Универзитет у Београду – Електротехнички факултет

Катедра за рачунарску технику и информатику

ДИПЛОМСКИ РАД

Интерпретативни емулатор рачунарског система са CHIP-8 архитектуром

Ментор:

доц. др Саша Стојановић

Студент:

Марко Кујунџић 2016/0418

Београд, август 2021. године

Универзитет у Београду – Електротехнички факултет

Катедра за рачунарску технику и информатику

ДИПЛОМСКИ РАД

Интерпретативни емулатор рачунарског система са CHIP-8 архитектуром

Ментор:

доц. др Саша Стојановић

Студент:

Марко Кујунџић 2016/0418

Београд, август 2021. године

Садржај

1 Увод

Од старих дана аркадних видео игара, па све до данас, индустрија видео игара се променила онолико колико је и остала иста. Иако су социјални аспекти играња видео игара и даље присутни, дизајн игара се драстично променио са развојем технологије. То је узроковало носталгију код људи који воле да играју видео игре, што је довело до тога да су старе аркадне видео игре поново постале популарне у свету видео игара.

Један од начина за корисника да проживи старе дане видео игара би био да набави физичку копију конзоле на којој би игра могла бити покренута. Међутим, како су старе аркадне видео игре произведене за играчке системе осмишљене пре чак тридесет или четрдесет година, набавити један од таквих система би било веома тешко и изазовно, можда чак и немогуће. Алтернатива физичкој конзоли би се могла наћи у емулатору ретро конзоле за видео игре, преко којег би корисник био омогућен да успешно покреће те игре на свом рачунару. Са развојем технологије и софтвера генерално, емулатори су постајали све популарнији и моћнији у заједници видео игара, што је успешно довело до тога да се премости технолошки јаз од тридесет или четрдесет година.

Емулатори представљају софтвер који служи за то да представи имитацију дизајна или функционалности неког хардвера. Ово омогућава људима да покрећу софтвер који је дизајниран и програмиран за специфичну машину на потпуно другој машини, која ће кроз добро дизајнирано софтверско решење успешно имитирати циљану машину.

Са проласком времена, хардвер напредује много брже од софтвера, и због тога чињеница да је могуће покретати стари софтвер на новијим платформама постаје невероватно битна, а постоје и назнаке да ће бити још битнија у будућности. У избору старог софтвера коришћеног за програмирање аркадних машина за видео игре, као најбољи намеће се *CHIP-8*.

CHIP-8 представља интерпретирани програмски језик, развијен почетком седамдесетих година прошлог века. Иницијално је коришћен на *COSMAC VIP* и *Telmac 1800* 8-битним микрорачунарима. Језик је осмишљен да би видео игре могле лакше да се програмирају за тадашње рачунаре, због хардверских ограничења која су те машине поседовале. Због свог хексадецималног формата, језик се сматрао јако

ефикасним, због тога што је минимална обрада текста била потребна, како би програм могао успешно да се изврши, а текст да се прикаже на екрану. Програми су се извршавали у оквиру *CHIP-8* виртуелних машина на тим микрорачунарима. [1]

У овом раду представљен је један могући начин реализације емулатора за машину која подржава *CHIP-8* архитектуру. Имплементација подразумева интерпретативни емулатор, који ће бити детаљно описан у другом поглављу.

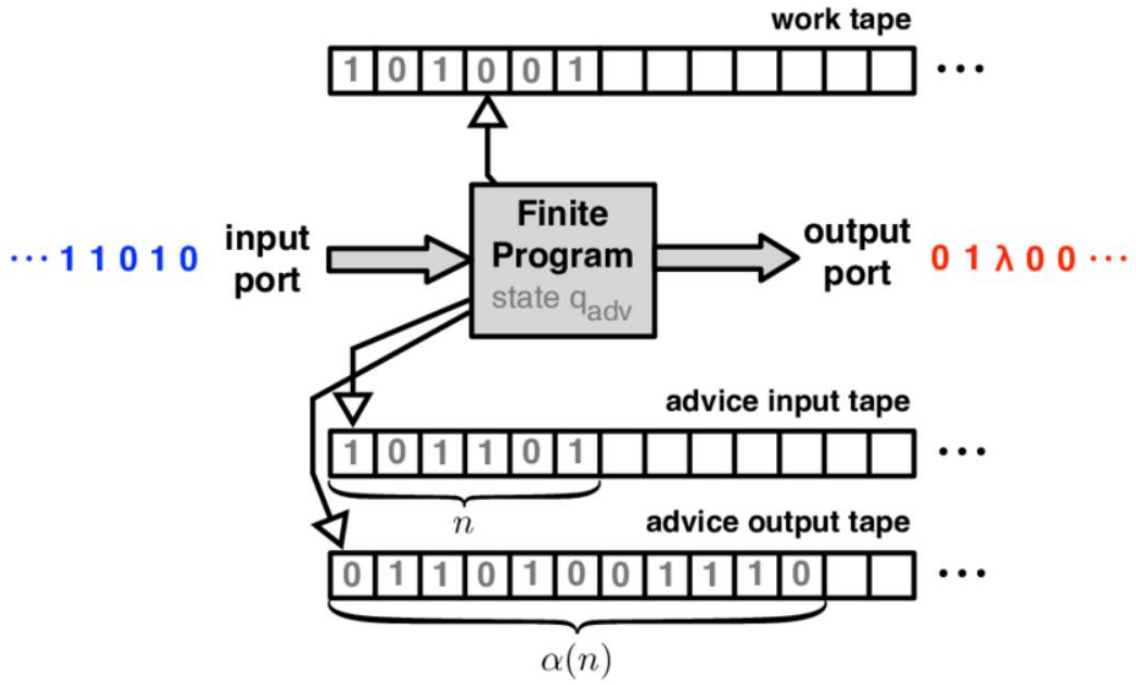
Рад је структуиран на следећи начин: поглавље два описује различите процесе емулације у рачунарским системима. У поглављу три детаљно су описане различите врсте емулатора. У поглављу четири детаљно је описан *CHIP-8* рачунарски систем и његова архитектура. У поглављу пет описани су најрелевантнији делови имплементације. У поглављу шест описан је пример коришћења програма, док је поглавље седам резервисано за закључак. На крају, поглавље осам представља списак коришћене литературе, док је поглавље девет резервисано за списак коришћених слика.

2 Процес емулације у рачунарским системима

Стандардна дефиниција за емулацију је „покушај да будеш једнак или бољи од некога или нечега“. Емулатор је, према томе, неко или нешто, ко емулира некога или нешто.

Емулатор у рачунарским системима користи исту аналогију, односно да емулира понашање хардверског уређаја на софтверски начин, или да емулира понашање неког дела софтвера коришћењем хардвера или софтвера. Врста емулатора који су у фокусу овог рада представљају **софтверске** емулаторе рачунарских система. Софтверски рачунарски емулатор мора да емулира све компоненте правог рачунарског система коришћењем својих рачунарских ресурса. Типична Фон Најманова архитектура [2], која се састоји од једног или више процесора, магистрале која служи да повеже меморију са осталим уређајима и периферијама, представља типичан пример архитектуре која се користи код емулатора. Да би се емулатор сматрао успешним, потребно је да постигне функционалност која успешно опонаша рад свих компоненти укључених у ову архитектуру. Ова архитектура, ће детаљно бити описана у наредном делу рада.

Емулатори су базирани на Тјуринговим машинама [4]. Тјурингова машина представља аутомат који је способан да чита податке са траке, која представља њену меморију, и извршава одређене операције над тим подацима. Тјурингове машине су корисне за моделирање рачунара и проблема који могу бити решени коришћењем ралунара, користећи математичку теорију. Једна од карактеристика Тјурингових машина је да свака Тјурингова машина може да емулира, односно опонаша другу Тјурингову машину, користећи искључиво своје ресурсе. Због тога се каже да су емулатори базирани на математичким способностима Тјурингових машина. Горенаведена особина везана за опонашање друге Тјурингове машине назива се Тјуринг комплетност.



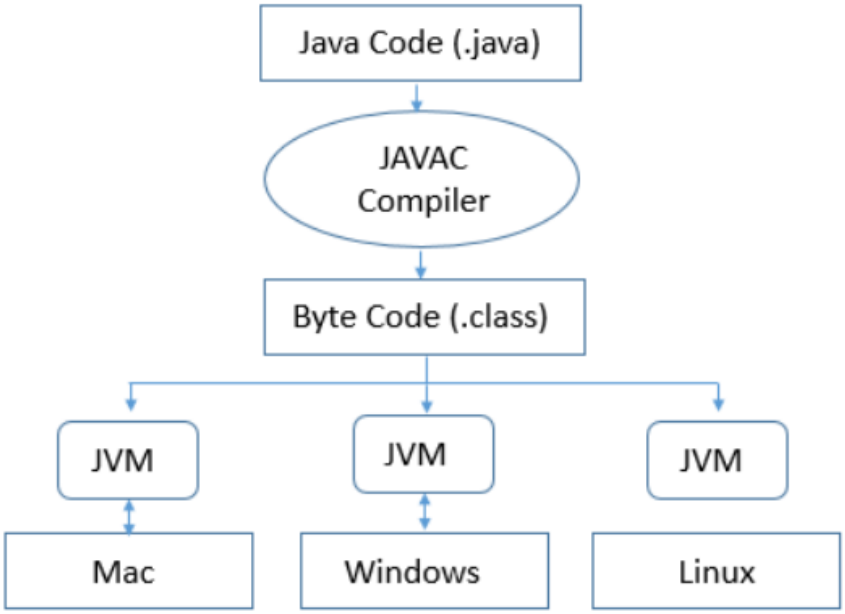
Слика 2.1. *Изглед Тјурингове машине*

2.1 Виртуелне машине

Дефинифија софтверског емулатора је веома уско повезана са концептом **виртуелне машине**. Виртуелна машина представља било коју врсту рачунара која не постоји у виду правог хардвера, него је имплементирана софтверским путем, на некој машини. Виртуелна машина поседује разне корисне карактеристике, као што су пружање преносивости програмског кода, или сакривање неких хардверских карактеристика од софтвера. На пример, оперативни систем пружа виртуелну машину кориснику програма. Наравно, виртуелна машина може бити имплементирана тако да емулира одређену врсту рачунара на рачунару на којем се тренутно извршава. Из тих разлога, емулатор се може сматрати виртуелном машином. Имплементација виртуелних машина је постала веома популарна са пробијањем програмског језика *Јава* на софтверско тржиште. *Јава* као виртуелна машина, користећи виртуелни процесор који користи бајткод је најприближнији концепту емулатора.

Јава виртуелна машина, познатија као *JBM,* се може посматрати као емулатор за стандардну машину која не поседује праву хардверску имплементацију, а опет је у стању да преводи и извршава код написан на програмском језику *Јава. Јава* изворни код се, после успешног превођења уз помоћ програмског преводиоца, преводи на

бајткод инструкције. Бајткод инструкције се после тога могу извршавати на било којој машини која на себи поседује Јавину виртуелну машину, без обзира на оперативни систем на том рачунару. Јавина виртуелна машина користи метод који се назива *Just In Time* компилација, који служи за рекомпилацију Јава кода у код који може да тумачи машина на којој се код извршава (нпр. инструкције *x86* процесора). Овај метод се другачије назива и *Динамичка рекомпилација*, о којој ће бити речи када се буде причало о разлилитим врстама емулатора.



Слика 2.1.1. *Процес превођења Јава кода који се извршава на виртуелној машини*

2.2 Разлике између емулатора и симулатора

Тема везана за виртуелне машине јесу симулатори. Симулатори постојећег и непостојећег хардвера су дизајнирани и коришћени за скупљање информација о томе како систем ради интерно. Ова врста профилисања је веома тешка и захтевна за имплементацију користећи прави хардвер, због тога се софтверски симулатори корисни, између осталог, за сврхе тестирања. Симулаторима се могу сматрати веома тачни емулатори.

Емулатор је, у неку руку, сличан симулатору, тако да терминологија која се користи често пермутује ова два појма. Начин на који се емулатор разликује од симулатора је тај да је симулација покушај да се постигне исти изглед и корисничко искуство оригиналног производа на другој машини. Та машина се разликује од оригиналне машине за коју је тај производ оригинално дизајниран. Симулатор може

бити потпуно нови програм или „порт“ оригиналног, изворног кода. Реч „порт“ подразумева прилагођавање изворног кода који је написан за једну архитектуру другој архитектури, тако да може да се оствари преносивост тог кода. Емулатор, за разлику од симулатора, је у могућности да репродукује изглед и осећај оригиналног програма, јер то је, на крају крајева, оригинални програм који се помоћу емулатора извршава на другој врсти хардвера.

Пример ове разлике би био када бисмо имали видео игру, дизајнирану за одређену конзолу, где та конзола представља оригинално одредиште за које је игра направљена. Када би програмери који су радили на овој игри одлучили да подрже функционалност да игра може да се покрене на рачунару, искористили би неке делове изворног кода који је коришћен за конзолу, док би остале делове морали да напишу изнова. Тада би нова верзија игре изгледала исто, и имала исто корисничко искуство као игра за конзолу. То представља приступ који би се односио на симулатор. Са емулатором дизајнираним да емулира оригиналну верзију игре за конзолу на рачунару, били бисмо у могућности да покрећемо оригиналну верзију игре, уместо модификовања неких њених делова.

Симулатори за специфичан програм, у општем случају, дају боље перформансе, у поређењу са емулаторима, иако у неким ситуацијама кориснику не може бити јасно која од ова два приступа се користи, због постизања сличних перформанси. Главна предност писања емулатора у односу на писање симулатора је та што је могуће покретати велики број програма који су дизајнирани за тај хардвер и архитектуру, док се симулатор може применити само на један програм. Због тога је јако тежак, ако не и немогућ задатак написати програм који ће се извршавати исто, на различитим системима, без коришћења емулације, или модификације оригиналног изворног кода програма.

2.3 Употреба емулатора

Емулатори се генерално праве за много моћније системе од оних које емулирају. Разлог је тај што емулатори захтевају велику процесорску снагу како би одрадили посао за који је задужен хардвер, само на софтверски начин. Случај када циљани систем не би био моћан као оригинални систем би био када би се правила функционалност новог система, кроз прављење прототипа тог новог система. Тачније, тај случај представља ситуацију када се емулација користи за креирање софтвера за систем за који хардвер још није потпуно развијен. Ово омогућава да се софтвер за

нови систем потпуно тестира, иако хардвер није потпуно завршен. Ово убрзава развијање новог система, зато што је могуће развити добар део софтвера за одредишни уређај, иако у том тренутку хардвер, који би се користио за тестирање, не би био завршен.

Добар пример овога би био то како је компанија *Microsoft* дизајнирала и направила емулатор, који је коришћен за развијање софтвера за *Windows CE* уређај. Корисници би, коришћењем њиховог емулатора, били у могућности да развијају и тестирају *Windows CE* апликацију на *Windows x86* одредишној машини у потпуности, без долажења у додир са *Windows CE* уређајем.

Процес емулације се такође може користити приликом развијања оперативног система. Коришћењем емулатора, много је лакше открити различите хардверске проблеме који се јављају приликом развијања оперативног система. Било би могуће проверити вредности процесорских регистара, као и стање хардвера у остатку система у случају неке грешке, што не би било могуће у случају да се емулатор не користи. Уколико би софтвер који се развија узроковао неки проблем, постојала би могућност да се хардвер на којем се развија закључа, тако да би сва помагала коришћена за дијагностику била изгубљена. У случају да је емулатор коришћен, вероватно је да би пријавио неку врсту упозорења о проблему који је настао, достављајући информације за дебаговање програма самом кориснику.

2.4 Перформансе

Перформансе емулатора представља веома битну тему, поготово за системе који се обрађују у склопу овог рада. Ту се најпре мисли на машине за аркадне видео игре. Машине за аркадне видео игре су тзв. „машине у реалном времену“, што значи да игре које се на њима покрећу директно зависе од тренутног времена. Игра мора да се извршава истом брзином и у емулираном окружењу, као и у оригиналном окружењу, тј. на машини за аркадне игре. Ова чињеница има за последицу то да ако је емулатор превише брз, он мора да се синхронизује са оригиналним тајмингом, а ако је емулатор превише спор, потребно је на неки начин смањити одређена рачунања која се дешавају у систему (нпр. смањити број фрејмова који се приказују на екрану) како би емулатор одржао корак са оригиналном машином.

Што се самих перформанси емулатора тиче, потребно је пажљиво бирати програмске језике у којима ће емулатори бити писани. Због тога што је брзина кључ успешног извршавања једног емулатора, виши програмски језици нису препоручљиви за њихово писање. Две главне варијанте које се истичу су *Асемблер* и *C/C++*. Иако се

C++ може сматрати донекле вишим програмским језиком, то је језик који је веома ефикасан, јер вуче корене од програмског језика *C*, који се сматра за један од најефикаснијих, ако не и најефикаснијим програмским језиком.

Када је *Асемблер* у питању, генерално генерише бржи код. Процесорски регистри који се емулирају се могу користити како би се директно у њима чувале вредности емулираних процесорских регистара. Велики број операционих кодова се могу емулирати користећи сличне операционе кодове од процесора који се емулира. Што се нехативних страна тиче, код који *Асемблер* генерише није преносив, што значи да се он не може извршавати на машини са различитом архитектуром. Такође, јако је тешко и незахвално дебаговати и одржавати асемблерски код, јер он представља најнижи могући ниво апстракције.

Када су језици *C/C++* у питању, код који они генеришу је преносив, што значи да се успешно извршава на различитим рачунарским архитектурама, као и на различитим оперативним системима. Релативно су једноставни за коришћење, дебаговање и одржавање у поређењу са асемблерским кодом. Такође, то је код који се може јако брзо и ефикасно тестирати, што додатно убрзава и побољшава процес имплементације емулатора. Због свега наведеног, језици *C/C++* се узимају као оптимални за имплементацију свих различитих врста емулатора.

2.5 Емулација на ниском и високом нивоу

Емулатори на **ниском нивоу** представљају емулаторе који служе за емулацију хардвера користећи званичну документацију самог хардвера. Већина данас доступних емулатора представљају управо емулаторе на ниском нивоу. Они постављају баријеру између хардвера и кернела оперативног система, тако да у ситуацијама када би било потребно приступити одређеном хардверу, емулатор би пресрео тај позив и сам извршио потребну операцију [8].

Са друге стране, емулација на **високом нивоу** представља хибридну технику која се састоји од хардверске емулације и софтверске симулације. Ова врста емулације је јако корисна у ситуацијама када програми који се емулирају користе дељени статички програмски код. Ова врста емулације је претежно коришћена приликом емулирања оперативних система, где су системски позиви оперативног система симулирани, док док је сам оперативни систем емулиран [7]. То би значило да у ситуацијама када би било потребно одрадити неку операцију оперативног

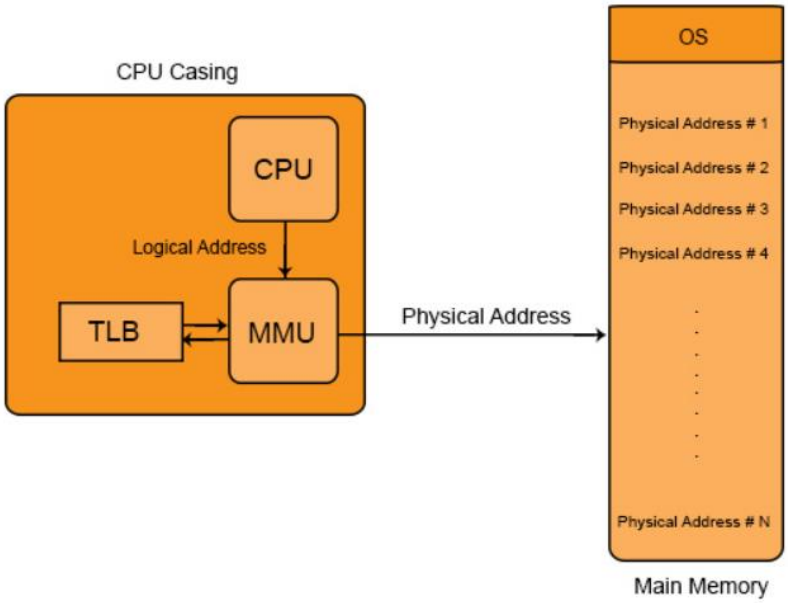
система (нпр. отворити фајл, направити нову нит контроле тока итд.), тај системски позив би био обрађен од стране кода написаног за симулацију [8].

Емулација на високом нивоу је бржа у добијању резултата од емулатора који извршава специфични програм. Такође, лакше је да се извршава брже из простог разлога што је код који се симулира оптимизован за циљни систем, тако да методи нису зависни од тога колико брзо други систем интерно ради. Још једна велика предност емулације на високом нивоу је та када више програма користе исти статишки програмски код како би приступили специјалним чиповима. Када дође до ове ситуације, могуће је емулирати позиве функције без знања о томе како чипови интерно раде. Потребно је само имати у виду за шта је одређена функција предвиђена. Још једна битна ствар за емулацију на високом нивоу је та што нису потребни сви бинарни фајлови како би се одређени програм успешно извршавао. То конкретно значи да би у случају емулације високог нивоа оперативног система било могуће покренути апликацију за тај оперативни систем, без покретања саме инстанце оперативног система. То за последицу има чињеницу да би корисник могао да користи емулирану апликацију за неки оперативни систем, без да заправо има инсталиран тај оперативни систем на својој машини.

Све ово чини емулацију на високом нивоу веома привлачном и корисном у одређеним ситуацијама. Проблем са емулацијом на високом нивоу представља то да ако не постоји статички дељени код између програма, онда сам процес емулације на високом нивоу није могућ. У општем случају, покретање неке специфичне апликације је брже уколико се користи емулација на високом нивоу. Међутим, уколико је потребно покренути већи број апликација у исто време, корисник би увидео проблеме у перформансама, односно велики пад у брзини извршавања тих апликација. Разлог би био тај што у великом броју системских позива, емулирани оперативни систем би морао увек да ради са симулираним системским позивима, што би значило да би други програмски код морао да се извршава за сваки системски позив, тј. не би било могуће искористити код од једног системског позива за други системски позив.

Разлике између емулације на ниском и високом нивоу би било могуће видети на још једном примеру. Тај пример се тиче оперативног система рачунара, тачније **Јединице за управљање меморијом** (енгл. *Memory Management Unit*). Лаички речено, ова јединица служи за то да свакој апликацији додели парче оперативне меморије, тако што мапира адресе из „виртуелног адресног простора“ у адресе из „физичког адресног простора“. Ово је позната техника која се назива техником *Виртуелне меморије,* коришћена како би оперативни систем одао утисак да рачунар поседује више меморије на располагању него што он заправо поседује. Сваки пут када

би апликација затражила приступ меморији неком процесорском инструкцијом, у позадини би Јединица за управљање меморијом транслирала виртуелну адресу у физичку.



Слика 2.5.1. *Процес превођења виртуелне адресе у физичку коришћењем ММУ*

Емулатор на високом нивоу не би морао да брине око Јединице за управњане меморијом гостујуће апликације, будући да када год би гостојућа апликација затражила још меморије, оперативни систем би пресрео тај позив и уделио меморију машине домаћина. Са друге стране, емулатор на ниском нивоу би морао да брине и алокацији меморије за гостујућу апликацују. То би значило да би било потребно алоцирати гостујућу физичку меморију, а потом приликом сваке процесорске инструкције која приступа меморији, било би потребно превести виртуелну гостујућу меморију у физичку. То би изискивало стотинак додатних процесорских инструкција приликом сваког приступа меморији. Финално, то би значило да бисмо за сваки гостојући приступ меморији морали да прођемо кроз четири слоја [8]:

1. Гостујућа виртуелна меморија
2. Гостујућа физичка меморија
3. Домаћинска виртуелна меморија
4. Домаћинска физичка меморија

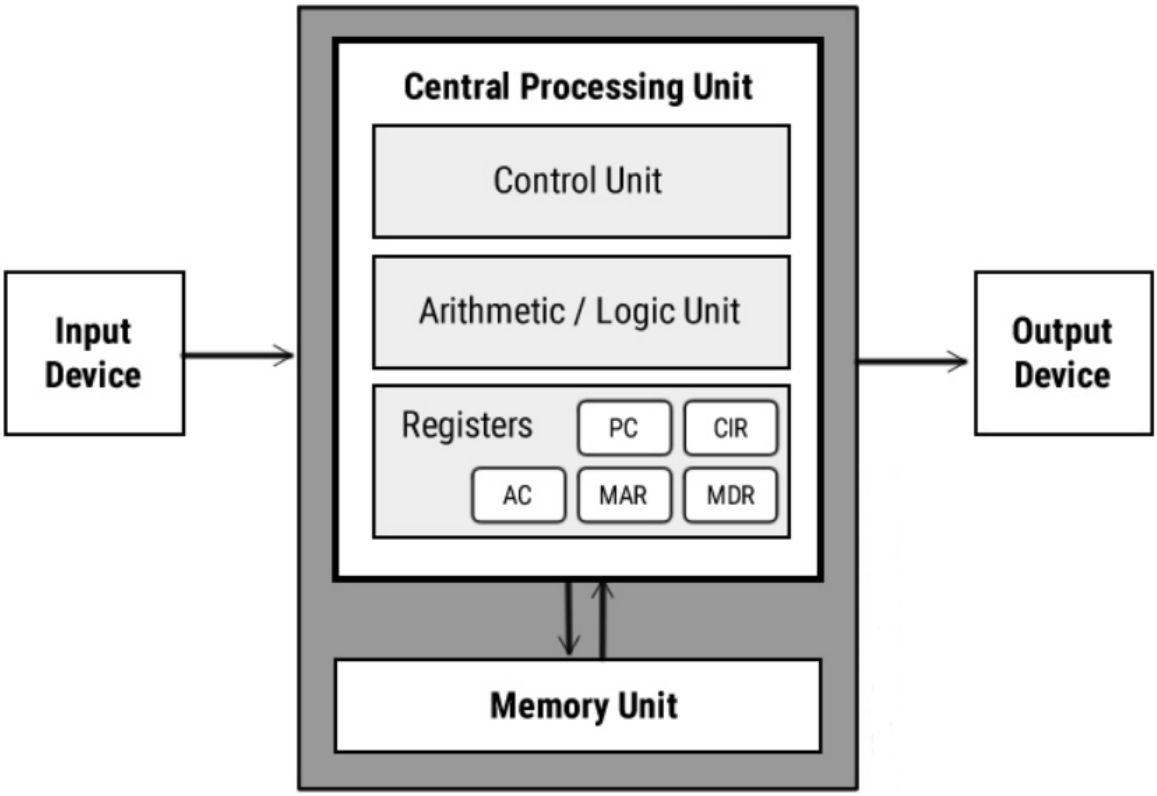
Коначно, долазимо до следећих закључака: емулација на високом нивоу је лакша за покретање и када је оптимизована, може постићи велику брзину чак и на слабијем хардверу. Међутим, то постиже жртвујући аутентичност. Такође, прецизност

емулације на високом и ниском нивоу не може да се пореди. Иако је брзина највећи адут емулације на високом нивоу, та брзина се постиже симулацијом жељеног излаза, а не тачног математичког прорачуна. Код емулатора на ниском нивоу, будући да софтвер покушава да реплицира оригинални хардвер, тај софтвер би требало да ради без багова и различитих врста хакова, које би било потребно применити код емулатора на високом нивоу, у случају да постоји нека грешка која се не може исправити, са потпуно прихватљивим перформансама. Због свега наведеног, одржавање компатибилности и прецизности на емулатору вишег нивоа представља прави изазов на машини за коју се тај софтвер пише [9].

3 Типови емулатора

Емулатор служи за то да покуша да репродукује понашање рачунара коришћењем софтверских програма друге машине. Због тога, емулатор који се дизајнира мора узети у обзир различиту интерну архитектуру рачунара који се емулира.

Сви модерни рачунари су већински засновани на *Фон Нојмановој* архитектури [2]. *Фон Нојманова* архитектура се базира на магистрали на коју су повезани процесор, меморија и улазно-излазни уређаји, заједно са аритметичко-логичком јединицом.



Слика 3.1. *Фон Нојманова архитектура*

Процесор представља део који контролише целу машину, извршава већину рачунских операција у систему. Укратко, то је централни део рачунара, као што сама реч каже (енгл. *Central Processing Unit*), који представља мозак самог рашунара.

Магистрале представљају групе електричних линија које повезују процесор са свим осталим уређајима у рачунарском систему. Типишна магистрала се може поделити на три дела:

- Адресну магистралу
- Магистралу података
- Контролну магистралу

Адресна магистрала је магистрала која преноси информације о томе којем уређају, или којем делу неком уређаја је потребно приступити.

Магистрала података је магистрала која преноси податке од периферних уређаја до процесора, као и од процесора до периферних уређаја.

Контролна магистрала је магистрала која преноси додатне сигнале и контролне информације како би помогла у процесу **арбитрације**. Арбитрације представља процес одлучивања о томе којем уређају ће бити дозвољено да приступи магистрали у одговарајућем тренутку. Магистралу дели више уређаја истовремено, тако да је могуће да само један уређај приступа магистрали у једном тренутку, или два уређаја могу да размењују информације док процесор чека на приступ магистрали.

Меморија представља уређај који служи као примарни извор за перзистенцију и чување података. Постоји неколико типова меморије:

- Меморија која подржава читање података (енгл. *Read Only Memory*) – у наставку текста *ROM*
- Меморија која подржава читање и упис података (енгл. *Random Access Memory*) – у наставку текста *RAM*

Улазно-излазни уређаји служе различитим сврхама, и укључују велики број уређаја. Од секундарних уређаја за одлагање података (хард дискови, ЦД медијуми), преко уређаја за графички приказ или аудио уређаја, мрежних картица, тастатура, мишева, слушалица и сл. Они представљају суштински било коју врсту хардвера којом је могуће управљати помоћу процесора.

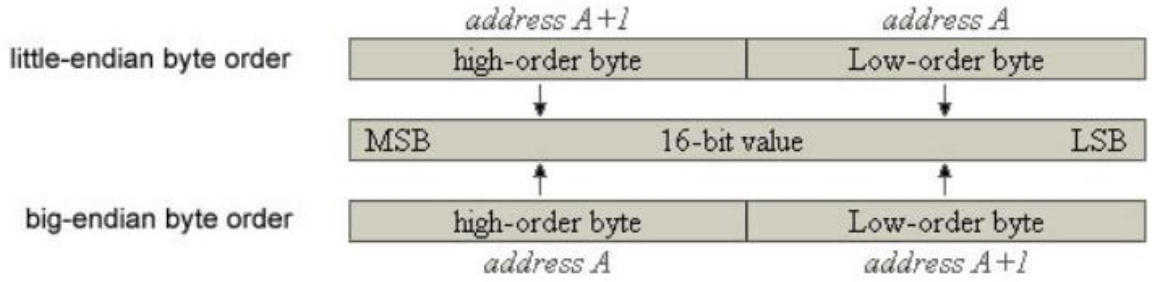
Процесор чита инструкције, тј. операционе кодове из меморије, који говоре процесору да дохвати потребне податке из меморије или одређених процесорских регистара, како би могао да одради одређене неохподне операције. После тога, резултат тих операција се уписује назад у меморију. Процесор приступа улазно-излазним уређајима како би добио информације о спољашњем свету (нпр. који тастер на тастатури је корисник притиснуо), а затим користи ту информацију да би кориснику вратио одређену информацију назад (нпр. приказао притиснути тастер на екрану).

Потребно је приметити да постоје различите врсте меморијске репрезентације у различитим верзијама емулатора [7]. Постоје четири главне алтернативе које се помињу:

- **Главна меморије** – техника која дозвољава директан приступ свим меморијским локацијама
- **Меморија у огледалу** – техника код које постоји више меморијских локација са које је могуће модификовати једну секцију меморије. Када је једна секција меморије модификована, та промена је видљива у свим „клонираним“ меморијским секцијама
- **Заштићена меморија** – техника којом је могуће заштитити одређене делове меморије од уписа, тако да је посебна врста грешке враћена приликом покушаја уписа у заштићене меморијске регионе
- **Страничена меморија** – техника која подразумева постојање више страница у меморијском адресном простору. Да би се добио приступ некој другој меморијској страници, потребно је заменити тренутно учитану страницу са новом страницом

Начин на који се бајтови података учитавају у меморију је веома битан у процесу емулације. Постоје два начина учитавања података, а то су тзв. *„little-endian*“ и *„big-endian“* приступи [7]. Претпоставимо да имамо 16-битни податак, који се састоји од два податка од по 8 бајтова. Тада се *„little-endian*“ односи на складиштење нижег бајта податка на нижу меморијску локацију, док се виши податак складишти на вишу меморијску локацију. Код *„big-endian“* приступа се нижи бајт податка складишти на

вишу меморијску локацију, док се виши бајт података складишти на нижу меморијску локацију.



Слика 3.2. *Меморијска репрезентација различитих техника складиштења података*

Постоје два главна начина за емулирање процесора. Једна од опција је та да дохватамо податке из изворног кода, и да обрађујемо сваки бајт у тзв. *„fetch-decode-execute“* петљи. Ова врста емулације се назива **интерпретативна емулација**, и она представља тему коју ћемо у највише детаља обрађивати у оквиру овог рада. Ово је најједноставнија форма емулације процесора, што за последицу има то да су емулатори овог типа геренално лакши за написати, међутим, пошто се цео програм извршава у оквиру једне петље, интерпретативни емулатори имају слабије и спорије перформансе од конкуренције.

Друга врста емулације изискује то да набавимо изворни код машине домаћина, и потом да тај код преведемо у нови код за процесор на циљаној машини. Овај процес се назива **рекомпилација** или другачије **бинарна транслација** [5]. Транслација може бити статичка или динамичка (повезана са временом извршавања), тако да постоје статичка и динамичка бинарна транслација, односно статичка и динамичка рекомпилација. Због тога што је посао који емулатори раде углавном динамички, што значи да могу да извршавају више програма, динамички приступ је више коришћен.

У наредним поглављима ће детаљно бити обрађени архитектура и особине ових различитих приступа емулације у рачунарским системима.

3.1 Интерпретативни емулатор

Емулација се може поделити у више делова. Емулација процесора се доста разликује од процеса емулације аудио или графичког система. Процесор дохвата бајтове из меморије и извршава инструкције на које се ти операциони кодови односе. Графички систем користи видео меморију и команде које је добио од процесора, како би генерисао слику на екрану корисника. Аудио систем користи податке и инструкције да би изгенерисао електрични сигнал који се после шаље до звучника корисника. Генерално, алгоритми који се користе за емулацију графике или звука немају директне везе са алгоритмом који се користи за емулацију процесора [5].

Главни део емулатора представља петљу, због тога што се начин на који рачунар ради може посматрати као једна петља. Процесор у сваком циклусу емулације извршава *„fetch-decode-execute“* секвенцу, што значи да дохвата инструкцију из меморије, декодује је, а затим је извршава. Како се процесор користи као централна тачка система који се емулира, емулација свих осталих уређаја је условљена самим процесором. Све остале периферије раде у паралели са процесором, тако да процесор извршава инструкције у истом тренутку када и хардвер задужен за графику приказује слику на екрану, или када хардвер за звук репродукује одређени звук. Скица главне петље емулатора, дата у програмском језику *C++*, може се видети на следећој слици:

```
Instruction instruction;
bool isRunning = true;
while (isRunning) {
    instruction = fetch();
    decodeAndExecute(instruction);
    emulateGraphicsAndSound();
    generateInterrupts();
    timeSynchronization();
}
```

Слика 3.1.1. *Петља интерпретативног емулатора*

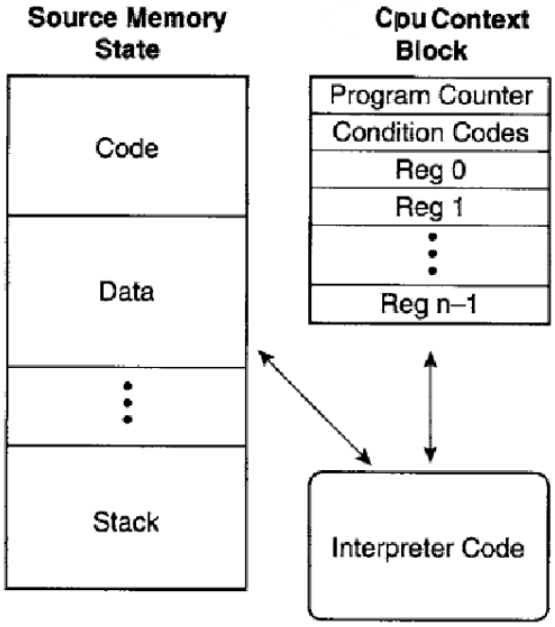
Процесор представља центар емулације емулираног система, тако да се он користи за емулирање времена. Многи рачунари користе неку врсту хардвера како би

водили рачуна о протеклом времену (нпр. тајмери или прекиди везани за тајмере), али главни начин мерења времена се мери у броју извршених процесорских инструкција у оквиру једне временске јединице. Када знамо колико инструкција се може извршити у оквиру једне временске јединице, обично секунде, можемо рећи процесору колико процесорских циклуса треба да изврши пре него што у главној петљи проверимо да ли је одређена временска јединица протекла.

Постоје и периодични таскови који се морају емулирати како би систем коректно функционисао. Ови таскови најчешће обухватају прекиде попут тастатуре, неке друге хардверске периферије или тајмера. Користећи знање које поседујемо о томе како се главна петља емулатора понаша, закључујемо да је потребно периодично проверавати услове окидања неког од прекида. Сви прекиди се подразумевано морају десити у другој нити контроле, у односу на нит у којој се извршава главна петља емулатора, из очигледних разлога. У главној петљи емулатора, испитују се услови за генерисање прекида, и ако су они испуњени, јавља се процесору, који би онда у складу са тим обрадио прекиде на одређени начин. На слици ***3.2.***, обрада прекида се дешава у методи `generateInterrupts()`, док се обрада времена дешава у методи `timeSynchronization()`.

Треба нагласити да класа `Interrupt` са слике ***3.2.*** представља класу која енкапсулира све потребне операције над инструкцијама, попут чувања информација о самој инструкцији, као што су дужина инструкције, начини адресирања операнада или број операнада.

Емулаторова меморија садржи слику меморије емулиране машине, укључујући програм и податке. Емулатор такође у меморији одржава контекст емулираног процесора који обухвата све елементе архитектуре: регистре опште намене, програмски бројач, статусне и разне контролне регистре. На слици ***3.4.*** дата је скица генералне структуре интерпретативног емулатора, која описује горепоменуте елементе.



Слика 3.1.2. *Скица генералне структуре интерпретативног емулатора*

Као што је раније поменуто, интерпретативна емулација представља најлакши начин емулације процесора. Процесор чита бајтове података из меморије на коју показује посебан програмски регистар, звани *PC* (енгл. *Program Counter*). Ти бајтови података садрже информације о томе шта процесор тачно треба да уради, односно коју инструкцију треба да изврши. Процесор, после дохватања бајтова из меморије, треба ту инструкцију да декодује, а потом да изврши саму инструкцију. У зависности од саме инструкције и врсте емулатора, могуће је да извршавање инструкције може да утиче на посебан регистар, звани *PSW* (енгл. *Program Status Word*). Регистар садржи информације о свим прекидима у систему, тако да сваки бит тог регистра одговара посебном прекиду.

Потребно је приметити и да код већине процесора, PC увек указује на прву наредну инструкцију која треба да се изврши. То можемо видети и на слици ***3.5.***, где се налази програмски код `fetch()` наредбе.

```
void Emulator::fetch() {
    instruction = Instruction::readMemory(PC);
    PC += instruction.size();
}
```

Слика 3.1.3. *Скица дохватања податка из меморије и ажурирања регистра PC*

После процеса дохватања податка и ажурирања програмског бројача, потребно је декодовати инструкцију, а потом је и извршити. Процес декодовања и извршавања се обично имплементира коришћењем *„switch-case“* програмског конструкта, где се за сваки операциони код може извршити потребна операција.

```
void Emulator::decodeAndExecute(Instruction instruction) {
    OpCode opCode = Emulator::decode(instruction);
    switch (opCode) {
        case OPCODE1:
            executeOpCode1();
            break;
        …
    }
}
```

Слика 3.1.4. *Скица декодовања и извршавања инструкције*

Интерпретативни емулатор је спорији у поређењу са осталим типовима емулатора [5]. Спорост се може приписати безусловном декодовању и извршавању инструкција. Такође, машина која емулира циљану машину треба да буде бржа од ње бар за два реда величине [6]. Главна врлине овог емулатора су једноставност, портабилност и прецизност. Постоје одређене врсте оптимизација код емулатора који користе технику рекомпилације, које за циљ имају да побољшају перформансе интерпретативног емулатора кеширањем инструкција које се више пута извршавају.

3.2 Рекомпилација

Много брже од декодовања и извршавања сваке инструкције безусловно би било превођење операционог кода на операциони код циљне машине. То представља геренални циљ процеса рекомпилације.

Рекомпилација представља технику где машински код емулиране машине преводи на машински код одредишне машине. Преведени код се чува у меморији и извршава сваки пут када га емулирани процесор позове.

Ово превођење се ради у блоковима кода, некада повезаним са концептом *базичних блокова* у теорији програмских преводилаца, а некада не. Разлога за рад са овим блоковима има доста, од чињенице да сав код не може бити познат одмах, до чињенице мора постојати тачка где се емулација може зауставити да би се олакшао процес превођења, који је свакако бржи у случају рада са блоковима кода.

Постоје два начина превођења кода: **статички** и **динамички**. Статичка рекомпилација подразумева то да је пре покретања емулације цео програм већ преведен на машински код циљане машине. Превођење се обавља динамички када се код за емулирани процесор преводи у оном тренутку када се и извршава, тј „у лету“ [6]. Процес рекомпилације се може поделити на два дела, фазу превођења и фазу извршавања. У фази превођења код се са процесора који се емулира преводи у код за одредишни процесор. У фази извршавања, код који је преведен се извршава од стране циљног процесора.

Процес рекомпилације може бити спорији у поређењу са чистим извршавањем, односно интерпретирањем кода, али само у случају када би се код извршио само једном, односно у једној итерацији. То генерално није случај код емулатора, пошто се код емулатора извршава у петљи, често извршавајући поново исте инструкције и блокове кода. Транслација може бити много бржа од интерпретирања због тога што је могуће искористити ресурсе одредишне машине, тако да се декодовање сваке инструкције дешава само једном. То за последицу има то да је могуће, у случају препознавања блока кода приликом емулације, директно извршити тај код на одредишној машини, без потребе за емулаторском петљом.

Иако процес бинарне транслације, односно рекомпилације, у теорији има велике предности у брзини у односу на емулацију, проблем је што процес рекомпилације може бити веома тежак и изазован, и може захтевати познавање разних концепата везаних за писање програмских преводилаца. Такође, рекомпилација има проблеме које интерпретативни емулатор нема, а ти проблеми се тичу кода који се сам мења

(енгл. *self-modifying code*). Статички рекомпилатор не би могао да детектује код који се сам мења, из разлога што се тај код мења у време извршавања, а статичка рекомпилација је процес који се дешава пре тога. Иако би динамички рекомпилатор могао да препозна меморијске регионе који садрже код који се сам мења, процес детекције, модификације и поновног превођења тог кода би могао да уведе огромне деградације у перформансама емулатора заснованог на динамичкој рекомпилацији [3]. Такође, као проблем се намеће и преносивост. Рекомпилација се користи за изворни процесор, и за одредишни процесор, тако да се може десити да емулатор који се извршава на једној архитектури неће радити на другој архитектури. Добра страна ове технике је то што је могуће постићи десет пута бржу емулацију, у поређењу са интерпретативним емулатором, међутим брзина може зависити од изворишног и одредишног процесора, као и од тога како су интерпретативни емулатор и рекомпилатор имплементирани.

Закључак који би се могао извући приликом поређења различитих врста емулатора је следећи: када год перформансе одредишне машине дозвољавају, прихватљиво је определити се за интерпретативну емулацију, поготово што интерпретативна емулација најаутентичније емулира циљани систем. Са друге стране, ако је брзина емулације приоритет, и ако је ако тежина имплементације није толико битан фактор, као и ако није битно да ли је код преносив, техника рекомпилације је бољи избор.

4 Опис CHIP-8 система

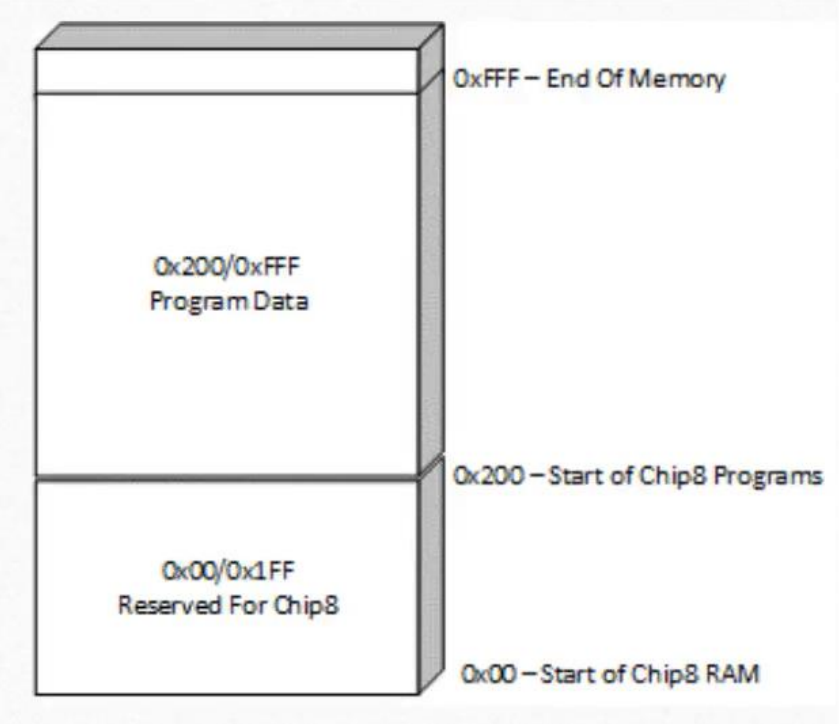
Као што је већ поменуто у претходном делу рада, *CHIP-8* представља интерпретирани програмски језик, развијен почетком седамдесетих година прошлог века. Осмишљен је као програмски језик који ће се користити за лакше прављење програма и игара за рачунаре. У овом поглављу ће детаљно бити описана архитектура самог система [10], као и инструкцијски сет који овај систем користи.

4.1 Архитектура CHIP-8 система

CHIP-8 систем представља систем који располаже са 4KB меморије. То значи да у овом систему постоји 4096 меморијских локација, од којих је свака меморијска локација бајт адресибилна. Према томе, доступне меморијске локације налазе се у опсегу *0x000-0xFFF*. Подаци се у меморији смештају по *„big-endian“* формату. Осим поменуте меморије, овај систем се састоји још од различитих врста регистара, тајмера, дисплеја и тастатуре, о којима ће бити реч у наставку овог рада.

Адресни простор *CHIP-8* система је подељен на два сегмента:

- ***0x000 - 0x1FF***: део меморије резервисан за иницијалну имплементацију *CHIP-8* интерпретера који је био директно мапиран у меморију. Кориснички програми не би требало да користе ове меморијске локације. Локације 0x000-0x050 су резервисане за учитавање фонта који служи за приказ различитих карактера емулатора на екрану у случају модерне имплементације емулатора. Остатак меморијских локација у овом опсегу није дозвољено користити ни за читање, ни за упис.
- ***0x200 - 0xFFF***: део меморије резервисан за учитавање корисничких програма, остатак простора који остане након учитавања корисничког програма остаје слободан за коришћење. Све корисничке инструкције учитане у овај сегмент меморије, учитавају се из тзв. *ROM* (енгл. *Read-Only Memory*) фајлова, који представљају копију картриџа за видео игре.



Слика 4.1.1. *Приказ адресног простора CHIP-8 емулатора*

Осим меморије, овај рачунарски систем садржи и 16 осмобитних регистара, који се називају **V0-VF**. Уз име сваког регистра се додаје као суфикс редни број од 0 до 15, односно од 0 до F, пошто F у хексадецималном запису представља број 15. Регистар представља поцебну меморијску локацију на процесору. Свака операција коју процесор одради, одрађује се унутар процесорских регистара, због тога што су они веома брзи. Операције над регистрима обично обухватају учитавање вредности у регистар из меморије, као и упис вредности из регистра у меморију.

Регистар **VF** се користи и као тзв. „*flag*“ регистар, односно служи за то да се у њему памти нека одређена информација. Због тога што се овај регистар користи како би се у њему памтила информација о томе да ли се нека акција догодила, није дозвољено да буде коришћен од стране корисничког програма.

Поред 16 регистара опште намене, од периферија постоје још неколико регистара који се користе, различите врсте тајмера, као и тастатура и графички дисплеј:

- Регистар **I** – шеснаестобитни индексни регистар, који се користи услед неких рачунских операција са адресама. Величине је шеснаест бита, из разлога што највећа могућа адреса у овом систему *0xFFF*, односно величине дванаест бита.
- Регистар **PC** - шеснаестобитни регистар који указује на следећу инструкцију коју је потребно извршити. Операциони код је величине шеснаест бита, тако да када дохватамо инструкцију, потребно је дохватити по један бајт са адреса *PC* и *PC+1*, а затим увећати вредност регистра за 2, како би поново указивао на прву следећу инструкцију коју је потребно извршити.
- **Delay Timer** – тајмер који служи да се декрементира по фреквенцији од 60Hz, односно декрементира се сваке секунде, све док му вредност не опадне на нулу.
- **Sound Timer** – тајмер који се, такође, декрементира по фреквенцији од 60Hz, односно сваке секунде, само је разлика у томе да све док има ненулту вредност, овај тајмер генерише звук. Када вредност падне на нулу, звучни сигнал овог тајмера се деактивира.
- **Стек** – структура података која служи за чување шеснаестобитних адреса. Користи се приликом инструкција скока и потпрограма, како би могла да се сачува повратна вредност на коју је потребно вратити се при завршетку операције.
- **Тастатура** – хексадецимална тастатура која се састоји од шеснаест тастера, која имају вредности 0-F. Потребно је да систем буде интерактиван, тако да може да реагује на притиске одређених тастера на тастатури.
- **Дисплеј** – монохроматски дисплеј, димензија 64 пиксела у дужини и 32 пиксела у висини. Графика која се генерише се представља у облику елемената који се на енглеском називају ***„sprite“***. Елементи представљају репрезентацију из меморије пиксела који се штампају на екрану. Сваки елемент заузима пет бајтова, од којих сваки бајт представља један ред који ће се исцртати. Оваква репрезентација фонта се приликом почетка извршавања програма учитава у меморију на адресе *0x000-0x050*.

"0"	Binary	Hex
****	11110000	0xF0
* *	10010000	0x90
* *	10010000	0x90
* *	10010000	0x90
****	11110000	0xF0

"1"	Binary	Hex
*	00100000	0x20
**	01100000	0x60
*	00100000	0x20
*	00100000	0x20
***	01110000	0x70

"2"	Binary	Hex
****	11110000	0xF0
*	00010000	0x10
****	11110000	0xF0
*	10000000	0x80
****	11110000	0xF0

"3"	Binary	Hex
****	11110000	0xF0
*	00010000	0x10
****	11110000	0xF0
*	00010000	0x10
****	11110000	0xF0

"4"	Binary	Hex
* *	10010000	0x90
* *	10010000	0x90
****	11110000	0xF0
*	00010000	0x10
*	00010000	0x10

"5"	Binary	Hex
****	11110000	0xF0
*	10000000	0x80
****	11110000	0xF0
*	00010000	0x10
****	11110000	0xF0

"6"	Binary	Hex
****	11110000	0xF0
*	10000000	0x80
****	11110000	0xF0
* *	10010000	0x90
****	11110000	0xF0

"7"	Binary	Hex
****	11110000	0xF0
*	00010000	0x10
*	00100000	0x20
*	01000000	0x40
*	01000000	0x40

"8"	Binary	Hex
****	11110000	0xF0
* *	10010000	0x90
****	11110000	0xF0
* *	10010000	0x90
****	11110000	0xF0

"9"	Binary	Hex
****	11110000	0xF0
* *	10010000	0x90
****	11110000	0xF0
*	00010000	0x10
****	11110000	0xF0

"A"	Binary	Hex
****	11110000	0xF0
* *	10010000	0x90
****	11110000	0xF0
* *	10010000	0x90
* *	10010000	0x90

"B"	Binary	Hex
***	11100000	0xE0
* *	10010000	0x90
***	11100000	0xE0
* *	10010000	0x90
***	11100000	0xE0

"C"	Binary	Hex
****	11110000	0xF0
*	10000000	0x80
*	10000000	0x80
*	10000000	0x80
****	11110000	0xF0

"D"	Binary	Hex
***	11100000	0xE0
* *	10010000	0x90
* *	10010000	0x90
* *	10010000	0x90
***	11100000	0xE0

"E"	Binary	Hex
****	11110000	0xF0
*	10000000	0x80
****	11110000	0xF0
*	10000000	0x80
****	11110000	0xF0

"F"	Binary	Hex
****	11110000	0xF0
*	10000000	0x80
****	11110000	0xF0
*	10000000	0x80
*	10000000	0x80

Слика 4.1.2. *Репрезентација елемената који се приказују на екрану у виду матрице пиксела*

4.2 Инструкцијски сет CHIP-8 система

Инструкцијски сет *CHIP-8* емулатора се састоји од 35 различитих инструкција, од којих је свака инструкција величине 2 бајта. Као што је претходно поменуто, бајтови инструкције се у меморији чувају у *„big-endian“* формату. Операциони кодови су представљени и описани у табели испод, у хексадецималном формату. Детаљни опис инструкција биће дат у наредном поглављу, када буде било речи о имплементацији.

У оквиру операционих кодова датих у табели, потребно је описати одређене карактере који се јављају [11]:

- *nnn* или *addr*: нижих дванаест бита инструкције
- *n* или *nibble*: нижих четири бита инструкције
- *x*: нижих четири бита вишег бајта инструкције
- *y*: виших четири бита нижег бајта инструкције
- *kk* или *byte*: нижих осам бита инструкције

Табела 4.2.1. *Приказ инструкцијског сета од 35 инструкција CHIP-8 емулатора*

Операциони код	***Инструкција***	***Опис инструкције***
00E0	*CLS*	Очисти дисплеј
00EE	*RET*	Повратак из потптограма
0nnn	*SYS addr*	Скок на машинску рутину на адреси *nnn,* некоришћено у модерним имплементацијама
1nnn	*JP addr*	Скок на адресу *nnn*
2nnn	*CALL addr*	Позив потпрограма на адреси *nnn*
3xkk	*Se Vx, byte*	Прескочи сл. инструкцију ако је *Vx == kk*
4xkk	*SNE Vx, byte*	Прескочи сл. инструкцију ако је *Vx != kk*
5xy0	*SE Vx, Vy*	Прескочи сл. инструкцију ако је *Vx == Vy*
6xkk	*LD Vx, byte*	Учитај *кк* у *Vx*
7xkk	*ADD Vx, byte*	У *Vx* учитај *Vx + kk*

Операциони код	***Инструкција***	***Опис инструкције***
8xy0	*LD Vx, Vy*	У *Vx* учитај *Vy*
8xy1	*OR Vx, Vy*	У *Vx* учитај *Vx OR Vy*
8xy2	*AND Vx, Vy*	У *Vx* учитај *Vx AND Vy*
8xy3	*XOR Vx, Vy*	У *Vx* учитај *Vx XOR Vy*
8xy4	*ADD Vx, Vy*	У *Vx* учитај *Vx ADD Vy*
8xy5	*SUB Vx, Vy*	У *Vx* учитај *Vx SUB Vy*
8xy6	*SHR Vx {, Vy}*	У *Vx* учитај *Vx SHR 1*
8xy7	*SUBN Vx, Vy*	У *Vx* учитај *Vy SUB Vx,* у *VF* учитај *1* ако је *Vy > Vx*
8xyE	*SHL Vx {, Vy}*	У *Vx* учитај *Vx SHR 1*
9xy0	*SNE Vx, Vy*	Прескочи сл. инструкцију ако је *Vx != Vy*
Annn	*LD I, addr*	У *I* учитај *nnn*
Bnnn	*JP V0, addr*	Скок на адресу *nnn + V0*
Cxkk	*RND Vx, byte*	У *Vx random byte Vx AND kk*
Dxyn	*DRW Vx, Vy, nibble*	На координатама *(Vx, Vy)* на екрану исцртај *„sprite“* узет са меморијске локације *I*, у *VF* учитај *collision*
Ex9E	*SKP Vx*	Прескочи сл. инструкцију ако је дугме са вредношћу *Vx* притиснуто
ExA1	*SKNP Vx*	Прескочи сл. инструкцију ако дугме са вредношћу *Vx* није притиснуто
Fx07	*LD Vx, DT*	Учитај *DT* у *Vx*
Fx0A	*LD Vx, K*	Чекај на притицак дугмета, учитај вредност дугмета у *Vx*
Fx15	*LD DT, Vx*	Учитај *Vx* у *DT*
Fx18	*LD ST, Vx*	Учитај *Vx* у *ST*
Fx1E	*ADD I, Vx*	У *I* учитај *Vx ADD I*
Fx29	*LD F, Vx*	У *I* учитај локацију за „*sprite*“ који одговара вредности *Vx*

Операциони код	***Инструкција***	***Опис инструкције***
Fx33	*LD B, Vx*	На меморијске *локације I, I + 1, I + 2* учитај *BCD* вредности *Vx*
Fx55	*LD [I], Vx*	Учитај вредности регистара *V0 – Vx* у меморију почевши од локације *I*
Fx65	*LD Vx, [I]*	Прочитај вредности регистара *V0 – Vx* у меморију почевши од локације *I*

5 Имплементација

Имплементација емулатора обухвата 2 класе: `CHIP8CPU` и `CHIP8Menu`. Класа `CHIP8CPU` служи да енкапсулира све потребне податке и логику везане за сам процесор који се емулира. Класа `CHIP8Menu` служи да имплементира логику везану за приказ менија, као и да имплементира основне операције система попут прелажења са једног прозора на други, покретања неке од игара, паљења или гашења звука. Класа `CHIP8Menu` садржи класу `CHIP8CPU`, односно ради се о релацији агрегације између две класе.

У оквиру поглавља о имплементацији биће дати само битнији детаљи имплеметације класа и метода, док ће сегменти кода који нису од суштинске важности бити изостављени. Приликом имплементације на оперативном систему *Windows* коришћен је *MinGW* компајлер заједно са *CLion* развојним окружењем. Такође, коришћена је и *SDL* библиотека за графику.

Основни изглед класе `CHIP8CPU` је приказан на следећој слици. Осим главних компоненти које су већ поменуте у поглављу о архитектури попут регистара, стека и периферија треба издвојити методу `CHIP8CPU::emulate()` која заправо служи за иницијализацију целог система, учитавање *ROM* фајлова са видео игрима и покретање главне петље емулатора.

```
class Chip8CPU {
public:
    /* Emulator loop */
    void emulate(const std::string& path, const SDL_Color& color,
uint8_t beep_duration, uint8_t sleep_duration);

private:
    /* Display screen */
    std::array<std::array<bool, DISPLAY_WIDTH>, DISPLAY_HEIGHT>
display;
    /* RAM memory */
    std::array<uint8_t, MEMORY_SIZE> memory;
    /* Stack */
    std::array<uint16_t, STACK_SIZE> stack;
    /* Register bank */
    std::array<uint8_t, REGISTER_NUMBER> V;
    /* Keyboard */
    std::array<bool, NUMBER_OF_KEYS> keyboard;
```

```
    /* PC */
    uint16_t PC;
    /* SP */
    uint8_t SP;
    /* Instruction register */
    uint16_t I;
    /* Timers */
    uint8_t DT, ST;
    /* Emulator state */
    bool running;
};
```

Слика 5.1. *Скица костура класе Chip8CPU*

Метода `CHIP8CPU::emulate()` задужена је за иницијализацију система, односно свих регистара на подразумеване вредности. Поред тога, задужена је за учитавање фонта и улазних датотека, тј. *ROM* фајлова. На крају, задужена је за покретање главне петље емулатора и извршавање инструкција у петљи. На сликама ***5.2***. и ***5.3***. можемо видети процес учитавања фонтова и *ROM* фајлова директно у меморију. Треба приметити да је мапа коришћена приликом учитавања фонтова иницијализована вредностима приказаним на слици ***4.1.2.***

```
    static constexpr std::array<uint8_t, FONT_SIZE> font_map = {
            /* 0 */
            0xF0, 0x90, 0x90, 0x90, 0xF0,
            /* 1 */
            0x20, 0x60, 0x20, 0x20, 0x70,
            /* 2 */
            0xF0, 0x10, 0xF0, 0x80, 0xF0,
            /* 3 */
            0xF0, 0x10, 0xF0, 0x10, 0xF0,
            /* 4 */
            0x90, 0x90, 0xF0, 0x10, 0x10,
            /* 5 */
            0xF0, 0x80, 0xF0, 0x10, 0xF0,
            /* 6 */
            0xF0, 0x80, 0xF0, 0x90, 0xF0,
            /* 7 */
            0xF0, 0x10, 0x20, 0x40, 0x40,
            /* 8 */
            0xF0, 0x90, 0xF0, 0x90, 0xF0,
            /* 9 */
            0xF0, 0x90, 0xF0, 0x10, 0xF0,
            /* A */
            0xF0, 0x90, 0xF0, 0x90, 0x90,
            /* B */
            0xE0, 0x90, 0xE0, 0x90, 0xE0,
```

```
            /* C */
            0xF0, 0x80, 0x80, 0x80, 0xF0,
            /* D */
            0xE0, 0x90, 0x90, 0x90, 0xE0,
            /* E */
            0xF0, 0x80, 0xF0, 0x80, 0xF0,
            /* F */
            0xF0, 0x80, 0xF0, 0x80, 0x80
    };

inline void Chip8CPU::load_font() {
    for (uint32_t i = FONT_START; i < font_map.size(); i++) {
        memory[i] = font_map[i];
    }
}
```

Слика 5.2. *Процес учитавања фонта у меморију*

Треба приметити коришћење методе `Chip8CPU::write_mem()` на слици ***5.3.*** Она представља омотач око директног уписа у меморију, који служи да провери да ли је индекс на који се уписује у дозвољеном опсегу. У случају да није, биће бачена грешка у време извршавања, а програм ће бити заустављен.

```
void Chip8CPU::load_rom(const std::string& path) {
    std::ifstream file(path, std::ios::binary | std::ios::in);

    if (!file.is_open()) {
        throw std::runtime_error(
                    "Cannot open ROM file for reading.");
    }

    std::for_each(
            std::istreambuf_iterator<char>(file),
            std::istreambuf_iterator<char>(),
            [this, index = PROGRAM_START](const uint8_t& c)
                    mutable {
                write_mem(c, index++);
            });
}
```

Слика 5.3. *Процес учитавања ROM фајла у меморију*

Главни део методе `CHIP8CPU::emulate()` представља метода `CHIP8CPU::run(const SDL& color)` која обухвата петљу емулатора. Такође, она поседује и интеграцију са *SDL* библиотеком која је коришћена за графику.

Главни ентитети *SDL* библиотеке који су коришћени су *SDL_Window* и *SDL_Rendered. SDL_Window* представља структуру која садржи све информације о прозору који се исцртава на екрану, попут његових димензија, позиције, обода итд. *SDL_Rendered* је структура задужена за рендеровање, односно приказивање на екрану свих ствари које су повезане са прозором који смо специфицирали. Ова структура ради по принципу исцртавања различитих облика на екрану, који су пре тога обојени у жељену боју. У овом случају, коришћена је структура *SDL_Rect*, односно правоугаоник који представља један пиксел на екрану. Према томе, сваким позивом методе `CHIP8CPU::render()` на екрану бива приказан нови пиксел који није био приказан раније. Из разлога што се ово рендеровање дешава у петљи, у свакој итерацији добијамо другачији приказ пиксела на екрану, у зависности од тога како су они промењени.

```
void Chip8CPU::run(const SDL_Color& color) {
    bool pause {false};

    SDL_Window *window;
    SDL_Renderer *renderer;

    sdl_initialize(&window, &renderer);

    while (running) {
        sdl_poll_events(pause);

        if (!pause) {
            sdl_render(renderer, color);
            timer_tick();
            auto opcode = decode();
            fetch();
            execute(opcode);
        }
    }

    sdl_restore(&window, &renderer);
}
```

Слика 5.4. *Приказ главне петље емулатора*

Методе `Chip8CPU::sdl_initialize()` и `Chip8CPU::sdl_restore()` нису од великог значаја, јер оне представљају обичне методе у којима се ради иницијализација и уништавање *SDL* објеката. Од много већег значаја су методе `Chip8CPU::sdl_poll_events()` и `Chip8CPU::sdl_render()`, о којима ће бити речи у наставку рада.

Метода `Chip8CPU::sdl_render()` представљена је на слици ***5.5.*** У телу методе можемо видети две петље које се крећу дуж целог прозора, односно за сваки пиксел који се налази у опсегу дужине и ширине прозора проверавамо да ли има активну вредност, односно да ли га је резултат неке претходне инструкције означио да треба да се исцрта. У случају да јесте, правимо нову инстанцу *SDL_Rect* објекта, а затим га исцртавамо на екрану. На крају извршавања ове методе, промене за све пикселе ће бити видљиве.

Треба приметити да се све операције са пикселима дешавају над матрицом елемената типа `bool`. Према томе, свако „паљење“ или „гашење“ одговарајућег пиксела се реализује само уписивањем вредности `true` или `false` у одговарајући елемент матрице.

```
void Chip8CPU::sdl_render(SDL_Renderer *renderer, const SDL_Color&
 color) {
    SDL_SetRenderDrawColor(renderer, 0, 0, 0, 0);
    SDL_RenderClear(renderer);
    SDL_SetRenderDrawColor(renderer, color.r, color.g, color.b, 0)
;

    for (auto x = 0; x < Chip8CPU::DISPLAY_WIDTH; x++) {
        for (auto y = 0; y < Chip8CPU::DISPLAY_HEIGHT; y++) {
            if (is_pixel_set(x, y)) {
                SDL_Rect sdl_rect;
                sdl_rect.x = x * DISPLAY_PIXEL_SCALE;
                sdl_rect.y = y * DISPLAY_PIXEL_SCALE;
                sdl_rect.w = DISPLAY_PIXEL_SCALE;
                sdl_rect.h = DISPLAY_PIXEL_SCALE;
                SDL_RenderFillRect(renderer, &sdl_rect);
            }
        }
    }
    SDL_RenderPresent(renderer);
}
```

Слика 5.5. *Приказ исцртавања пиксела на екрану*

Метода `Chip8CPU::sdl_poll_events()` служи за то да реагује на корисников унос са тастатуре. Цела метода се такође извршава у петљи. Идеја је да се обраде сви пристигли догађаји када се корисник нађе у оквиру ове петље. Метода *SDL_PollEvent()* представља препоручени начин рада са системским догађајима, зато што се главна нит не блокира приликом чекања на неки догађај, већ се догађај, уколико га има у реду за чекање обради, а потом се контрола тога враћа програму који излази из *while* петље и секвенцијално наставља своје извршавање. Општа пракса је

да се потпуно обради ред за чекање догађаја у сваком фрејму, обично на почетку методе, пре ажурирања стања самог емулатора.

У овој конкретној имплементацији, систем може реаговати на различите уносе са тастатуре. У случају притиска тастера *Escape* или у случају притиска тастера *X* у горњем углу прозора, извршавање се прекида и емулатор се зауставља, заједно са уништавањем свих прозора који постоје. У случају притиска тастера *Space*, извршавање се привремено суспендује до следећег притиска истог тастера. Наравно, омогућено је и препознавање тога да ли је тастер притиснут или отпуштен, што представља фундаментални механизам у емулирању неке видео игре, где је потребно одрадити одређену акцију у зависности од тога да ли је тастер притиснут или отпуштен.

Треба приметити да, налик на рад са пикселима, рад са тастерима се симулира на сличан начин, само што се уместо матрице елемената типа `bool` користи низ. Низ се ажурира на исти начин као што се ажурирају и пиксели.

```
void Chip8CPU::sdl_poll_events(bool& pause) {
    SDL_Event event;

    while (SDL_PollEvent(&event)) {
        switch (event.type) {
            case SDL_QUIT:
                running = false;
                break;
            case SDL_KEYUP: {
                auto key = get_keyboard_mapping_value(
                              event.key.keysym.sym);

                if (key != KEY_NOT_FOUND) {
                    key_release(key);
                }
            }
                break;
            case SDL_KEYDOWN: {
                auto key = get_keyboard_mapping_value(
                              event.key.keysym.sym);

                if (key != KEY_NOT_FOUND) {
                    key_press(key);
                }

                auto& scancode = event.key.keysym.scancode;

                if (scancode == SDL_SCANCODE_SPACE) {
```

```
                    pause = !pause;
                }
                else if (scancode == SDL_SCANCODE_ESCAPE) {
                    running = false;
                }
            }
                break;
        }
    }
}
```

Слика 5.6. *Метода Chip8CPU::sdl_poll_events()*

Метода `Chip8CPU::timer_tick()` представља ништа више него методу у којој се ажурирају вредности тајмера *DT* и *ST*. Ова метода се извршава после сваке итерације окружујуће петље, односно после сваке извршене инструкције. У свакој итерацији декрементирају се вредности тајмера, и у случају да је њихова вредност пала на нулу, долази до успављивања главне нити на неко време, што у ствари симулира неку временску зардшку, или до генерицања звучног сигнала, у случају тајмера *ST*.

```
void Chip8CPU::timer_tick() {
    if (DT > 0) {
        Sleep(sleep_duration);
        DT--;
    }
    if (ST > 0) {
        Beep(BEEP_FREQUENCY, beep_duration * ST);
        ST--;
    }
}
```

Слика 5.7. *Метода Chip8CPU::timer_tick()*

Метода `Chip8CPU::execute()` имплементирана је по угледу на ону са слике ***3.1.4.*** Коришћена је велика *switch* наредба, у којој су покривене све инструкције дефинисане инструкцијским сетом. За сваку инструкцију постоји операциони код, на основу којег се бира одређена грана *switch* наредбе која ће се извршити. Од занимљивијих инструкција, треба издвојити три главне, а то су инструкције са кодовима:

- ***Dxyn*** *-> DRW Vx, Vy, nibble*
- ***Fx0A*** *-> LD Vx, K*
- ***Fx33*** *-> LD B, Vx*

Инструкција *DRW Vx, Vy, nibble* дата је на слици ***5.8.*** Приликом обраде ове инструкције, позива се метода `Chip8CPU::draw_sprite()` која ће бити приказана на слици ***5.9.***

```
case 0xD000: {
            auto x = get_third_nibble(opcode);
            auto y = get_second_nibble(opcode);
            auto n = get_first_nibble(opcode);
            auto f = 0xF;

            V[f] = draw_sprite(V[x], V[y], n, memory[I]) ? 1 : 0;
        }
            break;
```

Слика 5.8. *Имплементација инструкције DRW Vx, Vy, nibble*

Метода `Chip8CPU::draw_sprite()` користи се за рачунање свих пиксела који ће се исцртати на екрану, као и оних који су били исцртани, а више не треба да буду.

Будући да је операција која се користи приликом рада са пикселима ексклузивно или, то значи да у случају да желимо да нацртамо пиксел одређене величине на екрану, морамо да извршимо еклклузивно или између координата на којима се налази пиксел, са истим тим координатама и њиховим вредностима које су већ запамћене у матрици пиксела. У случају да је за неку координату ова операција вратила *0*, потребно је да поставимо вредност регистра ***VF*** на *1*. Укратко речено, брисањем пиксела који је претходно био активан, постављамо вредност регистра *VF*. Пиксел одређене величине се чита из регистра ***I***, и он треба да представља један од карактера који су приказани на слици ***3.1.4.***, односно представља графичку репрпезентацију података величине пет бајтова. У случају да величина пиксела који цртамо на екрану излази ван опсега матрице пиксела, потребно је да остатак пиксела транслирамо на другу страну, како би могао цео да буде приказан. То се постиже коришћењем модуларне аритметике.

```
bool Chip8CPU::draw_sprite(uint8_t x, uint8_t y, uint8_t count,
uint8_t index) {
    bool collision{false};

    const auto *sprite = static_cast<const uint8_t *>(&index);

    for (auto y_offset = 0; y_offset < count; y_offset++) {
        for (auto x_offset = 0; x_offset < BITS_IN_BYTE;
               x_offset++) {
            if (sprite[y_offset] & (MSB_SET >> x_offset)) {
                if (display[(y + y_offset) % DISPLAY_HEIGHT]
                    [(x + x_offset) % DISPLAY_WIDTH]) {
```

```
                    collision = true;
                }
                invert_pixel(x + x_offset, y + y_offset);
            }
        }
    }

    return collision;
}
```

Слика 5.8. *Имплементација методе Chip8CPU::draw_sprite()*

Инструкција *LD Vx, K* имплементирана је коришћењем методе `Chip8CPU::sdl_wait_for_key_press()` која је по имплементацији слична методи `Chip8CPU::sdl_poll_events`. Разлика је у томе што се уместо `SDL_PollEvent` користи `SDL_WaitEvent` функција. `SDL_WaitEvent` се користи да се безусловно чека на нови догађај, тако да долази до блокирања приликом позива ове функције, све док корисник не притисне неки тастер на тастатури.

```
int Chip8CPU::sdl_wait_for_key_press() {
    SDL_Event event;

    while (SDL_WaitEvent(&event)) {
        if (event.type == SDL_KEYDOWN) {
            auto key = get_keyboard_mapping_value(
                        event.key.keysym.sym);

            if (key != KEY_NOT_FOUND) {
                return key;
            }
            else if (event.key.keysym.scancode ==
                        SDL_SCANCODE_ESCAPE) {
                running = false;
                break;
            }
        }
        else if (event.type == SDL_QUIT) {
            running = false;
            break;
        }
    }

    return KEY_NOT_FOUND;
}
```

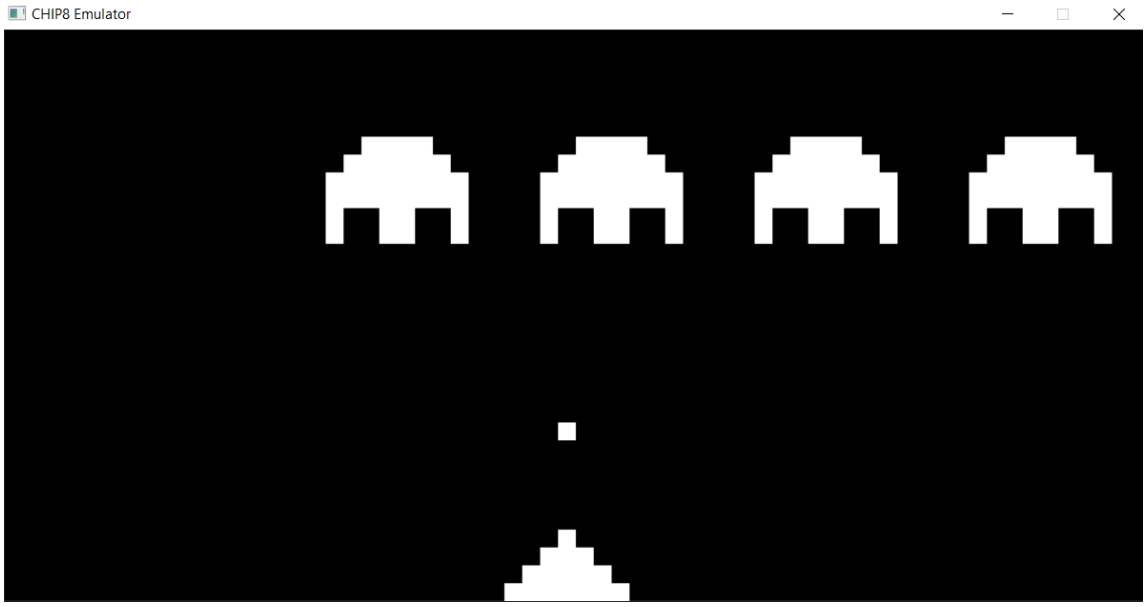
Слика 5.9. *Имплементација методе Chip8CPU::sdl_wait_for_key_press()*

Имплементација инструкције *LD B, Vx* приказана је на слици ***5.10.*** Приликом имплеметације, коришћена је модуларна аритметика приликом дељења броја на цифре. Идеја је да се цео број раздели на три цифре, од којих свака представља стотине, десетице и јединице, респективно. Након дељења броја на три цифре, одговарајуће цифре су уписане на три сукцесивне меморијске локације на које указује регистар ***I***.

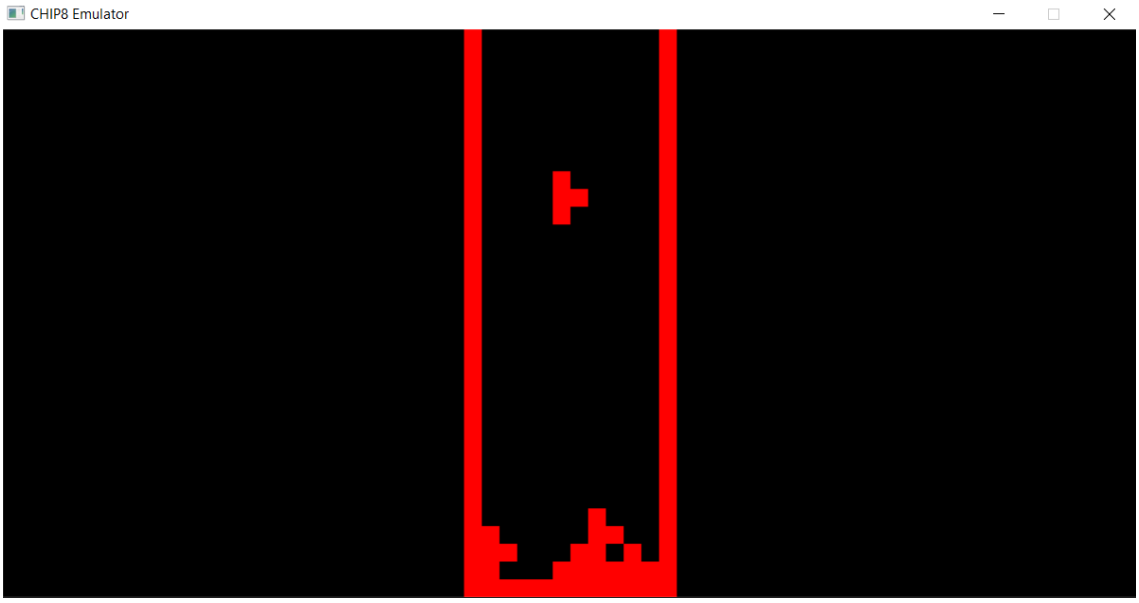
```
case 0x33: {
    auto x = get_third_nibble(opcode);

    write_mem(V[x] / HUNDREDS, I);
    write_mem((V[x] / TENS) % TENS, I + 1);
    write_mem(V[x] % TENS, I + 2);
}
    break;
```

Слика 5.10. *Имплементација инструкције LD B, Vx*

На сликама ***5.11***. и ***5.12.*** можемо видети емулатор у акцији како покреће две различите видео игре. Прва игра која се покреће је Chicken Invaders, а друга је *Tetris*. Будући да *CHIP-8* систем има монохроматски дисплеј, у овој имплеметацији је додата функционалност покретања са различитим бојама. Осим комбинације црне и беле, уз црну је могуће комбиновати још и црвену, плаву, зелену и жуту боју.



Слика 5.11. *Chicken Invaders игта на емулатору у црно-белој боји*



Слика 5.12. *Tetris игта на емулатору у црно-црвеној боји*

Што се тиче саме имплеметације класе `CHIP8Menu`, најбитнији делови имплементације представљени су на следећој слици. У оквиру ове класе бирамо једну од боја коју желимо да користимо у монохроматском дисплеју, као и један од фајлова који ћемо користити приликом покретања емулатора.

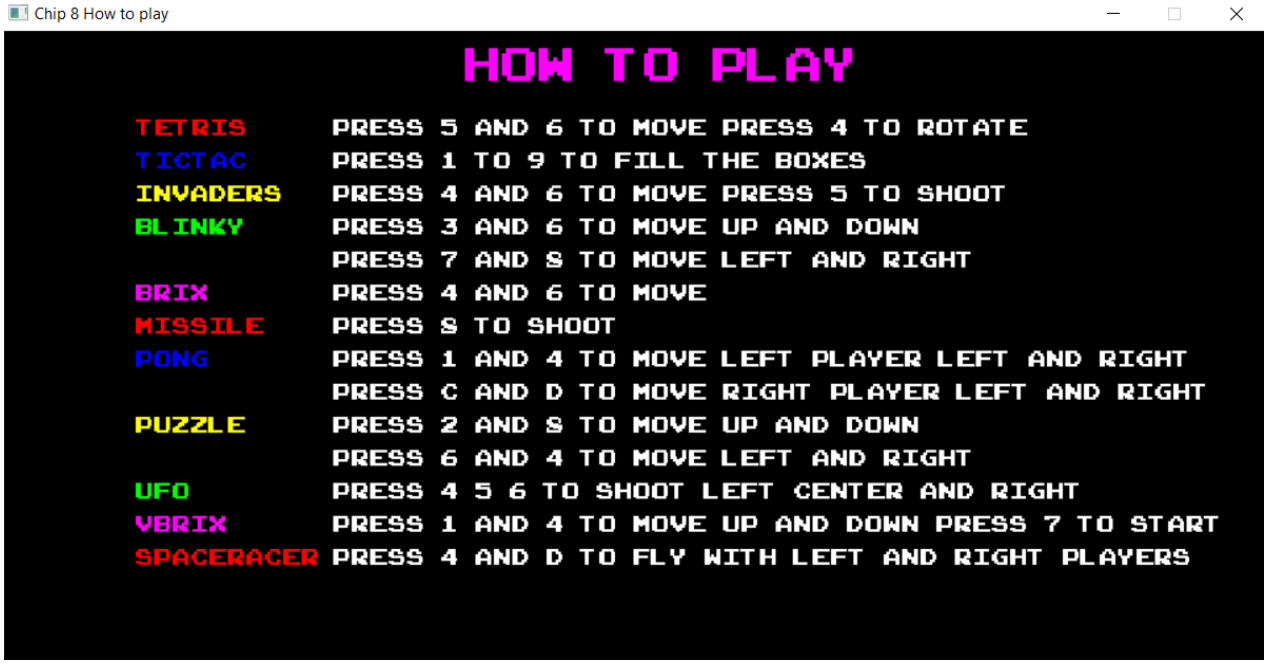
```
class CHIP8Menu {
public:
    void render_menu();
private:
    /* ROMS */
    static constexpr std::array<const char *, NUMBER_OF_ROMS> roms
            {
                    "TETRIS",
                    "TICTAC",
                    "INVADERS",
                    "BLINKY",
                    "BRIX",
                    "MISSILE",
                    "PONG",
                    "PUZZLE",
                    "UFO",
                    "VBRIX",
                    "SPACERACER"
            };
```

```
/* Colors */
static constexpr std::array<const char *, NUMBER_OF_COLORS> colors
            {
                    "WHITE",
                    "RED",
                    "GREEN",
                    "YELLOW",
                    "BLUE"
            };
};
```

Слика 5.13. *Скица костура класе Chip8Menu*

Метода `CHIP8Menu::render_menu()` служи за приказивање менија који је видљив приликом непосредног покретања емулатора. У оквиру почетног менија можемо изабрати боју коју желимо користити, игру коју желимо покренути, отворити нови прозор који ће приказати упутство за играње свих игара или упалити, односно угасити музику на притисак одговарајућег тастера. На сликама ***5.14.*** и ***5.15.*** можемо видети изглед главног менија, као и изглед помоћне странице са инструкцијама за различите игре.

Слика 5.14. *Приказ почетне странице менија емулатора*



Слика 5.15. *Приказ помоћне странице са инструкцијама*

Имплементација саме методе веома подсећа на имплементације SDL метода које смо описали раније. Цело извршавање се дешава у петљи која проверава да ли је дошло до неког догађаја, и у случају да јесте, тај догађај се обрађује на одговарајући начин. На следећој слици видећемо имплементацију мењања боја и одабира видео игре која се покреће.

```
case SDL_KEYDOWN: {
    auto& scancode = event.key.keysym.scancode;

    if (scancode == SDL_SCANCODE_RIGHT) {
        SDL_SetRenderDrawColor(renderer, 0, 0, 0, 0);
        SDL_RenderFillRect(renderer, &r_13);
        SDL_RenderPresent(renderer);

        rom_index = (rom_index + 1) % NUMBER_OF_ROMS;

        surface_13 = TTF_RenderText_Solid(font_type_medium,
                          roms[rom_index], green_color);
        texture_13 = SDL_CreateTextureFromSurface(renderer,
                          surface_13);
    }
    else if (scancode == SDL_SCANCODE_LEFT) {
        SDL_SetRenderDrawColor(renderer, 0, 0, 0, 0);
        SDL_RenderFillRect(renderer, &r_13);
```

```
        SDL_RenderPresent(renderer);

        if (--rom_index < 0) {
            rom_index += NUMBER_OF_ROMS;
        }

        surface_13 = TTF_RenderText_Solid(font_type_medium,
                         roms[rom_index], green_color);
        texture_13 = SDL_CreateTextureFromSurface(renderer,
                         surface_13);
    }
     ...
```

Слика 5.16. *Имплементација промене боје и видео игре*

Треба приметити да се исти поступак коришћења *SDL* компоненти користи и овде. Правоугаоници са одређеним координатама се попуњавају одређеном бојом и потом се приказују на екрану. Као нова компоненте се уводе *SDL_Surface* и *SDL_Texture*. Обе од ових компоненти се користе како би их компонента *SDL_Renderer* успешно приказала на екрану. *SDL_Surface* и *SDL_Texture* садрже информације о пикселима које треба приказати на екрану и оне се користе овде јер је потребно на екрану приказати релативно комплексније објекте од једног пиксела, као што су фонтови. Разлика између *SDL_Surface* и *SDL_Texture* је та што први користи софтверско рендеровање користећи *RAM* меморију, док други користи хардверско рендеровање користећи графичку карту.

Притиском тастера ***S*** на тастатури у оквиру главног менија, музика се може зауставити или поново покренути. Имплементацију тога можемо видети на слици ***5.17.***

```
...
else if (scancode == SDL_SCANCODE_S) {

    if (Mix_PausedMusic() == 1) {
        Mix_ResumeMusic();
    }
    else {
        Mix_PauseMusic();
    }
}
...
```

Слика 5.17. *Приказ заустављања и покретања музике*

Коначно, притиском тастера ***Enter*** започињемо сесију играња и покрећемо емулатор. Приликом притиска овог тастера, све приказане структуре које су у том тренутуку биле на екрану се уништавају, као и прозор менија који је био активиран. Ствара се нова инстанца емулатора, која приликом своје креације креира и нови прозор који ће бити приказан на екрану. Нови прозор представља прозор који смо могли да видимо на сликама ***5.14.*** и ***5.15.***, само уз различиту конфигурацију боје и игара који ће бити покренути.

```
else if (scancode == SDL_SCANCODE_RETURN) {
    SDL_DestroyTexture(texture_1);

    ...

    Mix_Quit();
    SDL_Quit();

    Chip8CPU emulator;
    std::string path =
          std::string{"../roms/"} + roms[rom_index];

    emulator.emulate(path, color);

    running = false;
}
```

Слика 5.18. *Приказ креирања нове инстанце емулатора из менија*

6 Корисничко упутство

Као што је већ претходно поменуто, као улаз емулатора се користе бинаризоване датотеке које је могуће преузети са интернета. Те датотеке се састоје од низа нула и јединица које представљају операционе кодове инструкција које емулатор треба да испарсира и изврши. Због тога, уз аквизицију нових *ROM* фајлова, код би се могао модификовати тако да подржи и додатне игре које је могуће довести на улаз емулатора.

Детаљне инструкције о самим играма дате су на помоћној страници на коју је из главног менија могуће отићи притиском тастера на тастатури. За сваку од игара које текућа имплементација емулатора подржава, на тој страници је могуће пронаћи команде које корисник треба да зна да би умео да игра одговарајућу игру. Треба напоменути и да постоје одређене игре које је могуће играти у двоје људи, што је такође назначено на помоћној страници. Помоћну страницу можемо видети на слици ***5.15.***

Сценарио употребе емулатора је следећи: приликом покретања апликације корисник је презентован главним менијем. У оквиру главног менија, корисник може да специфицира различите ствари у зависности од тога шта жели. Корисник може да промени боју у којој ће емулатор радити, као и да изабере игру коју жели да покрене. Такође, корисник је у могућности да угаси музику која се пушта приликом покретања апликације притиском на одређени тастер, као и да притиском тастера отвори нови прозор који ће му показати све команде за сваку тренутно подржану игру. Након што је корисник завршио са намештањем боје и одабиром игре коју жели да игра, притиском тастера на тастатури може започети нову сесију која заправо покреће емулатор и отвара нови графички приказ на екрану. Корисник у том тренутку, пратећи инструкције за играње које је могао видети на помоћној страници, преузима контролу и у могућности је да игра одговарајућу видео игру коју је претходно одабрао. После завршене игре, корисник може изаћи из апликације притиском тастера на тастатури или кликом на ***X*** дугме својим мишем.

7 Закључак

У овом раду представљена је једна имплементација интерпретативног емулатора за систем са *CHIP-8* архитектуром. У раду је, осим емулатора, имплементиран и интерактивни, графички мени, који заједно са емулатором као целину кориснику пружа целокупно искуство играња ретро видео игара. Корисник може, у зависности од тога коју игру жели да покрене, као и од тога коју боју жели да искористи да прикаже на монохроматском дисплеју, по свом нахођењу покрене систем са јединственим параметрима, баш онако како је он замислио.

Ова имплементација би се потенцијално могла унапредити тако што би се интерпретативни емулатор заменио емулатором који ради на бази *динамичке рекомпилације*, у том случају би било могуће постићи већу брзину извршавања инструкција. Будући да су сви *ROM* фајлови који се користе за покретање емулатора већ унапред написани и бинаризовани, још једно унапређење би било написати *асемблер* за овај језик, који би служио да се програм написан на *CHIP-8* језику после асемблирања преведе на низ нула и јединица који у ствари представљају операционе кодове инструкција које треба извршити.

Пошто званична граматика *CHIP-8* језика није у потпуности дефинисана, зарад ове имплементације било би потребно прво осмислити тачну граматику и синтаксу језика, а тек онда написати одговарајући *асемблер* који би био у стању ову граматику да парсира и обради. Онда би корисник био у стању да сам напише програм који жели на замишљеном језику, а онда би такав програм после асемблирања могао да учита у емулатор и да га изврши. У случају самосталног писања програма од стране корисника, било би пожељно имати и алат који би проверио да ли је корисник добро превео све инструкције у операционе кодове. Тада би *дисасемблер* могао помоћи ономе који пише програм, како би писац могао да потврди да ли је оно што је написао заиста очекиваног формата.

Постоје новије верзије *CHIP-8* архитектура које су унапређивале инструкцијски сет како су године пролазиле, тако да би ова имплеметација такође могла бити унапређена тако што би се проширио њен инструкцијски сет, а самим тим би се и отворила могућност за учитавањем нових игара које имају другачије функционалности, а нису компатибилне са тренутном архитектуром [12].

8 Литература

[1] *CHIP-8* Википедија страница: https://en.wikipedia.org/wiki/CHIP-8 *приступано јун 2021.*

[2] Фон Нојманова архитектура: https://en.wikipedia.org/wiki/Von_Neumann_architecture *приступано јун 2021.*

[3] Michael Steil, Georg Acher, Dynamic Re-compilation of Binary RISC Code for CISC Architectures, Munchen: Technische Universitat Munchen Institut fur Informatik

[4] Тјурингова машина: https://en.wikipedia.org/wiki/Turing_machine *приступано јун 2021*.

[5] Victor Moya del Barrio, Agustin Fernandez, Study of the techniques for emulation programming, Barcelona: FIB UPC, 2001

[6] С.Стојановић, Д.Бојић, „Материјали са предавања и вежби курса Системски софтвер“, Електротехнички факултет, Универзитет у Београду, *приступано јун 2021.*

[7] Memory Emubook: http://emubook.emulation64.com/memory.htm *приступано јун 2021.*

[8] Поређење перформанси емулације: https://alexaltea.github.io/blog/posts/2018-04-18-lle-vs-hle/ *приступано јун 2021.*

[9] Предности емулације на високом у поређењу са емулацијом на ниском нивоу: https://emulation.gametechwiki.com/index.php/High/Low_level_emulation *приступано јун 2021.*

[10] Архитектура *CHIP-8* емулатора: https://austinmorlan.com/posts/chip8_emulator/#how-does-a-cpu-work *приступано јул 2021.*

[11] *CHIP-8* techical reference *v1.0*. http://devernay.free.fr/hacks/chip8/C8TECH10.HTM *приступано јул 2021.*

[12] Унапређења CHIP-8 архитектуре: https://chip-8.github.io/extensions/ *приступано август 2021.*

9 Списак коришћених слика

Слика 2.1.	*https://www.researchgate.net/figure/An-interactive-Turing-machine-with-advice_fig2_272684665*
Слика 2.1.1.	*http://net-informations.com/java/intro/jvm.htm*
Слика 2.5.1.	*https://cstaleem.com/memory-management-unit-in-os*
Слика 3.1.	*https://www.computerscience.gcse.guru/theory/von-neumann-architecture*
Слика 3.2.	*http://emubook.emulation64.com/memory.htm*
Слика 3.1.2.	*С.Стојановић, Д.Бојић, „Материјали са предавања и вежби курса Системски софтвер“, Електротехнички факултет, Универзитет у Београду*
Слика 4.1.2.	*http://devernay.free.fr/hacks/chip8/C8TECH10.HTM*

Све остале слике коришћене у овом раду, које се не налазе у табели горе је аутор овог рада сам обезбедио.