пројекта је дански програмер Ларс Бак [15]. Првобитна архитектура предвиђала је његову употребу само у оквиру веб прегледача.

V8 се може покренути на 3 различита оперативна система (*Windows*, *MacOS* и *Linux*), али и на различитим процесорским архитектурама, од којих су најпопуларнији *x64*, *IA-32*, *ARM* и *MIPS*.

Временом се мењала архитектура *V8* мотора. У тренутку појављивања, она је била врло једноставна. Парсер је од улазног *JavaScript* кода стварао апстрактно синтаксно стабло, које је било улаз компилатору под називом *Codegen*, чији резултат је био оптимизован код. Архитектура је приказана на слици 4.1. Ипак, даљом анализом и развојем, дошло се до закључка да и даље постоји доста простора за нове оптимизације, те се тако променила и архитектура,



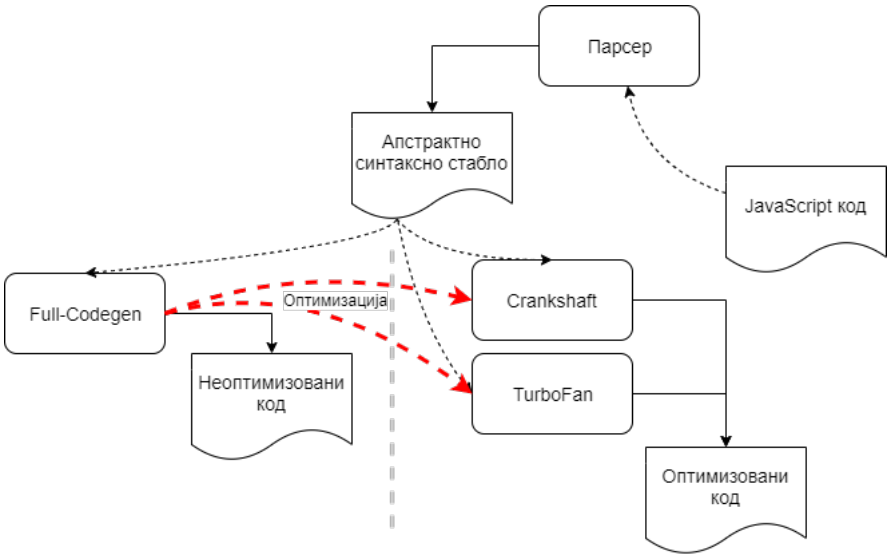
Слика 4.1: Архитектура *V8* из 2008. године



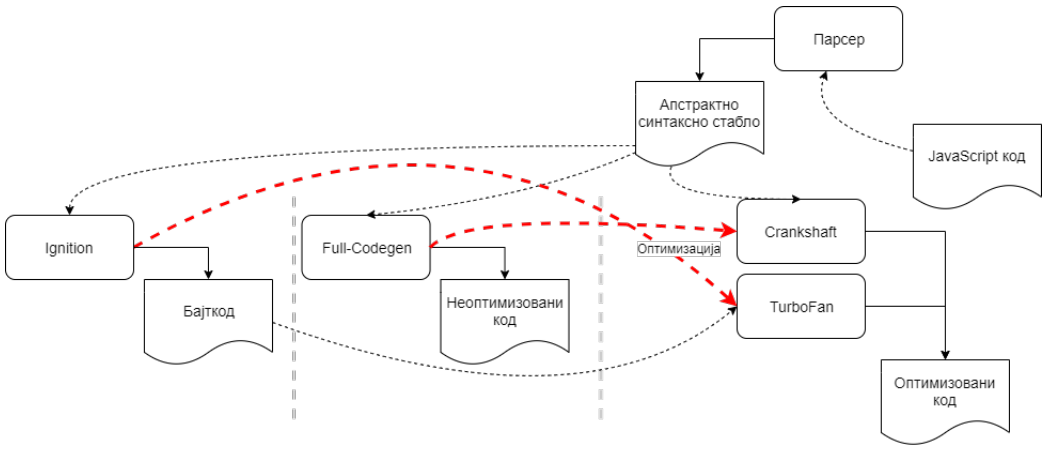
Слика 4.2: Архитектура *V8* из 2010. године

слика 4.2. 2010. године *Codegen* замењен је неоптимизујућим компилатором, под називом *Full-Codegen*, који је, осим генерисања, скупљао и информације о извршавању и прослеђивао их оптимизујућем компилатору, који носи назив *Crankshaft*.

Ова архитектура је само 4 године касније добила новог члана. Нови оптимизујући компилатор, под називом *TurboFan*, представљао је идеју за будућност, са циљем да временом потпуно одмени *Crankshaft*. Нова архитектура је приказана на слици 4.3. Да би се транзиција на *TurboFan* у потпуности обавила, био



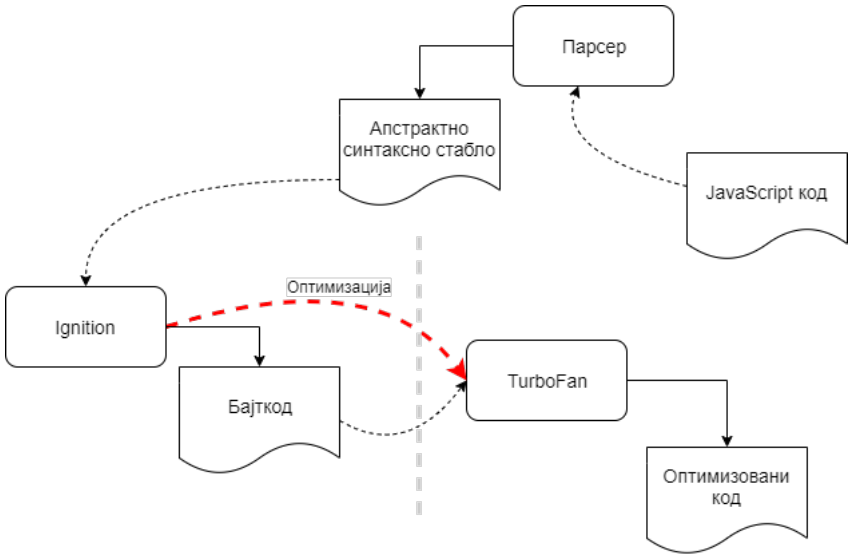
Слика 4.3: Архитектура *V8* из 2014. године



Слика 4.4: Архитектура *V8* из 2016. године

је неопходан још један део целој слагалици. У питању је интерпретер, под називом *Ignition*. Његовим додавањем, *TurboFan* је могао да престане да користи постојеће компоненте. Ова архитектура приказана је на слици 4.4 и представља стање из 2016. године. Само годину дана касније, застарели делови архитектуре, *Full-Codegen* и *Crankshaft*, су уклоњени из кода. Тако је добијена тренутна верзија, приказана на слици 4.5.

Промене архитектуре које су се догодиле су довеле до тога да *V8* нађе примену и мимо самог прегледача, те је тако постао неизоставни део *Node.js* про-



Слика 4.5: Архитектура *V8* из 2017. године

јекта. *V8* се може покренути и као део *C++* апликације, те се убрзо нашао и у другим популарним порјектима, као што су *Couchbase* (*NoSQL* база података), *Electron* (библиотека за развој корисничких интерфејса) и многи други.

4.2 Карактеристике

V8 је динамички компилатор за језик *JavaScript*. Техника динамичке компилације предвиђа одлагање дела (или свих) оптимизација за фазу извршавања. Предност оваквог приступа у односу на традиционалан, где се све оптимизације обављају унапред, огледа се у томе што, потенцијално, сваки вид употребе може резултовати другачије изгенерисаним кодом, оптималнијим за конкретну ситуацију.

Динамичка или *JIT* (скраћено од енгл. *Just-in-time*) техника компилације подразумева следеће елементе:

- Профајлер (енгл. *profiler*)
- Неоптимизујући копмилатор (енгл. *baseline compiler*)
- Оптимизујући компилатор

Профајлер или контролор (енгл. *monitor*) задужен је да прати извршавање кода и води евиденцију о „врућим" деловима кода. Код је „врућ" уколико извршавање често доводи до истог дела кода. Такви делови кода су погодни за оптимизацију.

Неоптимизујући компилатор је први вид побољшања перформанси. Његово задужење је да изгенерише бајт-код за „врући" део кода. Над таквим бајт-кодом може доћи до мањих оптимизација, које не захтевају дубљу анализу и могу се брзо испоручити. На пример, неоптимизујући компилатор не прави претпоставке о типовима, већ покрива све могуће случајеве. Тако, може издвајати сегменте функција у мање функције (енгл. *stubs*). Уколико се овакве функције позивају из неке петље и сваки пут се врши провера типа, убрзање практично и не постоји. Пример можемо видети у коду 4.1.

```
function acc(array) {
  var result = 0;

  for (let i = 0; i < array.length; i++) {
    result += array[i]
  }

  return result;
}
```

Код 4.1: Пример функције са петљом

Управо је то пример у ком до изражаја долази оптимизујући компилатор. Узимајући у обзир резултате добијене од профајлера, али и бајт-код добијен неоптимизујућим компилатором, оптимизујући компилатор врши оптимизације у циљу уштеде ресурса. Уколико претпоставимо да је неоптимизујући компилатор изгенерисао мање функције за код из петље из примера 4.1, знамо да се при сваком пролазу кроз петљу врши провера типа како би се позвала одговарајућа функција. Врло је вероватно да сваки пролазак кроз петљу заправо има исти тип у оквиру једног пролаза кроз низ. Дакле, уколико би оптимизујући компилатор проширио *stub* изгенерисан од стране неоптимизујућег компилатора и уместо само тог малог сегмента изгенерисао код за целу функцију, или бар покрива више од саме петље (иако остатак функције не спада нужно у „врући" код), провера типа би се извукла ван петље.

4.3 Оптимизујући компилатор

Оптимизујући компилатор који се користи у оквиру *V8* пројекта назива се *Turbofan*. Заснован је на концепту званом „море чворова" (енгл. *Sea-of-nodes*) [2]. Овај концепт подразумева употребу једноставног графа за међукод (енгл. *Intermediate Representation*, скраћено *IR*), али, уместо уобичајених графова контроле тока (енгл. *Control-Flow-Graph* , скраћено *CFG*) и графова тока података (енгл. *Data-Flow-Graph*, скраћено *DFG*), користи се њихова комбинација.

Оваквом комбинацијом графова, тежи се структури која ће задржати повољне карактеристике из оба графа. Оригиналном *DFG* графу додају се информације о контроли тока. Осим грана зависности података, додају се гране зависности контроле тока. Због својстава које има *CFG* граф, ова структура је погодна за даљу обраду у циљу добијања крајњег производа, машинског кода. Другим речима, није потребно направити корак у назад зарад лакшег оптимизовања. Такође, добијени граф није стриктно условљен редоследом чворова, као што је то случај у *CFG* графу.

Са друге стране, карактеристике *DFG* графа, у форми статичке јединствене доделе (енгл. *Static Single Assignment*, скраћено *SSA*), пружају добру стартну позицију за даље оптимизације, као што су пропагација константи и анализа живости променљивих, али и многе друге. Овакве карактеристике омогућавају лаку манипулацију над чворовима графа. Најчешће операције над оваквим графом су операције редукције.

Највећи део оптимизација се одвија управо над *Sea-of-nodes* графом. Ипак, једном када се фаза оптимизације заврши, прелази се на традиционалан *CFG* граф. Управо предностни, као што су, на пример, већа слобода у редоследу чворова у графу, које су погодне за оптимизацију, чине овај граф мање погодним за фазу распоређивања инструкција (енгл. *Instruction Scheduling*). Веома често је, међу мноштвом потенцијалних распореда инструкција, тешко или немогуће изабрати прави. Стога се примена стриктнијег *CFG* графа показала успешнијом у овој фази [10].

4.4 *WebAssembly* подршка

Као што је поменуто у секцији 3.1, сви водећи произвођачи веб прегледача избацили су прототип подршке за *WebAssembly* у току 2016. године, а у току

2017. и прве званичне верзије са подршком. Такав је случај и са *V8* мотором компаније *Google*.

Донета је одлука да се подршка у оквиру *V8* мотора дода кроз *TurboFan* оптимизујући компилатор. У потпуности је искоришћена структура и функционалност која се користи за *JavaScript* код, једино је додат нови ниво енкапсулације за чворове изгенерисаног *Sea-of-nodes* графа, како би се сачувале доданте информације неопходне за *WebAssembly*.

Овакав приступ је довео до тога да се уз минималан посао дође до потпуне подршке и потпуно функционалног компилатора. Ипак, оптимизације које пружа *TurboFan*, осим погодности, донела је и један велики проблем: брзину превођења. Колико год код изгенерисан *TurboFan* оптимизујућим компилатором био ефикасан, време потребно да тај код постане расположив представља огроман губитак, чиме се анулира предност која се добија у извршавању, поготову за код који се извршава само једном.

Такве карактеристике указале су на потребу за неоптимизујућим компилатором. Улога неоптимизујућег компилатора би била да што брже изгенерише валидан код, а да се тек у наредним итерацијама, по потреби, позива *TurboFan* који би пружио оптималан код и побољшање перформанси у току извршавања. Тако се, почетком 2018. почело са увођењем неоптимизујућег *Liftoff* компилатора, са циљем да се комплетира подршка за *WebAssembly*.

Глава 5

Имплементација

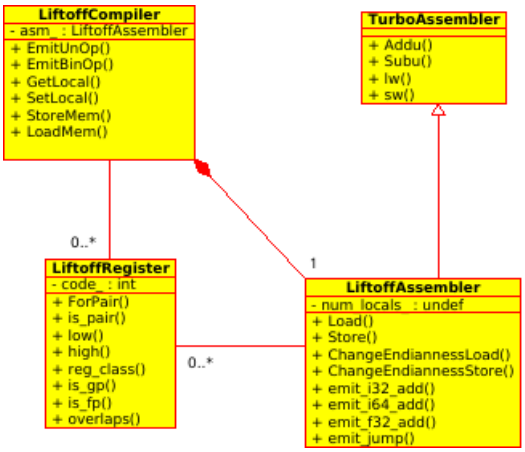
Имплементација *Liftoff* неоптимизујућег компилатора за *MIPS* архитектуру у оквиру *V8* мотора за *JavaScript* компаније *Google* рађен је по узору на имплементацију за *Intel*-ове *x86* и *x64* процесоре. У поглављу 5.1 биће објашњена архитектура *Liftoff* неоптимизујућег компилатора, док ће у поглављу 5.2 бити приказане измене у дељеним фајловима неопходне за генерисање *MIPS* машинског кода.

5.1 Архитектура *Liftoff* компилатора

Као што је поменуто у поглављу 4.4, *WebAssembly* оптимизујући компилатор се, заправо, заснива на *TurboFan*-у. Тако, ако погледамо класу *WasmCompilationUnit*, чија се дефиниција налази у датотеци *function-compiler.h*, можемо видети да она садржи поље *turbofan_unit_*, која представља инстанцу оптимизујућег компилатора.

Ова класа представља тачку одабира начина компилације. Тако, уз горе поменуту инстанцу оптимизујућег компилатора, овде додајемо и поље *liftoff_unit_*, типа *LiftoffCompilationUnit*, који је дефинисан у датотеци *liftoff-compiler.h*. Класа *LiftoffCompilationUnit* садржи 3 кључне методе: *ExecuteCompilation*, *FinishCompilation* и *AbortCompilation*.

Док су методе *FinishCompilation* и *AbortCompilation* задужене за уређивање стања након компилације или у случају прекида, у методу *ExecuteCompilation* се обавља практично комплетан посао. Тако се, у оквиру ове методе иницијализује декодер, који се, између осталог, састоји од инстанце *LiftoffCompiler* класе. Дијаграм класа које представљају основу функционисања *Liftoff* компилатора



Слика 5.1: Дијаграм класа

приказан је на слици 5.1. Све наведене класе садрже огроман број метода, те је приказан само мањи део, који ће служити као пример.

Класа *LiftoffCompiler* садржи инстанцу класе *LiftoffAssembler* и задужена је да, на основу декодера (исти декодер се користи и за оптимизујући компилатор) одреди низ машинских инструкција које ће бити изгенерисане. У зависности од циљне архитектуре користиће се одговарајућа имплементација *LiftoffAssembler* класе. *LiftoffAssembler* класа наслеђује *TurboAssembler* класу. Ова класа садржи све постојеће машинске инструкције за одговарајућу архитектуру, али и бројне макро-инструкције (низ машинских инструкција) које се често користе. На тај начин, класа *LiftoffCompiler* има приступ целокупном сету инструкција за дату архитектуру.

Бројне инструкције, директно (*LiftoffAssembler*) или индиректно (*LiftoffCompiler*) зависне од архитектуре, могу захтевати употребу регистара. Класа *LiftoffRegister* пружа механизам за рад са регистрима који су доступни за тренутну архитектуру. Тако, на пример, пружа функционалност упаривања регистара опште намене на 32-битним системима зарад обављања операција над 64-битним подацима. Такође, ова класа пружа и додатни слој апстракције, који омогућава да се регистри опште намене и регистри за рад са бројевима у покретном зарезу посматрају идентично.

5.2 *Liftoff* подршка за *MIPS*

Глава 6

Закључак

Литература

[1] Tarun Agarwal. *What is RISC and CISC Architecture with Advantages and Disadvantages*. 2017. URL: https://www.watelectronics.com/what-is-risc-and-cisc-architecture/ (посећено 12/28/2019).

[2] Clifford Noel Click. „Combining Analyses, Combining Optimizations". Докторска теза. Rice University, Феб. 1995.

[3] *D3Wasm*. URL: http://www.continuation-labs.com/projects/d3wasm/ (посећено 12/29/2019).

[4] *Experimental support for WebAssembly in V8*. URL: https://v8.dev/blog/webassembly-experimental (посећено 12/29/2019).

[5] WebAssembly Community Group. *WebAssembly Specification, release 1.0*. 2018.

[6] *Guide for C/C++ developers*. URL: https://webassembly.org/docs/c-and-c++/ (посећено 12/29/2019).

[7] John L. Hennessy и David A. Peterson. *Computer Architecture - A Quantitive Approach, Fourth Edition*. Morgan Kaufmann, 2007.

[8] Dominic Sweetman. *See Mips Run, Second Edition*. Morgan Kaufmann, 2007.

[9] MIPS Technologies. *MIPS Architecture For Programmers Volume II-A: The MIPS 32 Instruction Set*. MIPS Technologies, 2013.

[10] Ben L. Titzer. *TurboFan JIT Design*. 2016. URL: https://docs.google.com/presentation/d/1sOEF4MlF7LeO7uq-uThJSulJlTh--wgLeaVibsbb3tc/edit#slide=id.p (посећено 06/10/2021).

[11] *Understanding the JS API*. URL: https://webassembly.org/getting-started/js-api/ (посећено 12/29/2019).

[12] *WebAssembly Binary Toolkit*. URL: https://github.com/WebAssembly/wabt (посећено 08/12/2018).

[13] *WebAssembly Tanks Demo.* URL: `https://www.wasm.com.cn/demo/Tanks/` (посећено 12/29/2019).

[14] *WebAssembly W3C Community Group.* URL: `https://www.w3.org/community/webassembly/` (посећено 12/28/2019).

[15] *What is V8 JavaScript Engine?* URL: `https://blog.stackpath.com/v8-javascript-engine/` (посећено 01/03/2020).