Универзитет у Београду

Електротехнички факултет



Конфигурабилни преводилац и симулатор архитектура рачунара

Мастер рад

|  |  |
| --- | --- |
| Ментор: | Кандидат: |
| доц. др Милош Цветановић | Бојан Јелача 3106/2014 |

Београд, Август 2016.

Садржај

[1. Увод 1](#_Toc460377187)

[2. Преглед постојећих решења 3](#_Toc460377188)

[2.1. Proteus Design Suite 3](#_Toc460377190)

[2.2. *VisualSim Architect* 4](#_Toc460377191)

[2.3. *Computer OS Simulator* 5](#_Toc460377192)

[2.4. Пресек анализираних решења 6](#_Toc460377193)

[3. Преглед функционалности 8](#_Toc460377194)

[3.1. Дефинисање архитектуре рачунарских компоненти 8](#_Toc460377195)

[3.1.1. Дефинисање архитектуре меморије 8](#_Toc460377196)

[3.1.2. Дефинисање архитектуре компоненти општег типа 10](#_Toc460377197)

[3.2. Дефинисање рада рачунарских компоненти 11](#_Toc460377198)

[3.3. Дефинисање изгледа рачунарских компоненти 12](#_Toc460377199)

[3.3.1. Подразумевани начин дефинисања изгледа рачунарске компоненте 13](#_Toc460377200)

[3.3.2. Дефинисање изгледа рачунарске компоненте кроз графички интерфејс 13](#_Toc460377201)

[3.3.3. Дефинисање изгледа рачунарских компоненти писањем наредби у коду 14](#_Toc460377202)

[3.4. Пројектовање рачунарског система 15](#_Toc460377203)

[3.4.1. Додавање компоненти 17](#_Toc460377204)

[3.4.2. Дефинисање веза између компоненти 17](#_Toc460377205)

[3.4.3. Додавање магистрале 18](#_Toc460377206)

[3.4.4. Meњање фреквенције рада система 18](#_Toc460377207)

[3.4.5. Чување пројектованог система 18](#_Toc460377208)

[3.4.6. Учитавање претходно сачуваног система 18](#_Toc460377209)

[3.5. Симулирање рада рачунарског система 18](#_Toc460377210)

[4. Реализација 20](#_Toc460377211)

[4.1. Реализација графичког интерфејса 20](#_Toc460377212)

[4.1.1. Реализација прозора за дефинисање изгледа компоненти 20](#_Toc460377213)

[4.1.2. Реализација прозора за пројектовање система и симулирање рада система 25](#_Toc460377214)

[4.2. Реализација логичких функционалности 29](#_Toc460377215)

[4.2.1. Дефинисање архитектуре и рада рачунарских компоненти 29](#_Toc460377216)

[4.2.2. Дефинисање изгледа рачунарских компоненти 32](#_Toc460377217)

[4.2.3. Пројектовање рачунарског система 35](#_Toc460377218)

[4.2.4. Симулирање рада рачунарског система 38](#_Toc460377219)

[5. Тестирање 42](#_Toc460377220)

[6. Закључак 49](#_Toc460377221)

[7. Литература 51](#_Toc460377222)

[8. Списак слика 52](#_Toc460377223)

[9. Списак табела 53](#_Toc460377224)

1. Увод

У последњих 50 година, рачунарска техника је наука која се најбрже и највише развијала од свих осталих наука. Стално се смишљају нова решења како да се постигне што боље пројектован рачунарски систем који ће показати што боље перформансе и прецизност у раду, а опет трошити што мање енергије и користити што јефтиније компоненте, како би цена била што прихватљивија.

Заједно са порастом потребе за што бољим процесорима, јавља се потреба да се пројектовање процесорског система обавља што брже, због брзих промена на тржишту. У ту сврху, потребно је дефинисање система учинити што једноставнијим, а да се опет при томе што мање изгуби на квалитету пројектованих решења. С том сврхом, јавља се велики број симулатора рачунарских система, који се користе да се, пре него што се пројектовано решење физички реализује, истестира у погледу перформанси, потрошње енергије, прецизности рада и других параметера. Такође, како истовремено расте и број корисника, потребно је њихово што лакше упознавање са неким системом, како би се што пре адаптирали на рад са истим. И у овом случају, симулатори играју значајну улогу, пошто се могу користити за лакше увођење нових корисника у рад или како би корисници што боље схватили архитектуру система.

Како развијање симулатора није мали посао, јавља се потреба за што конфигурабилнијим симулаторима, односно за симулаторима који нису прављени за једну специфичну архитектуру система, већ покривају више различитих или дозвољавају својим корисницима да сами дефинишу систем. Због тога се, паралелно са развојем рачунарских система, развијају и симулатори рачунарских система. Постојали велики број решења, од који су нека споменута у овом раду, али ниједно од ових решења није потпуно конфигурабилно, што ограничава могућности корисника и врло често намеће потребу за развојем специфичног симулатора. Због тога се јавила идеја за развојем потпуно конфигурабилног симулатора, за који ће моћи да се дефинише све, почевши од машинског језика, до архитектуре целог система.

Тема овог рада је израда софтверске апликације чија је сврха да кориснику омогући дефинисање произвољног рачунарског система за који онда добија преводилац и симулатор. Тако имплементирано окружење би онда могло да се користи за тестирање нових идеја у области архитектуре рачунара, као и за едукацију и обучавање нових корисника.

Сам рад представља надоградњу дипломског рада , који је кориснику омогућавао да дефинише само архитектуру рачунара да добије окружење које преводи и извршава програме за дефинисану архитектуру. Апликација која је предмет и циљ овог рада омогућава кориснику да, поред архитектуре рачунара, дефинише и архитектуре осталих рачунарских компоненти, начине на који компоненте међусобно интерагују, као и изглед сваке од компоненти. Након што то уради, корисник добија преводилац за језик дефинисане архитектуре рачунара и симулатор на коме може да покреће програмски код. Приликом покретања кода, корисник може да прати стања свих компоненти система, да прати вредности сигнала и магистрала које повезују компоненте, да дефинише фреквенцију рада система итд.

Основна намена апликације је едукативна и експериментална. У едукативном смислу, апликација је намењена наставницима како би лакше пренели знање студентима везано за неку конкретну архитектуру рачунарског система, као и како би омогућили студентима лакше разумевање теоријских основа рачунарских система. Такође, едукативна сврха апликације је и обучавање студената за пројектовање рачунарских система. У експерименталном смислу, апликација се може примењивати за тестирање нових идеја у пројектовању рачунарских система, пре него што се она хардверски реализују, како би се одмах виделе основне предности и мане идеја.

У поглављу 2 направљен је кратак преглед постојећих симулатора архитектура рачунара, истакнуте врлине и мане сваког од њих и направљен пресек постојећих решења. Поглавље 3 детаљно описује све функционалности које су реализоване у оквиру овог рада и како их постићи. Ово поглавље је практично корисничко упутство за рад са апликацијом. У поглављу 4 приказан је начин реализације функционалности приказан у поглављу 3. У овом поглављу могу се наћи детаљна објашњења најважнијих делова имплементације апликације, праћени примерима програмског кода и дијаграмима класа. Такође, речено је и на које препреке се наишло током развоја апликације и како су оне превазиђене и објашњене су неке одлуке донешене приликом пројектовања архитектуре ове апликације. Реализоване функционалности су тестиране помоћу система посебно дефинисаних за тестирање. Како је апликација тестирана и какви су резултати постигнути приказано је у поглављу 5.

У последњем поглављу, изведен је закључен и дате смернице како се реализована апликација може проширивати и надограђивати.

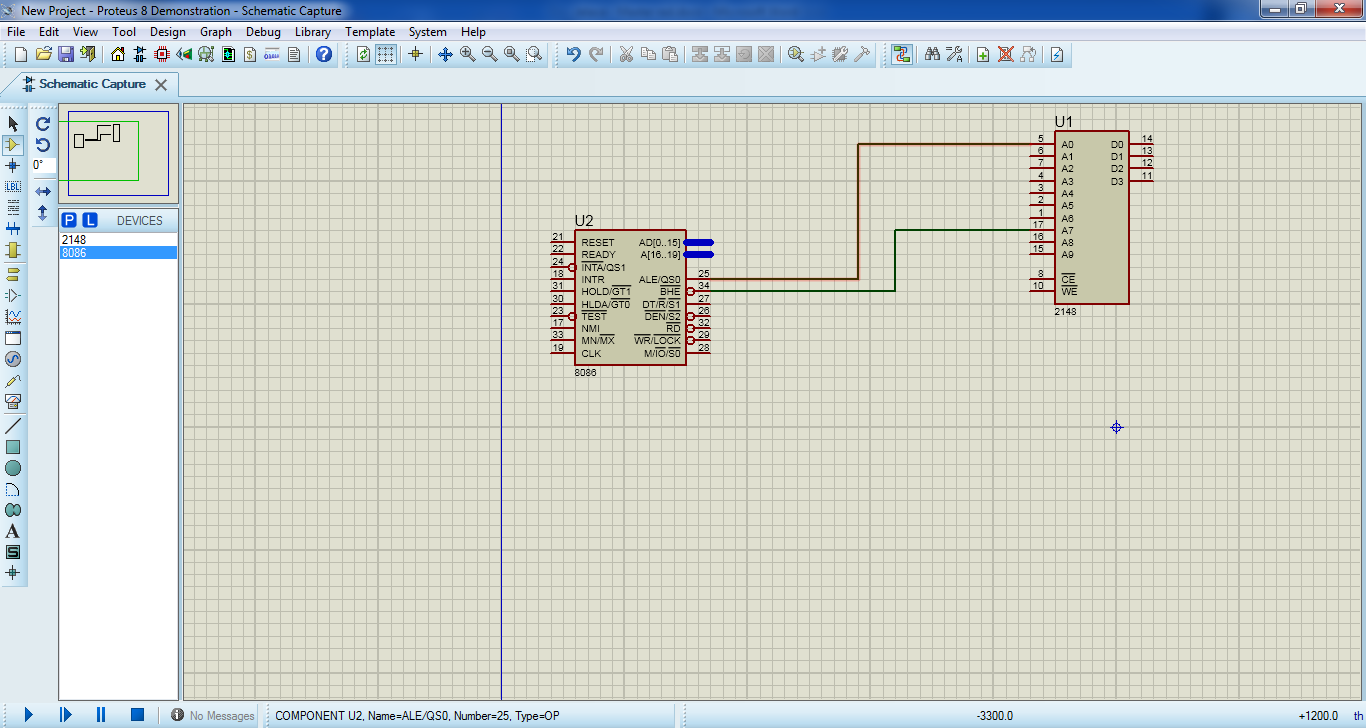
1. Преглед постојећих решења

Као што је наведено у претходном поглављу, постоји потреба за симулатором који би могао да буде потпуно конфигурабилан, односно да корисник може да дефинише све делове система, од програмског језика који се користи за писање програма, до симулирања рада тих програма, уз праћење стања целог система. У овом поглављу, биће дат преглед постојећих симулатора архитектура рачунара. Посебан акценат биће стављен на својства која се тичу конфигурабилности, пријатности коришћења и могућности.

* 2. Proteus Design Suite

*Proteus Design Suite* је алат који је развила британска компанија *Labcenter Electronics Ltd.* 1988. године. Алат се покреће на оперативном систему *Microsoft Windows.* Више о овом алату може се наћи на [4].

Ово је алат за аутоматизацију електронског дизајна који укључује дефинисање шеме система, симулацију рада микроконтролера и бројне модуле штампаних плоча. На слици 2.1.1. приказан је део алата *Proteus Design Suite* за дефинисање шеме система.



Слика 2.1.1. Дефинисање шеме система у алату *Proteus Design Suite*

Пошто не захтева постојање никаквог хардвера, има веома широку примену као алат за учење и вежбање, поготово што покрива широк спектар технологија.

Овај алат подржава широку палету компоненти које се могу користити, које могу бити најразличитијих намена и типова. Алат *Proteus Design Suite* подржава конфигурисање компоненти и то је могуће обавити кроз дизајн штампаних плоча. Ово се може урадити тако што се прво дефинише унутрашња шема штампане плоче, а затим и спољашњи изглед и пинови. Овакав начин дефинисања је прилично реалан и подсећа на пројектовање штампаних плоча у стварном животу, али може бити компликован и понекад одузети доста времена.

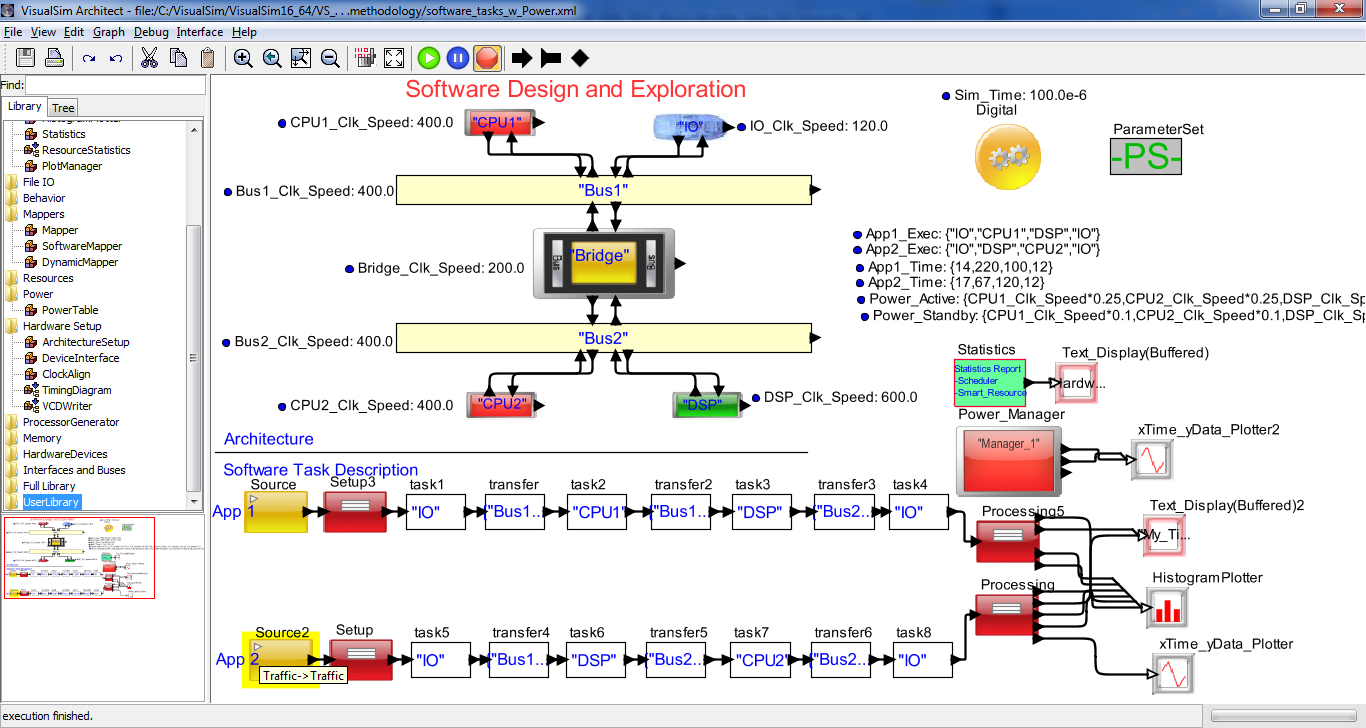
Алат *Proteus Design Suite* нема у себи уграђен преводилац, па се за писање и извршавање програма морају користити екстерни алати, што поједностављује коришћење алата, али смањује конфигурабилност.

Симулација рада микроконтролера се реализује тако што као улаз добије датотеку са машинским кодом. Након тога, симулација се обавља паралелно са било којом електронском компонентом повезаном било аналогним, било дигиталним везама. Симулација се увек одвија у потпуности, односно није могуће пратити извршавање у режиму такт по так.

* 1. *VisualSim Architect*

Окружење *VisualSim Architect* је окружење за моделовање и симулирање рада комплексних система. Ови системи не морају бити нужно рачунарски, већ је могуће моделовати и роботске системе, електронске системе и многе друге. Настао је 2003. године као комерцијална верзија истраживачког рада групе стручњака са Универзитета Беркли. Више о овом окружењу може се наћи на [5].

На слици 2.2.1. приказан је основни екран овог окружења.



Слика 2.2.1. Основни екран окружења *VisualSim Architect*

Програм који се извршава у овом окружењу се може дефинисати као низ софтверских задатака који се извршавају један за другим. Овакво решење је добро, јер је интуитивно и корисницима који нису програмери, али има недостатак који се огледа у смањеном броју могућности које се пружају при оваквом дефинисању програма.

Алат *VisualSim Architect* пружа велику библиотеку компоненти које се могу користити за моделовање система. Ове компоненте се могу додавати у систем и повезивати. Такође, свака компонента је конфигурабилна и могу се дефинисати њено понашање и изглед. Понашање се може дефинисати кроз скрипте написане у језику посебно осмишљеном за ову сврху, који највише подсећа на језик једноадресне архитектуре. Други начин за дефинисање понашања компоненте је кроз дефиницију компоненте као подсистема, односно, свака компонента се може направити као подсистем већ постојећих компоненти.

Извршавање програма се може покренути кликом на дугме Go, након што је систем дефинисан. Овакво извршавање обавља проточну обраду свих дефинисаних задатака. На крају извршавања кориснику се приказује низ графика корисник за статистичку обраду дефинисаног система, као што су:

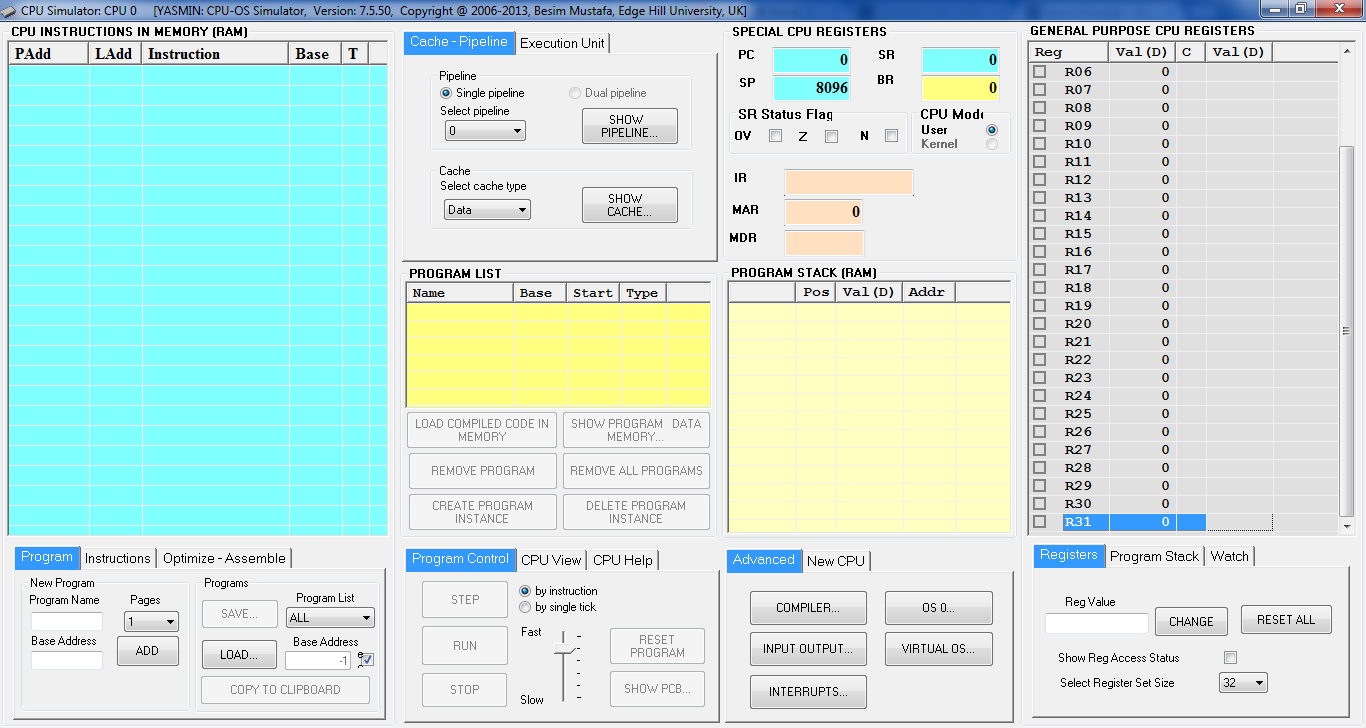
* списак направљених корака, заједно са временима
* хистограм кашњења софтверских задатака
* график кашњења софтверских задатака
* график потрошње енергије

Иако показује велики број статистички битних података, овакав начин извршавања има једну ману. Програми се увек извршавају од почетка до краја. Могуће је паузирати њихово извршавање, али је за то потребно притиснути дугме у правом тренутку, да би се проверило стање у одређеном тренутку извршавања.

* 1. *Computer OS Simulator*

*Computer OS Simulator* је окружење за симулирање рада *RISC* процесора, заједно са радом оперативног система који обавља више послова истовремено. Развио га је 2006. године Бесим Мустафа са Универзитета Еџ Хил у Великој Британији. Више о овом окружењу може се наћи на [6].

На слици 2.3.1. приказан је изглед екрана овог окружења.



Слика 2.3.1. Изглед екрана окружења *Computer OS Simulator*

Ово окружење је дизајнирано за писање и симулирање рада програма написаних за процесоре *RISC* архитектуре. Није могуће дефинисати систем, већ се за то увек користи стандардан *RISC* процесор. Из тог разлога, не постоји начин дефинисања ни компоненти система.

Код који се пише се пише у асемблерском језику. Постоји значајан број инструкција и начина адресирања који су предефинисани, али не постоји начин да се овај код конфигурише.

Извршавање програма се може обављати такт по такт или инструкцију по инструкцију. У сваком тренутку, могу се пратити вредности регистара, меморијских локација или дефинисати променљиве чија је вредност од интереса.

* 1. Пресек анализираних решења

Из до сада наведених решења, могло се видети да постојећа решења пружају широк спектар могућности за дефинисање разних система и симулирања њиховог рада. У табели 2.4.1. дат је пресек анализираних решења.

Табела 2.4.1. Пресек постојећих решења

|  |  |  |  |
| --- | --- | --- | --- |
| **Назив окружења** | ***Proteus Design Suite*** | ***VisualSim Architect*** | ***Computer OS Simulator*** |
| **Конфигурисање преводиоца** | - | - | - |
| **Конфигурисање компоненти** | + | + | - |
| **Конфигурисање система** | + | + | - |
| **Дефинисање програма** | +/- | +/- | + |
| **Симулирање целог програма** | + | + | + |
| **Симулирање такт по такт** | - | - | + |
| **Лакоћа коришћења апликације** | + | - | + |

Као што се може видети из табеле изнад, не постоји окружење које кориснику омогућило да сам дефинише језик у ком ће писати програме и преводилац у коме ће их преводити у машински код, да конфигурише компоненте и цео систем и да симулира рад тако дефинисаних система. Најближи овоме је алат *Proteus Design Suite*. Иако за овај алат не постоји начин да се дефинише преводилац, могуће је искористити екстерне алате за генерисање машинског кода, који се након тога може извршавати, што може задовољити потребе. Ипак, ово ограничава корисника да дефинише свој језик у коме ће писати програме, већ мора користити неки од постојећих језика. Такође, не постоји извршавање такт по такт, што додатно отежава праћења стања система.

Због свега наведеног, јавља се потреба за постојањем апликације која би ујединила све функционалности наведене у табели 2.4.1, а притом била и пријатна за коришћење. Управо таква апликација је тема овог рада. Оваква апликација би требало да кориснику омогући да дефинише програмски и машински језик који се користи за писање програма, дефинише компоненте које се могу користити, дефинише систем чији рад се симулира, да пише и преводи програме и симулира рад система, односно извршава написане програме. Управо таква апликација је тема овог рада.

1. Преглед функционалности

У овом поглављу дат је детаљан преглед свих функционалности које апликација која је предмет овог рада укључује, заједно са упутством за реализовање ових функционалности. Пошто је део функционалности развијен као тема дипломског рада [1] у овом поглављу пажња се посвећује искључиво функционалностима које нису биле део дипломског рада.

Функционалности које ова апликација додаје на функционалности апликације која је развијена као тема дипломског рада могу се поделити на:

* дефинисање архитектуре рачунарских компоненти
* дефинисање рада рачунарских компоненти
* дефинисање изгледа рачунарских компоненти
* пројектовање рачунарског система
* симулирање рада рачунарског система
  1. Дефинисање архитектуре рачунарских компоненти

Под термином *архитектура рачунарске компоненте* подразумевају се својства која описују појединачну компоненту, која заједно са другим компонентама, гради рачунарски систем. Примери оваквих компоненти су процесор, меморија, контролер прекида итд. Овај појам не треба мешати са појмом *архитектура рачунара* из наслова рада, јер се он односи на архитектуру целог рачунарског система, односно на архитектуру сваке појединачне компоненте и везе између њих.

Апликација разликује три типа рачунарских компоненти:

* процесор
* меморија
* компоненте општег типа

Процесор је централна компонента рачунарског система која може покретати машински код добијен превођењем програмског кода. Овом типу компоненте се неће посвећивати већа пажња, пошто је детаљно описан у дипломском раду [1].

У наредним одељцима, пажња ће бити посвећена меморији и компонентама општег типа.

* + 1. Дефинисање архитектуре меморије

Меморија је компонента чија је намена складиштење веће количине података. Она не може извршавати програмски код и нема регистарски фајл, већ само меморијске локације. Корисник може дефинисати протокол по коме се подаци дохватају из меморије и протокол по коме се подаци уписују у меморију. Детаљан опис дефинисања овог протокола дат је у потпоглављу 3.2.

Услов за реализацију функционалности дефинисања архитектуре меморије је претходно отворен пројекат.

Да би дефинисао архитектуру меморије, корисник треба из главног менија апликације да изабере опцију *Architecture*, из менија који се отвори опцију *Load* и затим опцију *Memory*. Све ово се може остварити и истовременим притиском тастера *CTRL* и *М* на тастатури. Након тога, отвара се дијалог за избор датотеке у којој се налази опис архитектуре меморије.

Уколико већ није, корисник може ван апликације да дефинише датотеку која описује архитектуру меморије. Ради се о датотеци написаној у *XML* језику чији је детаљан опис дат у наставку текста.

Цела датотека треба да садржи један главни *XML* чвор *memory.*Унутар овог чвора налазе се чворови који дефинишу својства меморије:

* **Чвор *name* -** Вредност овог чвора дефинише име меморијске компоненте
* **Чвор *filename* -** Вредност овог чвора дефинише локацију и име датотеке у којој се налази код који дефинише рад меморије, о чему ће више речи бити у потпоглављу 3.2. Уколико се овај чвор не наведе, узима се датотека са именом истим као име меморијске компоненте, са екстензијом *.cs* и смешта у директоријум */Data/Memories* унутар пројектног директоријума.
* **Чвор *size* -** Вредност овог чвора дефинише укупну величину меморије у бајтовима
* **Чвор *au*** *-* Вредност овог чвора дефинише величину адресне јединице
* **Чвор *rom\_start* -** Вредност овог чвора дефинише почетну адресу РОМ меморије
* **Чвор *rom\_end* -** Вредност овог чвора дефинише крајњу адресу РОМ меморије
* **Чвор *ram\_start*** *-* Вредност овог чвора дефинише почетну адресу РАМ меморије
* **Чвор *ram\_end* -** Вредност овог чвора дефинише крајњу адресу РАМ меморије
* **Чвор *init\_file*-** Вредност овог чвора дефинише локацију датотеке у којој се налази иницијални садржај меморије који корисник може да дефинише
* **Чвор *storage\_file*-** Вредност овог чвора дефинише локацију датотеке у којој се чува садржај меморије током извршавања програма
* **Чвор *dimensions* -** Вредност овог чвора дефинише величину визуелне репрезентације меморије. Више о овоме биће у потпоглављу3.3.
* **Чвор *ports* -** Вредност овог чвора дефинише портове које меморија садржи. Овај чвор има своје подчворове од којих сваки представља назив порта. Подчворови сваког овог чвора дефинишу његова својства:
  + **Чвор *name* -** Вредност овог подчвора дефинише име порта
  + **Чвор *side* -** Вредност овог подчвора дефинише страну на којој се налази порт. Може имати вредности *left, rigth, up* и *down.*
  + **Чвор *type* -** Вредност овог подчвора дефинише тип порта. Може имати вредности *in, out* и *inout.*
  + **Чвор *number* -** Вредност овог подчвора дефинише величину порта у битовима, тј. број пинова порта.
* **Чвор *design* -** Овај чвор се користи при дефинисању изгледа меморије. О овоме ће више речи бити у потпоглављу3.3.

Након што је учитао датотеку која описује архитектуру меморије, апликација сама генерише датотеку специфицирану чвором *filename*, уколико она већ није генерисана. Ова датотека дефинише код који описује рад меморије. Више речи о томе биће у потпоглављу 3.2.

* + 1. Дефинисање архитектуре компоненти општег типа

Компоненте општег типа су компоненте чија је намена обављање посла који корисник дефинише. Оне не могу извршавати програмски код и немајумеморијске локације, већ само регистарски фајл. Детаљнији опис дефинисања посла који компонента обавља дат је у потпоглављу 3.2.

Услов за реализацију функционалности дефинисања архитектуре компоненти општег типа је претходно отворен пројекат.

Да би дефинисао архитектуру компоненте општег типа, корисник треба из главног менија апликације да изабере опцију *Architecture*, из менија који се отвори опцију *Load* и затим опцију *Other Component*. Све ово се може остварити и истовременим притиском тастера *CTRL* и *T* на тастатури. Након тога, отвара се дијалог за избор датотеке у којој се налази опис архитектуре компоненте општег типа.

Уколико већ није, корисник може ван апликације да дефинише датотеку која описује архитектуру компоненте општег типа. Ради се о датотеци написаној у *XML* језику чији је детаљан опис дат у наставку текста.

Цела датотека треба да садржи један главни *XML* чвор *othercomponent.*Унутар овог чвора налазе се чворови који дефинишу својства компоненте општег типа:

* **Чвор *name* -** Вредност овог чвора дефинише име компоненте
* **Чвор *filename* -** Вредност овог чвора дефинише локацију и име датотеке у којој се налази код који дефинише рад компоненте општег типа, о чему ће више речи бити у потпоглављу 3.2.Уколико се овај чвор не наведе, узима се датотека са именом истим као име компоненте, са екстензијом *.cs* и смешта у директоријум */Data/Other* унутар пројектног директоријума.
* **Чвор *dimensions* -** Вредност овог чвора дефинише величину визуелне репрезентације компоненте. Више детаља о овоме биће изнето у потпоглављу 3.3.
* **Чвор *registers* -** Вредност овог чвора дефинише регистре које компонента садржи. Сваки подчвор представља дефиницију једног регистра. Његови подчворови дефиншу својства регистра:
  + **Чвор *size* -** Вредност овог подчвора дефинише величину регистра у битовима.
  + **Чвор *name* -** Вредност овог подчвора дефинише име регистра
  + **Чвор *value* -** Вредност овог подчвора дефинише почетну вредност регистра
  + **Чвор *part* -** Ово је посебан тип подчвора који служи за дефинисање делова регистра, односто подрегистара. Сваки подчвор овог типа дефинише један део регистра који се може користити у програмском коду.Дефинисање дела неког регистра се обавља кроз подчворове подчвора *part*:
    - **Чвор *start* -** Овај подчвор дефинише бит од кога део регистра почиње
    - **Чвор *end* -** Овај подчвор дефинише бит на којем се завршава део регистра
    - **Чвор *name* -** Овај подчвор дефинише име, односно псеудоним дела регистра који се може користити у програмском коду
* **Чвор *ports* -** Вредност овог чвора дефинише портове које компонента садржи. Овај чвор има своје подчворове од којих сваки представља назив порта. Подчворови сваког овог чвора дефинишу његова својства:
  + **Чвор *name* -** Вредност овог подчвора дефинише име порта
  + **Чвор *side* -** Вредност овог подчвора дефинише страну на којој се налази порт. Може имати вредности *left, right, up* и *down.*
  + **Чвор *type* -** Вредност овог подчвора дефинише тип порта. Може имати вредности *in, out* и *inout.*
  + **Чвор *number* -** Вредност овог подчвора дефинише величину порта у битовима, тј. број пинова порта.
* **Чвор *design* -** Овај чвор се користи при дефинисању изгледа компоненте. О овоме ће више речи бити у потпоглављу 3.3.

Након што је учитао датотеку која описује архитектуру меморије, апликација сама генерише датотеку специфицирану чвором *filename*, уколико она већ није генерисана. Ова датотека дефинише код који описује рад меморије. Више речи о томе биће у потпоглављу 3.2.

* 1. Дефинисање рада рачунарских компоненти

Да би корисник могао да дефинише рад неке рачунарске компоненте, треба да отвори пројекат и да учита архитектуру те компоненте.

Након што учита архитектуру компоненте, у директоријуму *Data/Memories* или *Data/Other* отвори датотеку са именом компоненте и суфиксом *.cs*, ако није другачије специфицирао својством *filename.* У овој датотеци, постоји предефинисана метода *Cycle,* која као аргумент прихвата компоненту чији рад се дефинише. У случају да се ради о меморији, компонента ће бити типа *Memory,* ау супротном ће бити типа *OtherComponent*. Дефинисање рада процесора је описано у [1] и овде се томе неће посвећивати пажња.

Корисник може да измени тело ове методе и тиме дефинише скуп наредби које компонента циклично обавља. Како би се кориснику олакшао посао, објекат компоненте који је прослеђен као аргумент пружа следеће методе:

* ***GetPort* -** Ова метода прихвата аргумент типа стринг и враћа порт са именом истим као прослеђени аргумент, уколико такав порт постоји
* ***GetPin* -** Ова метода прихвата аргумент типа стринг и враћа пин са именом истим као прослеђени аргумент, уколико такав пин постоји
* ***GetAllPorts* -** Ова метода враћа низ свих портова компоненте
* ***GetAllPins* -** Ова метода враћа низ свих пинова компоненте
* ***Wait*-**Ова метода реализована је на три начина. Уколико корисник проследи аргументе типа стринг и цео број, онда ова метода блокира извршавање док порт са именом истим као први аргумент не добије вредност другог аргумента. У случају да корисник проследи аргументе типа стринг и *PinValue*, метода блокира извршавање док пин са именом истим као први аргумент не добије вредност другог аргумента, која може бити из опсега *TRUE, FALSE* или *HIGHZ.*Ако корисник проследи један целобројни аргумент, метода блокира извршавање онолико откуцаја системског сата колико је специфицирано аргументом методе
* ***WaitForRisingEdge* -** Ова метода прихвата аргумент типа стринг који представља име пина. Метода блокира извршавање док се не појави узлазна ивица на пину специфицираном са аргументом методе, ако такав постоји
* ***WaitForFallingEdge* -** Ова метода прихвата аргумент типа стринг који представља име пина. Метода блокира извршавање док се не појави силазна ивица на пину специфицираном са аргументом методе, ако такав постоји
* **Дохватање податка са меморијске локације** *-* Уколико се дефинише рад меморије, корисник може прочитати вредност са било које локације у меморији. То може учинити са *memory[address],* где је*memory* аргумент прослеђен методи *Cycle*, а *address* представља адресу меморијске локације са које се жели дохватити податак
* ***GetRegister*--** Ова метода прихвата стринг као аргумент и враћа регистар специфициран аргументом, уколико такав постоји. Може се позвати само у случају да се дефинише рад компоненте општег типа.

Након што је дефинисао рад компоненте, корисник треба да поново учита архитектуру те компоненте, на начин описан у потпоглављу 3.1, да би апликација поново учитала код који описује рад компоненте. Алтернативни начин за остваривање овога је избором опције *Architecture* из главног менија, а затим кликом на опцију *Recompile.* То се може остварити и притиском тастера *F3* на тастатури.

* 1. Дефинисање изгледа рачунарских компоненти

Кориснику апликације пружена је могућност да дефинише визуелну репрезентацију сваке компоненте. Услов за реализацију ове функционалности је отворен пројекат и дефинисана архитектура компоненте.

Дефинисање изгледа компоненте може се остварити на три начина:

* **подразумевани начин** - корисник дефинише само димензије и позиције портова
* **кроз графички интерфејс** - корисник користи графички интерфејс специјално за намену дефинисања изгледа компоненте
* **писањем наредби у коду** - корисник дефинише код који се извршава при , исцртавању компоненте
  + 1. Подразумевани начин дефинисања изгледа рачунарске компоненте

За овај начин, већина својстава је предефинисана.

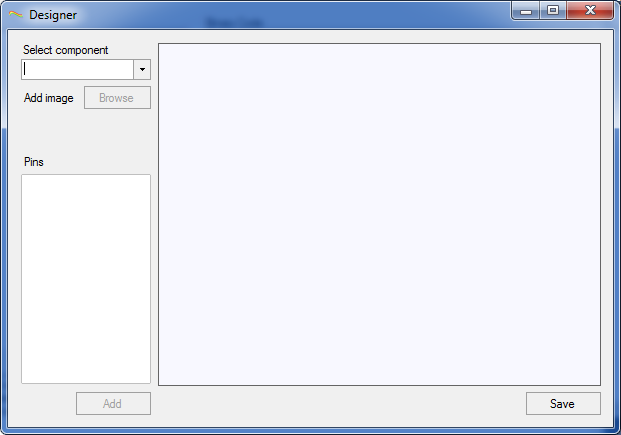
Корисник треба да, при дефиницији архитектуре компоненте (као што је описано у потпоглављу 3.1), дефинише локацију сваког порта. Поред тога, корисник треба да дефинише чвор *dimensions*, којим су обухваћена два подчвора:

* **Чвор *height* -** Вредност овог подчвора дефинише висину компоненте у пикселима
* **Чвор *width* -** Вредност овог подчвора дефинише ширину компоненте у пикселима

Када корисник на овај начин дефинише позиције портова и димензије компоненте, компонента ће се аутоматски исцртати. Компонента ће бити беле боје, са црним оквирима и портовима. Уколико се, при дефиницији система, о чему ће више речи бити у потпоглављу 3.4, изабере овако дефинисана компонента, њени оквири ће постати зелени.

* + 1. Дефинисање изгледа рачунарске компоненте кроз графички интерфејс

За дефинисање изгледа графичке компоненте, постоји посебан графички интерфејс. Да би се отворио овај графички интерфејс, корисник треба да изабера опцију *View* из главног менија апликације, а затим опцију *Designer*. Услов за ово је претходно отворен пројекат и дефинисана архитектура компоненте. На слици 3.3.1. приказан је поменути интерфејс.

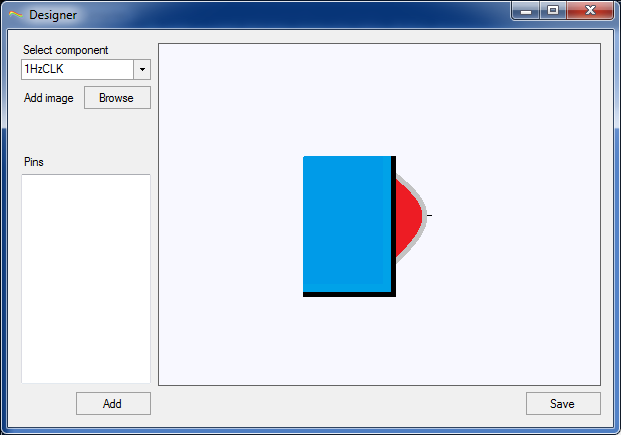


Слика 3.3.1. Графички интерфеjс за дефинисање изгледа компоненти

Да би изабрао компоненту чији изглед жели да дефинише, корисник треба да из падајућег менија са леве стране изабере компоненту. Тада се, у листи пинова са леве стране појављују сви пинови изабране компоненте. Корисник може изабрати пин и кликом на дугме *Add*, пин се појављује на панелу за дефинисање изгледа компоненте и корисник га може притиском и померањем миша превлачити по панелу.

Такође, може се додати слика по жељи тако што се притисне дугме *Browse*. Тада се отвара дијалог за избор слике. По одабиру слике, корисник треба да изабере да ли жели да слика буде транспарентна или не. Уколико одабере да слика буде транспарентна, уклања се сва позадина беле боје. Додавање слике се може поновити више пута. Свака слика се може уклонити са панела десним кликом миша на њену површину и одабиром опције *Remove* у менију који се појави. У случају да постоји више слика које се преклапају, могуће је послати одређену слику иза свих осталих или је довести испред свих осталих слика. То се постиже десним кликом на слику и одабиром опције *Bring to Front* за довођење слике испред осталих слика, односно *Send to Back,* за слање слике иза осталих.

Када је завршено дефинисање изгледа компоненте, иста се може сачувати притиском на дугме *Save*. Тада се изглед компоненте чува и он представља унију свих додатих слика заједно са пиновима. Уколико се, при дефиницији система, о чему ће више речи бити у потпоглављу 3.4, изабере овако дефинисана компонента, цела компонента ће добити јачу нијансу зелене боје. Пример овако дефинисане компоненте приказан је на слици 3.3.2.



Слика 3.3.2. Пример компоненте чији је изглед дефинисан кроз графички интерфејс

* + 1. Дефинисање изгледа рачунарских компоненти писањем наредби у коду

Уколико жели да има потпуну слободу при дефинисању изгледа рачунарске компоненте, корисник то може остварити писањем кода који се извршава при исцртавању компоненте.

При дефинисању архитектуре компоненте, корисник треба да дода чвор *design* и да му дода атрибут *code* чију ће вредност поставити на *true.* Када то уради, при учитавању архитектуре компоненте, аутоматски се генерише и датотека са именом компоненте и суфиксом *Design.cs*. У овој датотеци, налази се метода *DrawComponent* која прихвата два аргумента. Први аргумент је типа компоненте чије се исцртавање дефинише (*CPU, Memory, OtherComponent*). Други аргумент је типа *Graphics*, из библиотеке *System.Drawing .NET* радног оквира и он се користи за исцртавање компоненте.

Поред свих метода описаних у потпоглављу 3.2, на располагању је још и својство *DefaultPen* које пружа информацију о боји којом се исцртавају линије. Ова боја се мења у зависности да ли је компоненте изабрана или не, при пројектовању рачунарског система, о чему ће више речи бити у потпоглављу 3.4.

На слици 3.3.3. је приказан пример исцртавања компоненте у коду.

**public** static void DrawComponent**(**MultiArc\_Compiler**.**OtherComponent component**,** System**.**Drawing**.**Graphics graphics**)**

**{**

component**.**Height **=**20**;**

component**.**Width **=**25**;**

Rectangle rectangle **=new** Rectangle**(**0**,**0**,**20**,**20**);**

graphics**.**FillRectangle**(new** SolidBrush**(**Color**.**White**),** rectangle**);**

graphics**.**DrawRectangle**(**component**.**DefaultPen**,** rectangle**);**

component**.**GetPin**(**"CLK0"**).**Location **=new** Point**(**20**,**10**);**

**}**

Слика 3.3.3. Пример дефинисања изгледа компоненте кроз код

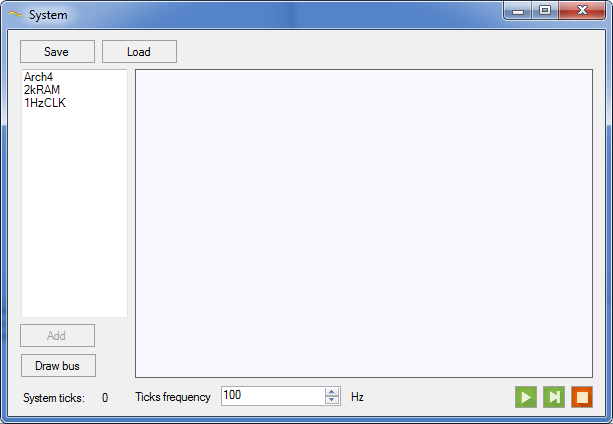
Након што је дефинисао исцртавање компоненте, корисник треба да поново учита архитектуру те компоненте, на начин описан у потпоглављу 3.1, да би апликација поново учитала код који описује рад компоненте. Алтернативни начин за остваривање овога је избором опције *Architecture* из главног менија, а затим кликом на опцију *Recompile.* То се може остварити и притиском тастера *F3* на тастатури.

* 1. Пројектовање рачунарског система

Корисник апликације има могућност да пројектује рачунарски систем. Ово обухвата дефинисање компоненти које ће систем користити, као и веза између њих.

Услов за реализовање ове функционалности је претходно отворен пројекат и учитана архитектура барем једне компоненте.

Да би започео дефинисање система, корисник треба у главном менију апликације да изабере опцију *View,* а затим из подменија који се отвори опцију *System.* Након тога, отвара се нови прозор, приказан на слици 3.4.1.

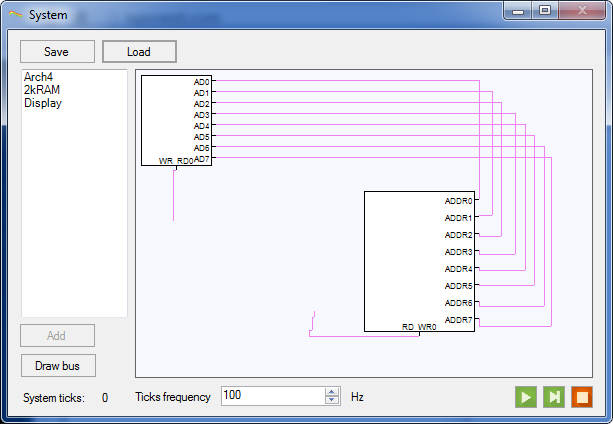


Слика 3.4.1. Прозор за пројектовање система

Пројектовање система је комплексан процес и може се поделити у неколико мањих функционалности, као што су:

* Додавање компоненти
* Дефинисање веза између компоненти
* Додавање магистрале
* Мењање фреквенције рада система
* Чување пројектованог система
* Учитавање претходно сачуваног система

Након обављања свих ових радњи, корисник добија систем који може изгледати као систем са слике



Слика 3.4.2. Пример система

У наставку текста, свака од ових функционалности је кратко објашењена.

* + 1. Додавање компоненти

Да би додао компоненту, корисник треба из листе постојећих компоненти, односно компоненти чију је архитектуру дефинисао, изабере на једну и затим кликне на дугме *Add.* Изабрана компонента се тада додаје на панел системски панел и корисник је може померати по системском панелу превлачењем миша.

Свака компонента, осим компоненти типа процесора, се може додавати више пута. Такође, све компоненте које нису процесор се могу копирати. То се може постићи десним кликом на компоненту која је додата на системски панел, затим одабиром опције *Copy*, а онда десним кликом на празан део системског панела и одабиром опције *Paste*.

Свака компонента се може и уклонити из система, десним кликом на њу, а затим одабиром опције *Remove*.

* + 1. Дефинисање веза између компоненти

Све компоненте се могу повезивати сигналима који повезују њихове пинове.

Левим кликом миша на пин компоненте, почиње се исцртавање сигнала. Док исцртавање траје, левим кликом миша се може направити промена правца исцртавања. Сигнал се може повезати са другим пином, левим кликом на њега, а може се и оставити неповезан са другим, што се чини десним кликом миша.

За сваки сигнал, може се везати други сигнал. То се може урадити именовањем сигнала. Десним кликом миша на сигнал, отвара се мени на коме треба изабрати опцију *Manage names.* Отвара се дијалог у коме се са леве стране види листа свих актуелних имена сигнала. У тој листи, бирањем било ког имена, а затим кликом на дугме *Remove*, изабрано име се уклања из листе имена. Укуцавањем имена у поље *New name,* а затим кликом на дугме  *Add,* ново име се додаје у листу имена. Кликом на дугме *Аpply,* извршене промене се чувају, док се кликом на дугме *Cancel* сачуване промене одбацују. Уколико се дода име које је већ додељено неком другом сигналу, та два сигнала се онда спајају и надаље чине један сигнал. Новодобијени сигнал има листу имена која одговара унији претходно спојених сигнала.

Сваки додати сигнал се може и уклонити. То се чини десним кликом миша на њега, а затим одабиром опције *Remove.*

* + 1. Додавање магистрале

Корисник може додати произвољан број магистрала, а свака магистрала може имати произвољан број битова, тј. садржати произвољан број сигнала.

Да би се додала магистрала, треба кликнути на дугме *Draw bus*. Тада се прелази на мод цртања магистрале, из кога се излази поновним кликом на исто дугме, које је сада постало *Stop*. Док се налази у моду цртања магистрале, левим кликом на било који празан део системског панела, почиње цртање магистрале. Левим кликом миша може се направити промена правца исцртавања. Десним кликом, цртање магистрале се завршава и отвара се дијалог у коме се од корисника тражи да изабере ширину магистрале. Након тога, магистрала је направљена и излази се из мода цртања магистрале. Овако направљена магистрала садржи онолико сигнала колико је корисник изабрао за њену ширину и има аутоматски генерисано име. Сваки од њених сигнала има име исто као и магистрала, са суфиксом [*brojSignala*], где *brojSignala* представља редни број сигнала на магистрали.

Свакој магистрали се може мењати име на исти начин као што се то може чинити и са сигналом, при чему важе иста правила за спајање сигнала као и раније.

Магистрала се може уклонити десним кликом на њу, па одабиром опције *Remove*.

* + 1. Meњање фреквенције рада система

Да би променио фреквенцију, корисник треба да у одељак испод системског панела унесе вредност која представља фреквенцију рада система у Hz.

* + 1. Чување пројектованог система

У сваком тренутку, корисник може сачувати пројектовани систем. Да би то учинио, треба да притисне дугме *Save*, а затим у дијалогу који се отвори да одабере локацију где жели да сачува систем.

* + 1. Учитавање претходно сачуваног система

Да би учитао претходно сачувани систем, корисник треба да притисне дугме *Load* и у дијалогу који се отвори одабере систем који жели да учита. Када ово уради, корисник губи све несачуване измене које је урадио на систему који је до тада пројектовао.

* 1. Симулирање рада рачунарског система

Корисник може симулирати рад рачунарског система. Услов за остваривање ове функционалности јесте претходно дефинисан пројекат и пројектован рачунарски систем.

Да би симулирао рад рачунарског система, корисник треба прво да напише програмски код у едитору кода и да га преведе у машински код. Овај код треба да одговара спецификацији језика који је дефинисао при дефинисању архитектуре процесора. Након што је код написан и преведен у машински код, он се може покретати на процесору који припада дефинисаном рачунарском систему. То се постиже притиском на дугме *Execute*, у доњем десном углу прозора за пројектовање система. Овакво покретање машинског кода извршава програм док не дође до краја или до прве следеће тачке заустављања, које се могу дефинисати у едитору. Други начин покретања је покретање у режиму такт-по-такт. Ово се може постићи притиском на дугме *Execute next step,* одмах поред *Execute* дугмета. Сваки притисак на ово дугме помера извршавање симулације за један такт системског времена, дефинисаног фреквенцијом рада система. Ова два режима се могу комбиновати, па се тако, на пример, првих 5 инструкција може извршити одједном, а затим се код може покретати такт-по-такт, да би се могли пратити неки специфични детаљи рада целог система.

У сваком тренутку, корисник може променити фреквенцију рада система, што се одражава на брзину рада система.

Током симулације, корисник на системском панелу може да прати вредности сигнала, пратећи боје сигнала. Такође, у сваком тренутку корисник може да види који пинови постављају вредности сигнала, као и које вредности постављају. То може урадити тако што направи десни клик на сигнал, а затим изабере опцију *See pin states*. Кориснику се приказује листа свих пинова и вредности које они постављају на изабрани сигнал.

У сваком тренутку, корисник може проверити или изменити вредности регистара компоненти које имају регистре. То може урадити десним кликом на жељену компоненту, а затим одабиром опције *Registers*. Тада се отвара дијалог са приказом вредности свих регистара компоненте, које је могуће изменити.

Корисник може у сваком тренутку проверити или изменити вредности меморијских локација компонената које имају меморијске локације. То може урадити десним кликом на жељен компоненту, а затим одабиром опције *Memory dump*. Тада се отвара дијалог са приказом вредности меморијских локације, које је такође могуће изменити.

1. Реализација

У овом поглављу, биће изнети детаљи имплементације апликације која представља предмет овог рада. Пошто је део апликације развијен као тема дипломског рада [1] у овом поглављу пажња се посвећује искључиво деловима који нису били део дипломског рада.

Апликација је реализована у програмском језилу Ц# и радном оквиру *.NET*, компаније *Microsoft.* За реализацију графичког интерфејса, коришћена је библиотека *Windows Forms*, из радног оквира *.NET.* Како би се поједноставила реализација појединих функционалности, коришћена је библиотека *LINQ (Language Integrated Query)*, а како би се омогућило извршавање динамичког кода коришћена је библиотека *Reflections.* У целој апликацији коришћене су парадигме објектно-оријентисаног програмирања.

Реализација се може поделити у две основне целине, реализацију графичког интерфејса и реализацију логичких функционалности, а свака од њих се даље може поделити на више мањих целина, према функционалностима које имплементирају.

* 1. Реализација графичког интерфејса

Највећи део графичког интерфејса реализованог у оквиру овог рада тиче се имплементације прозора за дефинисање изгледа компоненти и прозора за пројектовање система. У наставку овог поглавља, објашњена је имплементација ова два прозора.

* + 1. Реализација прозора за дефинисање изгледа компоненти

Током имплементације овог дела апликације, постојало је неколико идеја како би дефинисање изгледа компоненти требало да се реализује. Једна од идеја била је пружање могућности кориснику да руком црта компоненте. Међутим, такво решење би било слично апликацији *Paint,* коју данас има сваки оперативни систем *Microsoft Windows*, а имплементација оваквог решења би изискивала превише времена. Други проблем са овим приступом јесте чињеница да би у том случају, свако следеће исцртавање компоненте изискивало да се понове акције које је корисник извео, а цена имплементације овакве функционалности би била превелика. Због тога је одлучено да се изглед дефинише додавањем низа слика, као што је то објашњено у поглављу 0. Овом одлуком се уштедело на времену утрошеном на реализацију овог дела апликације, а да притом није изгубљено на функционалностима, јер корисник може и у апликацији *Paint* да нацрта компоненту на исти начин, да слику сачува и дода је на панел за дефинисање изгледа компоненти.

Главни део прозора за дефинисање изгледа компоненти реализован је у оквиру класе *Designer* која имплементира класу *Form* из библиотеке *WindowsForms*. Објекат ове класе апстрахује прозор који служи за интеракцију са корисником преко корисничких контрола као штп су дугме, клизач, мени итд. Поред ове класе, користе се и друге специфичне контроле као што су*DragAndDropPanel* и *DropableControl,* о којима ће више речи бити у наставку.

Дијаграм класа релевантних за имплементацију прозора за дефинисање изгледа компоненти приказан је на слици 4.1.1.



Слика 4.1.1. Дијаграм класа за имплементацију графичког дела дефинисања изгледа компоненти

У наставку текста ће укратко бити описане класе приказане на претходној слици.

Класа Designer

Као што је већ речено, ова класа садржи највећи део кода који имплементира функционалност дефинисања изгледа компоненти и на неки начин представља централну класу овог дела апликације. Креирањем објекта ове класе, кориснику се приказује прозор за дефинисање изгледа компоненти, приказан на слици 4.3.1. У овој класи дефинисане су методе које представљају ослушкиваче догађаја који се одигравају при интеракцији корисника са прозором за дефинисање изгледа компоненти. Списак ових догађаја, као и начин реаговања на сваки од догађаја, дат је у табели 5.1.1.

Табела 4.1.1. Преглед најважнијих догађаја класе *Designer*

|  |  |
| --- | --- |
| **Догађај** | **Реакција** |
| Промена изабране компоненте | Дозвољава се кориснику да кликне на дугме *Browse*, сви пинови изабране компоненте се додају у листу пинова и дозвољава се кориснику да кликне на дугме *Save*. Такође, брише се интерна листа додатих слика. |
| Одабир слике из дијалога | Кориснику се приказује дијалог за избор да ли жели да изабрана слика постане транспарентна. Након одлуке корисника, креира се објекат класе *ControlWithImage* којој се прослеђује одабрана слика и креирани објекат се додаје на позицију (0,0) на панелу за дизајн компоненте и у интерну листу додатих слика. |
| Клик на дугме *Save* | Листа додатих слика се копира у листу слика компоненте, а затим се чува дизајн компоненте у XML датотеку у којој је дефинисана архитектура компоненте. |
| Одабир *Send to Back* или *Bring to Front* | Изабрана слика се шаље иза или испред свих компоненти и ажурира се њено својство *Level,* како би се при поновном исцртавању компоненте могла задржати дубина сваке слике. |

Од значаја је детаљније објаснити како се реализује чување дизајна компоненте у XML датотеку у којој је дефинисана архитектура компоненте. Када корисник кликне на дугме *Save,* у изабраној компоненти се прво обришу све постојеће слике, ако таквих има. Затим се пролази кроз све додате слике да би се идентификовале границе свих слика, односно пронашао правоугаоник минималне површине који садржи све слике. Ово је потребно да би се могле одредити нове димензије компоненте. Након тога се све слике транслирају тако да се налазе у горњем левом углу панела за дефинисање изгледа компоненте, али да се њихове међусобне позиције не промене, а затим и ажурира регион који ће компонента од сада заузимати. Овај регион се добија као унија региона свих слика и пинова. Овим се омогућује да компонента има и неправилан облик, уколико је сликама тако дефинисано. На самом крају се дизајн компоненте чува тако што се чувају сва релевантна својства у чвору *design* датотеке у којој је дефинисана архитектура компоненте, а у директоријуму *Images* пројектног директоријума чувају се све слике које су додате на панел, како би се могле учитати и по поновном покретању апликације и како не би постојала опасност од брисања слика које су додате. Код који реализује проналажење граница компоненте приказан је на слици 4.1.2.

int upperBorder **=** DesignPanel**.**Height**;**

int lowerBorder **=**0**;**

int leftBorder **=** DesignPanel**.**Width**;**

int rightBorder **=**0**;**

**foreach(**var c **in** \_addedImages**)**

**{**

**if(**c**.**Location**.**X **<** leftBorder**)**

**{**

leftBorder **=** c**.**Location**.**X**;**

**}**

**if(**c**.**Location**.**X **+** c**.**Width **>** rightBorder**)**

**{**

rightBorder **=** c**.**Location**.**X **+** c**.**Width**;**

**}**

**if(**c**.**Location**.**Y **<** upperBorder**)**

**{**

upperBorder **=** c**.**Location**.**Y**;**

**}**

**if(**c**.**Location**.**Y **+** c**.**Height **>** lowerBorder**)**

**{**

lowerBorder **=** c**.**Location**.**Y **+** c**.**Height**;**

**}**

**}**

**foreach(**var p **in** \_addedPins**)**

**{**

**if(**p**.**Location**.**X **<** leftBorder**)**

**{**

leftBorder **=** p**.**Location**.**X**;**

**}**

**if(**p**.**Location**.**X **+** p**.**Width **>** rightBorder**)**

**{**

rightBorder **=** p**.**Location**.**X **+** p**.**Width**;**

**}**

**if(**p**.**Location**.**Y **<** upperBorder**)**

**{**

upperBorder **=** p**.**Location**.**Y**;**

**}**

**if(**p**.**Location**.**Y **+** p**.**Height **>** lowerBorder**)**

**{**

lowerBorder **=** p**.**Location**.**Y **+** p**.**Height**;**

**}**

**}**

\_selectedComponent**.**Size **=new** Size**(**rightBorder **-** leftBorder **+**1**,** lowerBorder **-** upperBorder **+**1**);**

Слика 4.1.2. Проналажење граница компоненте приликом исцртавања

Класа DragAndDropPanel

Класа *DragAndDropPanel* наслеђује класу *Panel* из библиотеке *WindowsForms.* Понашање ове класе у потпуности наслеђује понашање класе *Panel* и наме објеката ове класе јесте постављање на прозор како би корисник могао да додаје компоненте. Ова класа додаје и неке специфичности које класа *Panel* не дефинише.

Класа *DragAndDropPanel* имплементира специфичне ослушкиваче догађаја који се одигравају у оквиру реализоване апликације. У табели 4.1.2. приказани су сви имплементирани ослушкивачи и кратко описане акције које се изврше да би се реаговало на догађај. Објекат ове класе садржан је у објекту класе *Designer* или објекту класе *Clipboard*.

Табела 4.1.2. Преглед најважнијих догађаја класе *DragAndDropPanel*

|  |  |
| --- | --- |
| **Догађај** | **Реакција** |
| Спуштање компоненте на панел | Израчунава се нова локација компоненте. Ова локација се добија као локација на коју је извршено спуштање померена по *х* и *у* оси онолико колико је миш померен у односу на компоненту при подизању компоненте. |
| Притисак тастер миша | На панелу се започиње исцртавање правоугаоника којим се селектују компоненте. То се чини тако што се у приватне променљиве запамти тачка од које се почело селектовање и поставе логичке променљиве које прате да је селектовање у току. |
| Померање миша | Уколико је селектовање у току, исцртава се правоугаоник између тачке у којој је селектовање почело и тачке у којој се налази миш. |
| Пуштање тастера миша | Завршава се процес селектовања и пролази се кроз све компоненте и проверава која од њих је обухваћена правоугаоником за селектовање. На крај се обухваћене компоненте селектују, а необухваћене деселектују. |
| Одабир опције *Remove* у менију или притисак тастера *Delete* на тастатури | Пролази се кроз листу свих компоненти и селектоване компоненте уклањају са панела и из система, уколико је панел део прозора за пројектовање система. |
| Одабир опције *Copy* на менију неке од компоненти или притисак тастера *CTRL* и *C*на тастатури | Пролази се кроз листу свих компоненти и селектоване компоненте се пребацују у интерну листу копираних компоненти. Памти се њихова међусобна релативна позиција како би се при копирању задржао исти распоред свих селектованих компоненти. |
| Одабир опције *Paste* на менију панела или притисак тастера *CTRL* и *V* на тастатури | Све компоненте које су претходно биле запамћене у листу копираних компоненти се клонирају, а затим додају на панел, чувајући њихову међусобну релативну позицију. Уколико је одабрана опција *Paste,* компоненте се позиционирају у односу на тачку на којој је извршен клик миша. а у супротном, у односу на горњи леви угао панела. Додатно, уколико је панел део прозора за пројектовање система, компоненте се додају у систем. |
| Притисак тастера *CTRL* и *А* на тастатури. | Селектују се све компоненте. |

Класе које имплементирају интерфејс ISelectableControl

Објекти класа које имплементирају интерфејс *ISelectableControl* су контроле које се могу селектовати од стране корисника. Овај интерфејс пружа методе за селектовање и деселектовање објеката и утврђивање да ли је објекат потпуно прекривен правоугаоником за селекцију или делимично прекривен правоугаоником за селекцију.

Прва класа која имплементира интерфејс*ISelectableControl* је класа *DropableControl* која се још може и померати по објектима класе *DragAndDropPanel*. Ова апстрактна класа имплементира методе за селектовање и деселектовање компоненте, али не имплементира методе за утврђивање да ли је компонента обухваћена правоугаоником за селектовање. Имплементација ових метода је специфична јер зависи од облика компоненте, па су ове методе дефинисане у конкретним класама изведеним из класе *DropableControl*, као што су *Pin*, *ControlWithImage* и *SystemComponent*.

Класе *Pin*, *ControlWithImage* и *SystemComponent* су класе чији објекти користе за дефинисање компоненти система. О овим класама ће бити више речи у поглављу 4.1.2.

Оно што је важно напоменути везано за ове класе, са становишта графичког интерфејса јесте да све ове класе имплементирају методу *Draw*, која је задужена за исцртавање компоненти.

Објекти класе *Pin* се исцртавају тако што се исцрта кратка линија, која је, у зависности од оријентације пина на компоненти, хоризонтална или вертикална. Објекти класе *ControlWithImage* се исцртавају тако што се исцртава слика која им је додељена. Објекти класе *SystemComponent* се исцртавају тако што се утврди да ли садрже објекте класе *ControlWithImage.* Ако садрже објекте класе *ControlWithImage,* онда се редом, један по један, исцртавају сви објекти класе *ControlWithImage*, водећи при том рачуна о очувању дубине сваког од ових објеката. Уколико објекат класе *SystemComponent* не садржи објекте класе *ControlWithImage,* онда се утврђује да ли је корисник дефинисао кроз код цртање ове компоненте, па ако јесте, извршава се код који је корисник дефинисао, а ако није, онда се исртава подразумевани изглед компоненте, водећи рачуна о димензијама и распореду пинова дефинисаним при дефинисању архитектуре компоненте.

* + 1. Реализација прозора за пројектовање система и симулирање рада система

Главни део прозора за пројектовање система реализован је у оквиру класе *Clipboard* која имплементира класу *Form* из библиотеке *WindowsForms*. Објекат ове класе апстрахује прозор који служи за интеракцију са корисником преко корисничких контрола као штп су дугме, клизач, мени итд..

Прозор за пројектовање читавог система изгледа слично као и прозор за дефинисање изгледа компоненти. Пошто је интераговање корисника са системским компонентама слично као интераговање корисника са сликама које је додао, могу се изнова користити класе *DragAndDropPanel, ISelectableControl, Pin, NonPinDropableControl,* и *SystemComponent ControlWithImage*. Дијаграм класа које су од значаја за овај део система приказан је на слици 4.1.3.



Слика 4.1.3. Дијаграм класа за реализацију графичког дела пројектовања система и симулирања рада система

У наставку текста биће дат преглед класа које су од значаја за имплементацију графичког дела пројектовања система и симулацију рада система.

Класа Clipboard

Класа *Clipboard* је централна класа овог дела функционалности. Ова класа је изведена класе *Form* и она апстрахује понашање прозора за пројектовање система. Креирањем објекта ове класе, кориснику се приказује прозор за пројектовање система. Конструктор ове класе добија као аргумент листу објеката класе*SystemComponent*, која представља листу могућих рачунарских компенти које се могу додати систему. Такође, кроз аргументе, конструктору се прослеђује и објекат класе *UserSystem*, који је задужен за функционалности пројектовања система и симулирања рада система.

Садржи објекат класе *DragAndDropPanel*, која је описана у 4.1.1. Међутим, поред функционалности које пружа објекат класе *DragAndDropPanel*, класа*Clipboard* дефинише и додатне методе ослушкиваче догађаја који се дешавају када корисник интерагује са панелом, које су специфичне само за пројектовање система. Поред њих, ова класа дефинише и додатне методе ослушкиваче који се извршавају при интеракцији корисника са другим деловима система.

У табели 4.1.3. дат је преглед догађаја које објекат класе *Clipboard* ослушкује и описано је како реагује на сваки од њих.

Табела 4.1.3. Преглед најважнијих догађаја класе *Clipboard*

|  |  |
| --- | --- |
| **Догађај** | **Реакција** |
| Клик на дугме *Add* | Компонента која је означена у листи компоненти се узима из интерне листе компоненти и додаје систему. Уколико је компонента већ додата, пре додавања систему се изабрана компонента клонира. На крају се додаје на панел за пројектовање система. |
| Клик на пин неке компоненте | Уколико већ није започето цртање сигнала или магистрале, што се може видети на основу вредности интерних променљивих, започиње се цртање сигнала. Уколико је цртање сигнала већ у току, сигнал који се црта се повезује са кликнутим пином и цртање се завршава. |
| Померај миша преко панела | Уколико је у току цртање сигнала или магистрале, исцртавају се две нормалне линије на панелу, које заједно спајају тачку од које је започето цртање до тачке у којој се налази миш. Уколико није, догађај се прослеђује панелу на обраду. |
| Клик на панел | Уколико је клик направљен левим тастером миша, проверава да ли је у току цртање сигнала или магистрале. Ако јесте, прави се прелом на тачки клика и цртање наставља од тачке клика. Уколико је клик направљен десним тастером, завршава се цртање сигнала или магистрале. |
| Клик на дугме *Execute* | Покреће се извршавање симулације позивом методе *Execute* главне форме апликације. |
| Клик на дугме *Next step* | Цела симулација се помера за један такт позивом методе *ExecuteTickByTick* главне форме апликације и повећава број тактова система, што ажурира број приказан у доњем левом углу прозора |
| Клик на дугме *Stop* | Цела симулација се зауставља позивом методе *EndWorking* објекта класе *UserSystem*. |
| Клик на дугме *Save* | Отвара се дијалог за избор локације на коју се чува систем. Након одабира локације, до сада дефинисани систем се чува, чувањем најбитних својстава у *XML* документ на изабраној локацији. |
| Клик на дугме *Load* | Отвара се дијалог за избор датотеке из које се учитава систем. Након одабира датотеке, брише се до сада дефинисани систем и исцртава учитани. |
| Промена изабране компоненте | Уколико је нека компонента изабрана, кориснику се омогућује клик на дугме *Add,* док се у супротном забрањује. |
| Клик на дугме *Draw bus* | У интерној променљивој памти се да се тренутно налази у режиму цртања магистрале, а текст дугмета постаје *Stop*. Када корисник следећи пут кликне на панел, почеће цртање магистрале, које се завршава на исти начин као и цртање сигнала. Уколико корисник поново кликне на дугме, прекида се цртање магистрале и до сада нацртана магистрала уклања. |

Од значаја је нагласити како је реализовано цртање сигнала или магистрале, који ће се у наставку текста звати једном речју конектор. У случају да се ради о цртању сигнала, цртање започиње онда када корисник левим тастером миша кликне на пин неке од компоненти, док се, у случају да се ради о цртању магистрале, цртање започиње уколико корисник направи леви клик на панелу док је у режиму цртања магистрале. Када се започне цртање конектора, у интерним променљивама памте се координате тачке у којој се започело цртање и креира објекат класе *Connector*, у којем ће се памтити својства од значаја за цртање овог конектора. Како је класа *Connector* апстрактна, овај објекат ће заправо бити објекат класе једне од две класе изведене из класе *Connector*. У случају да се ради о цртању сигнала, то ће бити објекат класе *Signal* и он ће се одмах повезати са објектом класе *Pin* на који је кликнуто, а у случају да се ради о цртању магистрале, то ће бити објекат класе *Bus*. При померању миша по панелу, у сваком тренутку се прати тачка у којој се миш налази и исцртавају се линије које повезују почетну тачку и тачку у којој се миш налази. Уколико се са (*хр*, *ур*)означе координате почетне тачке, а са (*хt*, *yt*)координате тачке у којој се налази миш, онда се може рећи да се прво исцрта линија која повезује тачке (*хр*, *ур*) и(*хt*, *yр*), а затим линија које повезује тачке (*хt*, *yр*) и (*хt*, *yt*). Уколико током овог цртања корисник направи леви клик на панелу, онда се нацртане линије запамте у променљивама типа *Line* које се додају конектору који се тренутно црта, а затим се одређује нова почетна тачка као тачка у којој је направљен леви клик и поступак цртања наставља као и до сада. Уколико је, пак, током овог цртања корисник направио десни клик на панелу, онда се цртање конектора завршава тако што се последње две нацртане линије бити додате конектору. Специјално, уколико се ради о цртању сигнала, онда корисник може да заврши цртање сигнала и левим кликом на неки од пинова. Тада се пин на који је кликнуто додаје у листу пинова сигнала који се исцртава, чиме се заправо пинови повезују.

Класе које имплементирају интерфејс ISelectableControl

Класе које имплементирају интерфејс су описане у одељку4.1.1 и њихови објекти се на исти начин користе и у овом делу апликације, са становишта графичког интерфејса, па им се у овом одељку неће посвећивати већа пажња. За овај део апликације, ове класе су значајније са становишта логичких функционалности, али ће о томе више речи бити у потпоглављу 4.2.

Једини значај ових класа, са становишта реализације графичког дела симулације система, огледа се у понашању објеката класе *Pin*, који представљају пинове преко којих се компоненте повезују са сигналима.

Сваки пин има референцу на сигнал на који је повезан, ако такав постоји. Када се промени вредност пина, он, у зависности од вредности коју је добио, мења боју сигнала.

Класа Line

Класа *Line* је класа чији се објекти користе за графичку репрезентацију конектора. Ова класа изведена је из класе *Control* библиотеке *WindowsForms*, чији објекти могу интераговати са корисником. Значај ове класе огледа се у чињеници да се њени објекти користе за исцртавање конектора на панелу за пројектовање система. Сваки конектора садржи низ објеката класе *Line* који представљају линије преко којих је конектор представљен графички.Исцртавање конектора је заправо исцртавање свих његових линија. Такође, класа *Line* имплементира методе за ослушкивање догађаја корисника, који из угла корисника изгледају као догађаји са конектором. Тако, класа *Line* обрађије догађаје клика мишем, преласка мишем преко линије, које углавном прослеђују одговорности конектору који их садржи и на који имају референцу. Такође, ова класа дефинише методу *draw*, која се извршава када се линија исцртава на панелу за пројектовање система, као и методе за селектовање и деселектовање.

Класа Connector

Класом *Connector* представљају се објекти конектора. Ова класа је апстрактна и из ње су изведене класе које представљају сигнале и магистрале, *Signal* и *Bus,* респективно. Ове класе су значајне за симулирање рада система. Ту се посебно истиче значај класе *Signal,* која дефинише методу *SetColor* коју објекти класе *Pin*позивају када им се вредност промени. Сигнал тада информише све своје линије да треба да промене своју боју, па се приликом исцртавања, кориснику сигнали приказују одговарајућом бојом која репрезентује вредности пинова.

Класа UserSystem

Класа *UserSystem* је од много већег значаја за имплементацију логичких функционалности него за имплементацију графичког интерфејса. У погледу имплементације графичког интерфејса, објекат ове класе је значајан јер је он објекат који се гради при пројектовању система и чије се методе извршавају при извршавању симулације, па је из тог разлога приказан на дијаграму класа.

* 1. Реализација логичких функционалности

Реализација логичких функционалности покрива много већи део него што је случај са реализацијом графичког интерфејса. Овај део се може поделити према групама функционалности које су имплементиране и то на следећи начин:

* дефинисање архитектуре и рада рачунарских компоненти
* дефинисање изгледа рачунарских компоненти
* пројектовање рачунарског система
* симулирање рада рачунарског система

У наставку текста биће детаљно објашњена имплементација сваке од ових група функционалности.

* + 1. Дефинисање архитектуре и рада рачунарских компоненти

Дефинисање архтектуре рачунарских компоненти углавном подразумева парсирање *XML* датотека и постепено грађење објеката рачунарских компоненти и за ову функционалност углавном се користи библиотека *Xml* радног оквира *.NET*, док се за дефинисање рада рачунарских компоненти углавном користе функционалности које пружа библиотека *Reflection* из радног оквира *.NET*.

На слици 4.2.1. приказан је дијаграм класа значајних за ову групу функционалности.



Слика 4.2.1. Дијаграм класа од значаја за дефинисање архитектуре и рада компоненти

У наставку текста биће објашњена улога сваке од класа.

Класа SystemComponent

Класа *SystemComponent* представља рачунарску компоненту. Из ње су изведене класа *CPU*, која представља процесор и класа *NonCPUComponent*, која је апстрактна и из које су изведене класе *Memory* и *OtherComponent*, које представљају компоненте типа меморије и општег типа, респективно.

Објекат класе *SystemComponent* садржи листу објеката класе *Port*, који представљају све портове које компонента има. Ово омогућује овој класи да имплементира методе *GetPort*, *GetPin*, *GetAllPorts*, *Wait*, *WaitForRisingEdge* и *WaitForFallingEdge*, које се користе при дефинисању рада компоненте и о којима је било речи у потпоглављу 3.2.

Поред поменутих метода, класа *SystemComponent* декларише апстрактне методе *Load* и *CompileCode*, које су дефинисане у класама изведеним из класе *SystemComponent.* Метода *Load* се извршава када се учитава архитектура компоненте из датотеке која је описује, док се метода *CompileCode* извршава када се преводи код који описује рад компоненте. Ова класа имплементира методу *ProcessDesignNode*, која се извршава након методе *Load*, а која за циљ има да парсира чвор *design* из датотеке која описује архитектуру компоненте. Ова метода није специфична ни за једну од класа изведених из класе *SystemComponent,* па је из тог разлога реализована у оквиру ње. Заједничко за све класе изведене из класе *SystemComponent* у методи *Load* јесте чињеница да се, приликом позива ове методе, креирају објекти типа *Port*, који представљају портове и о којима ће више речи бити у наставку.

Од значаја је на овом месту објаснити процес учитавања архитектуре компоненте и превођења кода који описује рад компоненте, јер је начин реализације овог процеса сличан за све класе изведене из класе *SystemComponent* и разликује се једино по реализацијама методе *Load*. Када се учита датотека која описује архитектуру, позива се метода *Load*, која је специфична за сваку класу изведену из класе *SystemComponent* и о којој ће бити више речи у делу текста везаном за изведене класе. Метода *Load* парсира један по један чвор из датотеке која описује архитектуру и постепено гради компоненту. На крају се позива метода *ProcessDesignNode*, о којој ће више речи бити у одељку 4.2.2, а затим и метода *CompileCode,* која учита код из датотеке специфициране у оквиру описа архитектуре, динамички га преведе и запамти тако добијени бинарни код у интерној променљивој, како би могла да га изврши при раду.

При имплементацији процеса учитавања архитектуре, постављало се питање у ком тренутку треба извршити методу *CompileCode*, јер је ово временски захтеван процес, па с тога може бити критичан за перформансе апликације. Поред превођења током учитавања архитектуре, разматране су опције да се то чини сваки пут када се овај код извршава и једном, при првом извршавању кода. Прва опција је одбачена јер би значајно покварила рад система. Друга опција се наметала као добра, јер се чинило да ће на тај начин перформансе најмање трпети, с обзиром да, у тренутном случају, отварање пројекта узима доста времена јер се тада преводе кодови свих компоненти, од којих се неке врло често неће уопште користити. Међутим, уколико би се код преводио при првом извршавању кода, онда би постојала неконзистенција између извршавања кода први пут и свих осталих извршавања, па би деловало да компонента не ради конзистентно. Кратко, разматрана је и трећа опција, да се код преведе при додавању компоненте у систем, али, како је ова операција праћена са доста промена у графичком интерфејсу, то би цео тај процес успорило и рад са системом би био мање пријатан за корисника.

Класа *SystemComponent* имплементира методу *redraw* која се извршава онда када треба исцртати компоненту. Ова метода такође не доноси никакве специфичности у изведеним класама, па је реализована у оквиру класе *SystemComponent*. И ова метода ће бити детаљније објашњена у одељку 4.2.2.

Класа NonCPUComponent

Класа *NonCPUComponent* је апстрактна и из ње су изведене класе рачунарских компоненти које не представљају процесоре, *Memory* и *OtherComponent.*Ова класа дефинише методе *StartWorking*, *Run* и *StopWorking*. Ове методе су значајне јер се извршавају при симулацији рада компоненте, о чему ће више речи бити у оквиру одељка 4.2.4.

Класа Memory

Објекти класе *Memory* су репрезентација меморијских компоненти. Ова класа садржи својства неопходна опонашање рада меморије, као што су *RomStart, RomEnd, RamStart, RamEnd, AuSize* и *Size,* чије се вредности иницијализују у методи *Load*. Ова својства дефинишу својства меморијске компоненте и представљају почетну адресу РОМ меморије, крајњу адресу РОМ меморије, почетну адресу РАМ меморије, крајњу адресу РАМ меморије и величину адресибилне јединице.

Поред ових својстава, ова класа имплементира индексер који служи за постављање и читање вредности меморијских локација. Садржај меморије налази се у бинарној датотеци која се чува на диску, како би било могуће дозволити меморије са већим капацитетом. При читању вредности, овај индексер рачуна локацију унутар датотеке у којој се налази садржај меморије релативно у односу на последњу приступљену локацију, како би померање курсора било што је могуће брже, и позиционира се на ту локацију. Затим, почевши од те локације, чита онолико бајтова колика је вредност својства *AuSizе.* Прочитани бајтови се враћају као резултат позива индексера. При упису, на исти начин се одређује почетна локација уписа, само што се сада, бајтови који су као вредност прослеђени индексеру уписују на *AuSizе* наредних локација у бинарној датотеци.

Класа OtherComponent

Класа *OtherComponent* представља реализацију компоненте општег типа. Ова класа се понаша слично као класа *Memory*, са разликом што су њена својства садржана у објекту класе *ArchConstants*, која поред општих својстава, дефинише низ објеката типа *Register*, који се понашају као регистри и који се иницијализују при учитавању архитектуре компоненте општег типа.

Класа Port

Класа *Port* представља реализацију порта компоненте. При дефинисању архитектуре компоненте, иницијализују се портови те компоненте. Сваки порт је дефинисан именом, позицијом и врстом. Позиција порта представља вредност из скупа вредности { *LEFT, RIGHT, UP, DOWN* }, дефинисаним набројивим типом *Position.* Ова вредност садржи информацију о страни компоненте на којој се порт налази. Врста порта представља вредност из скупа вредности { *IN, OUT, INOUT* }*,* дефинисаним набројивим типом *Kind.* Ова вредност говори да ли је порт улазни, излазни или улазно-излазни, респективно. При иницијализацији порта, у зависности од величине порта, иницијализује се низ објеката класе *Pin*, који представљају пинове порта.

* + 1. Дефинисање изгледа рачунарских компоненти

У потпоглављу 3.3. већ је показано да се на три начина може дефинисати изглед компоненти:

* подразумевани начин
* кроз графички интерфејс
* писањем наредби у коду

За реализацију ове функционалности, дијаграм најзначајнијих класа приказан је на слици 4.2.2.



Слика 4.2.2. Дијаграм класа значајних за дефинисање изгледа компоненти

Од највећег значаја за овај део функционалности најзначајније су класа *SystemComponent* и *ControlWithImage*, па ће у наставку бити речи о њима.

Класа SystemComponent

Класа *SystemComponent* учествује у процесу дефинисања изгледа рачунарске компоненте на који год да се начин обавља дефинисање изгледа, од поменута три начина. При исцртавању компоненте, позива се метода *redraw*, која исцртава компоненту, у зависности како је изглед компоненте дефинисан.

У случају да се изглед компоненте дефинише на подразумевани начин, при учитавању архитектуре компоненте, учитају се својства која дефинишу димензије компоненте, а затим и величину и позицију сваког од портова, за које се онда креира одређени број пинова. При исцртавању компоненте, исцртава се правоугаоник димензија дефинисаних кроз опис архитектуре компоненте, а затим се пролази кроз листу пинова сваког од портова и они исцртавају један по један. При томе се води рачуна да пинови који се налазе са исте стране компоненте буду на међусобној удаљености одређеној количником величине те странице нацртаног правоугаоника и броја пинова на тој страни.

У ситуацијама у којима је изглед компоненте дефинисан кроз графички интерфејс, корисник онда треба да, на начин описан у одељку 3.3.2, обезбеди листу слика које дефинишу како се компонента исцртава, заједно са њиховим позицијама и позицијама пинова. Када је то урађено, објекту класе *SystemComponent* додају се објекти класе *ControlWithImage*, од којих сваки има слику и позицију у оквиру компоненте. Уочава се регион сваког од ових објеката, заједно са регионима пинова и од уочених региона прави се један регион као унија свих ових региона, који шредставља регион компоненте чији се изглед дефинише. Када се овако дефинисана компонента сачува, важно је ажурирати датотеку у којој је описана архитектура компонете, односно чвор *design*, додајући му подчворове *image* за сваку од слика, од којих сваки садржи информације о датотеци у којој је слика сачувана, позицији слике и дубини слике. Такође, свака слика се копира у директоријум *Images* пројектног директоријума, како не би постојала опасност од евентуалног брисања слике која је иницијално прослеђена.Уколико при исцртавању компоненте, компонента садржи барем један објекат класе *ControlWithImage*, исцртавање се одвија тако што се исцртавају сви објекти класе *ControlWithImage,* што ће детаљније бити описано у наставку.

Уколико је изглед компоненте дефинисан у коду, онда се, при парсирању датотеке са описом архитектуре, учитава садржај датотеке у којој се налази дефинисани код и он се динамички преводи на исти начин као што је то учињено у одељку 4.2.1, и добијени машински код се чува у интерној променљивој како би се могао касније извршити. При исцртавању компоненте, извршава се машински код сачуван у интерној променљивој.

И у овом случају, појавило се питање када треба динамички превести дефинисани код за исцртавање компоненте. Одлука је донета на исти начин као и у одељку 4.2.1.

Класа ControlWithImage

Ова класа је од значаја само уколико се изглед компоненте дефинише кроз графички интерфејс. Тада се, са сваким додавањем нове слике, креира један објекат класе *ControlWithImage*, који садржи низ бајтова који представљају учитану слику и има информације о позицији слике и нивоу слике који се може мењати.

* + 1. Пројектовање рачунарског система

Пројектовање рачунарског система обавља се кроз графички интерфејс. Графичко представљање овако дефинисаног система описано је у потпоглављу 4.1.2, док ће у овом потпоглављу бити више речи о процесима који се одвијају у позадини.

На слици 4.2.3. приказан је дијаграм класа значајних за ову групу функционалности.



Слика 4.2.3. Дијаграм класа значајних за пројектовање система

У наставку текста биће приказана улога сваке од класа приказаних на дијаграму са претходне слике у процесу пројектовања система.

Класа UserSystem

Класа *UserSystem* представља имплементацију целокупног рачунарског система. Ова класа садржи низ референци на рачунарске компоненте, односно низ референци на објекте класа изведених из класе *SystemComponent*, као и низ објеката класа изведених из класе *Connector*, који представљају конекторе, односно сигнале и магистрале. Тако се, када корисник дода компоненту рачунарском систему, она додаје променљивој типа *UserSystem*. Исто важи и за сигнал или магистралу.

При чувању пројектованог система, након што корисник одабере локацију на којој ће се чувати опис система, позива се метода *SaveSystemToFile*, која чува сва битна својства, односно информације о додатим компонентама и конекторима у *XML* формату у датотеци коју је корисник специфицирао. При учитавању претходно сачуваног система, одвија се обрнут процес.

Класа SystemComponent и класе изведене из ње

Као што је већ раније наглашено, класа *SystemComponent* је апстрактна и класе изведене из ње представљају рачунарске компоненте. У процесу пројектовања система, објекти ових класа се понашају на исти начин, па су из тог разлога објашњени заједно.

При пројектовању рачунарског система, објекти ових класа не обављају никакав смислени рад, већ служе као анемични објекти, из перспективе ове функционалности, иако то у суштини нису. Њихова сврха је да се, преко низа референци на њих и веза између њих, дефинише читав рачунарски систем, али се њихово додавање у систем и повезивање обавља преко објеката других класа. Додавање у систем се обавља преко објекта класе *UserSystem*, а повезивање преко објеката класа *Pin* и *Connector.*

Класа Port

Класа *Port* је класа чији објекти представљају портове рачунарских компоненти. Сваки порт садржи одређени број пинова. Ова класа се не користи директно, већ се користе њени пинови.

Класа Pin

Класа *Pin* представља пин, преко којих се компоненте могу повезивати. При додавању сигнала, сигнал добија референцу на пин и обрнуто, пин добија референцу на сигнал. При чувању описа система, чувају се информације о овим везама. Сваки пин се идентификује именом, које се изводи из имена порта и редног броја пина. Ове информације су битне, да би се, при поновном учитавању система, могло знати како су компоненте повезане, односно, који пинови су повезани са којим сигналом.

Класа Connector

Класа *Connector* је апстрактна и класе изведене из ње, *Bus* и *Signal*, које представљају магистралу и сигнал, респективно. Већи део њене имплементације значајан је за реализацију графичког дела ових функционалности, док су делови значајни за пројектовање система имплементирани у оквиру класа *Signal* и *Bus*. Од значаја је напоменути да објекти ове класе садрже референцу на рачунарски систем коме припадају и да садрже низ имена, који ће се користити у класама изведеним из ње. Такође, класа *Connector* декларише апстрактну методу *AddName*, која се извршава када се дода име објекту ове класе и која је имплементирана у изведеним класама. Ова метода је значајна јер она диктира како ће се сигнали и магистрале спајати када се промени име неком од њих.

Класа Signal

Класа *Signal* представља имплементацију сигнала. Сваки објекат ове класе референцира низ пинова на који је повезан. Приликом чувања система у датотеку, чувају се информације о пиновима на који су сигнали повезани.

Важно је објаснити како је реализовано спајање сигнала. Када корисник дода име неком сигналу, извршава се метода *AddName*.Ова метода прво додаје име у листу имена, а затим пролази кроз све сигнале дефинисане у систему и проналази све оне које у листи својих имена имају новодефинисано име. Уколико се пронађе барем један такав сигнал, позива се метода *MergeSignals*, која врши спајање сигнала и којој се прослеђује листа пронађених сигнала заједно са сигналом коме је додато ново име. Код ове методе приказан је на слици 4.2.4.

var signal**=**signalsToMerge**.**Last**();**

var system **=** signal**.**System**;**

var mergedSignal **=new** Signal**(**signal**.**System**);**

**foreach(**var s **in** signalsToMerge**)**

**{**

**foreach(**var name **in** s**.**Names**)**

**{**

**if(!**mergedSignal**.**Names**.**Contains**(**name**))**

**{**

mergedSignal**.**Names**.**AddLast**(**name**);**

**}**

**}**

**foreach(**var line **in** s**.**Lines**)**

**{**

**if(!**mergedSignal**.**Lines**.**Contains**(**line**))**

**{**

mergedSignal**.**Lines**.**AddLast**(**line**);**

**}**

**}**

**foreach(**var pin **in** s**.**pins**)**

**{**

**if(!**mergedSignal**.**pins**.**Contains**(**pin**))**

**{**

mergedSignal**.**pins**.**AddLast**(**pin**);**

pin**.**Signal **=** mergedSignal**;**

**}**

**}**

**if(**s**.**Bus **!=null)**

**{**

var node **=** s**.**Bus**.**Signals**.**Find**(**s**);**

s**.**Bus**.**Signals**.**AddBefore**(**node**,** mergedSignal**);**

s**.**Bus**.**Signals**.**Remove**(**s**);**

**}**

system**.**Signals**.**Remove**(**s**);**

**}**

system**.**Signals**.**AddLast**(**mergedSignal**);**

Слика 4.2.4. Спајање сигнала по имену

Најпре се идентификује сигнал који је последњи додат у листу, јер је то сигнал коме је додато ново име, и искористи се објекат класе *UserSystem* да се дода новонаправљеном сигналу који ће представљати сигнал који се добија спајањем свих сигнала и који се чува у променљивој *mergedSignal*. Након тога, пролази се кроз листу свих сигнала прослеђених овој методи и из листе имена сваког од сигнала се додају имена сигналу *mergedSignal*, уколико већ нису додата. Исти поступак се обавља и са линијама и са пиновима сваког сигнала. Уколико је неки од сигнала део магистрале, из њене листе сигнала се уклања тај сигнал и додаје сигнал *mergedSignal*. Сваки сигнал се уклања из система, и на крају се у систем додаје новонаправљени сигнал.

Класа Bus

Класа *Bus* представља имплементацију магистрале. Сваки објекат ове класе одређен је низом сигнала којих има онолико колика је ширина магистрале. При додавању магистрале у систем, додаје се онолико сигнала колика је ширина магистрале, од којих ће сваки сигнал имати име *imeMagistrale*[*redniBroj*], где *imeMagistrale* представља име магистрале, док је *redniBroj* редни број сигнала у листи свих сигнала.

При додавању имена магистрали, позива се метода *AddName*. У овој методи, додато име се додаје у листу имена магистрала, а затим се истоимена метода позива за сваки од сигнала магистрале, али јој се као аргумент прослеђује променљива са вредношћу *dodatoIme*[*redniBroj*], где је *dodatoIme* вредност променљиве прослеђене методи *AddName*магистрале, а *redniBroj* редни број сигнала на магистрали. Уколико буде потребно, извршиће се спајање сигнала, на исти начин као што је описано у претходном пододељку.

* + 1. Симулирање рада рачунарског система

Симулирање рада рачунарског система је урађено модуларно. Сваки део система, који чине компоненте и везе између њих, има своје дефинисано понашање којег је само он свестан. Дијаграм класа од значаја за симулирање рада система приказан је на слици 4.2.5.



Слика 4.2.5. Дијаграм класа од значаја за симулирање рада система

У наставку текста ће бити приказана улоге сваке од класа са претходно приказаног дијаграма у реализовању функционалности симулирања рада система.

Класа UserSystem

Класа *UserSystem* је централна класа ове функционалности. Објекат ове класе представља рачунарски систем. Због тога, ова класа пружа методе *StartWorking*, *StartWorkingTickByTick* и *EndWorking*, које се покрећу када треба покренути систем, када треба извршити следећи такт и када треба зауставити рад система, респективно.

При покретању рада система у било ком моду, креира се нит која извршава методу *run*. Метода *run,* у зависности од фреквенције рада система, повећава број откуцаја системског такта, односно један циклус представља један откуцај такта система. Све ово се извршава уколико је бинарна променљива *runninng* позитивна и уколико је вредност бинарне променљиве *tickByTickMode* негативна. При покретању методе *StartWorking,* променљива *runninng* се поставља на позитивну вредност. Метода *StartWorking* као аргумент прихвата бинарну вредност *shouldWorkInTickByTickMode*, која се преписује у променљиву *tickByTickMode*, чиме се означава да ли се стално жели инкрементирати системски такт. Такође, уколико извршавање није већ покренуто, ова метода покреће рад процесора, тако што проследи објекат процесора објекту класе *Executor*, а затим и рад свих осталих компоненти. Позив методе *StartWorkingTickByTick* је ништа друго до позив методе *StartWorking* којој се као параметар *shouldWorkInTickByTickMode* прослеђује позитивна вредност.

Од значаја је и објаснити улогу методе *Wait*. Ова метода зауставља извршавање неке компоненте одређени број тактова система. Ову методу позивају рачунарске компоненте како би се синхронизовале са осталим компонентама. Објекат класе *UserSystem* прати интерну листа времена на која нека од компоненти система чека. При позиву методе *Wait*, додаје се такт у коме се извршавање треба наставити у листу времена чекања. Овај такт се израчунава као збир тренутног такта система и броја тактова колико треба сачекати. Након додавања такта у листу чекања, блокира се рад компоненте. Током рада система, стално се инкрементира такт система. При овом инкрементирању, води се рачуна да ли у листи времена чекања постоји нови такт који се добија инкрементирањем. Ако постоји, све компоненте које чекају на тренутну вредност системског такта настављају са извршавањем.

Класа SystemComponent

Класа *SystemComponent*, осим што пружа методе које се могу користити при дефиницији рада компоненте набројане у 3.2, нема превелики значај за симулацију рада система. Много већи значај имају класе изведене из ње.

Класа NonCPUComponent и класе изведене из ње

Са становишта симуларије рада система, класа *NonCPUComponent* имплементира најважније методе, док класе изведене из ње користе њену имплементацију. Оне су значајније за дефиницију рада компоненти. Због тога је овде објашњено како функционише рад објеката класа изведених из класе *NonCPUComponent*, у шта спадају *Memory* и *OtherComponent.*

Да би се покренуло извршавање стварног посла, позива се метода *StartWorking*, која креира нову нит и покреће је, уједно постављајући вредност променљиве *running* на позитивну вредност. Ова нит извршава методу *Run*, која циклично испитује вредност логичке променљиве *running* и уколико је њена вредност позитивна, извршава код специфичан за рад компоненте, тако што, из интерне променљиве у којој памти машински код, покрене методу *Cycle*, о којој је било речи у потпоглављу , док се у случају да је вредност променљиве *running* не ради ништа. Позивом методе *StopWorking*, зауставља се извршавање стварног посла, постављајући променљиву *running* на негативну вредност.

Класа CPU

Класа *CPU*представља компоненту типа проццесора и има сличну улогу као и класе изведене из *NonCPUComponent*, са разликом што њено извршавање не обавља директно објекат ове класе, већ објекат класе *Executor*, који је чврсто спрегнут са објектом класе *CPU.* При симулацији рада процесора, циклично се не извршава се само једна метода коју је корисник дефинисао, већ комплексан код који користи бројач корака *PC* да израчуна адресу следеће инструкције, провери да ли је наишао на тачку паузирања програма, па ако јесте, чека на акцију корисника. Када се извршавање настави инкрементира се *PC* тако да показује на адресу следеће инструкције која ће се извршити, уколико се ова вредност не промени током извршавања инструкције. Затим се идентификује која је следећа инструкција коју треба извршити, а затим извршава динамички дефинисан код за дохватање операнада инструкције, извршавање инструкције и затим за чување резултата. Овај код дефинисао је корисник кроз неколико метода које је дефинисао при дефинисању архитектуре процесора.

1. Тестирање

У овом поглављу биће више речи о начину како је читава апликација тестирана и какви су закључци донети у погледу прецизности и брзине рада.

За тестирање апликације, креирана су два различита пројекта. Први пројекат је једноставнији и од компоненти садржи процесор, меморију и две компоненте општег типа - дисплеј и генератор сигнала такта.

За генерисање процесора, прво је написана датотека која описује његову архитектуру, приказана на слици, на начин објашњен у [1]. Процесор садржи 16 регистара опште намене. Овде је битно нагласити да овај процесор подржава само инструкију *load*, у формату   
*ld registar, operand*, која вредност задату кроз *operand* смешта у регистар *registar.* За ту сврху, реализована су два начина адресирања, регистарско директно и непосредно. На слици 5.1.1. може се видети дефиниција регистарског директног начина адресирања.

<regdir>

<name>**regdir**</name>

<file>**regdir.cs**</file>

<result>**w**</result>

<expression>

<registers\_group>**general\_purpose**</registers\_group>

</expression>

<expression\_value>

<expression>**"R0"**</expression>

<value>**00**</value>

</expression\_value>

<expression\_value>

<expression>**"R1"**</expression>

<value>**01**</value>

</expression\_value>

<operand>**read\_from\_expression**</operand>

<operand>**read\_from\_values**</operand>

<operand\_type>**absolute**</operand\_type>

</regdir>

Слика 5.1.1. Опис регистарског директног начина адресирања

На слици 5.1.2. приказан је опис непосредног начина адресирања.

<immed>

<name>**immed**</name>

<file>**immed.cs**</file>

<result>**w**</result>

<expression>**"#"IDENTIFIER**</expression>

<expression>**"#" [SIGN] DEC\_NUMBER**</expression>

<expression>**"#" [SIGN] HEX\_NUMBER**</expression>

<operand>**read\_from\_expression**</operand>

<operand\_type>**absolute**</operand\_type>

</immed>

Слика 5.1.2. Опис непосредног начина адресирања

Процесор има два излазна порта. Порта *AD*, ширине 8 бита, који служи за постављање адресе или читање података, и порта *RD*, чија је намена сигнализирање да се жели читање из меморије. Када жeли читање података, процесор поставља порт *RD* на високу вредност и на порт *AD* поставља адресу локације са које се жели читати. Меморија одговара након 13 тактова система, постављајући тражену вредност на свој порт *ADDR*, који је повезан на порт *AD* процесора.

Поред меморије и процесора, реализоване су и две компоненте општег типа. Једна је генератор такта, која са одређеном периодом генерише сигнал такта. Друга је дисплеј, који прихвата осмобитну вредност и исписује је. Може се приметити да ове две компоненте немају употребну вредност, али су погодне за тестирање осетљивости апликације на системе са већим бројем компоненти или на компоненте са компликованијим исцртавањем.

Меморија и компоненте општег типа реализоване су на начин објашњен у поглављу 3.

Компоненте направљене као део овог пројекта тестиране су са три различита система, реализована по упутствима датим у поглављу 3. Сва три система тестирана су са истим кодом, приказаним на слици 5.1.3.

ld r0, #3

ld r0, #4

ld r1, #3

ld r1, #4

ld r2, #3

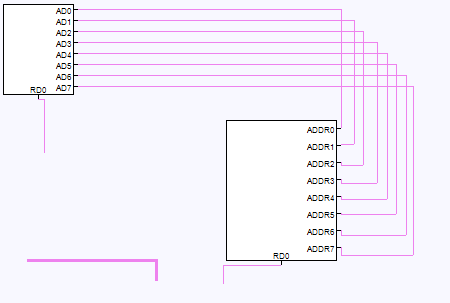
ld r2, #4

ld r3, #3

ld r3, #4

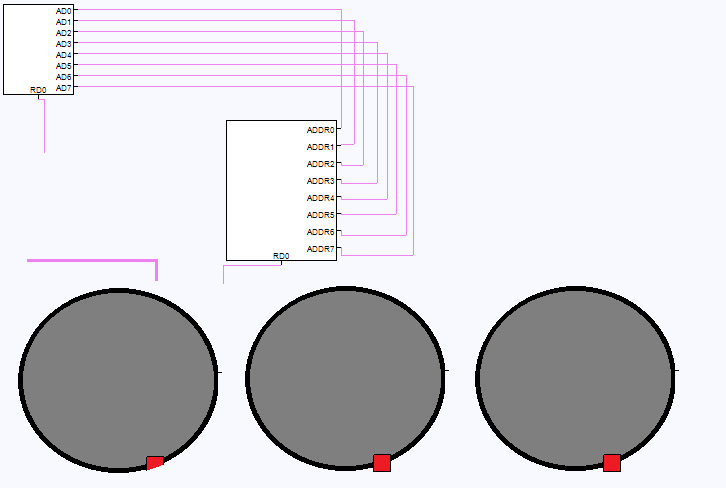
Слика 5.1.3. Код за тест системе 1, 2 и 3

На сликама 5.1.4, 5.1.5. и 5.1.6. приказана су ова три система.



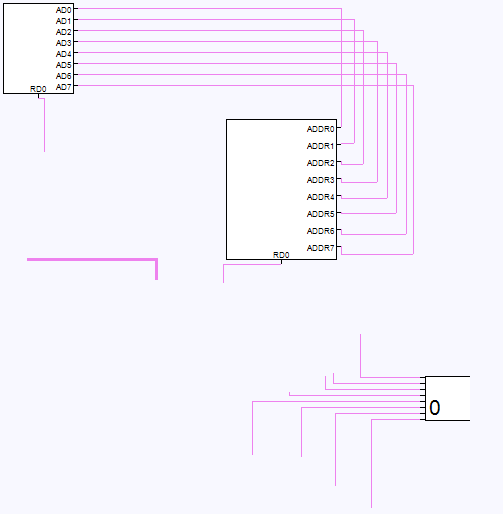
Слика 5.1.4. Тест систем број 1

Овај систем, који се састоји само од процесора и меморије. Портови *AD* и *ADDR* повезани су директним везама, док су портови *RD* повезани преко магистрале.



Слика 5.1.5. Тест систем број 2

Овај систем је сличан првом систему, али се разликује по томе што су му додате три генератора такта, приказана сивим кругом, који не раде никакав смислени посао, али троше ресурсе рачунара, а њихово исцртавање је захтевно.



Слика 5.1.6. Тест систем број 3

Тест ситем са слике изнад се, што се тиче смисленог посла, понаша потпупно исто као и прва два. Једина разлика је што су на сигнале који повезују магистралу и процесор додати и сигнали који их повезују са компонентом *Display*, која само приказује вредности које добија преко свог порта.

Други пројекат је доста компликованији од првог. Процесор је сличан процесору који је реализован у оквиру првог пројекта, али подржава шири спектар инструкција инструкција и има додатне портове *DATA*, ширине 8 бита, који се може користити за постављање података, и *INT*, преко кога процесор може да прима захтеве за прекид . Комуникација са меморијом се обавља на исти начин као у првом пројекту. Овај процесор подржава шири спектар инструкција:

* ***ld registar, operand*** - Ова инструкција функционише на потпуно исти начин као и у првом пројекту.
* ***jmp labela* -** Ова инструкција у вредност бројача корака уписује адресу на којој се налази инструкција означена лабелом *labela.* За адресирање се користи адресирање релативно у односу на бројач корака, односно, у бинарном коду инструкције садржана је вредност коју треба додати бројачу корака да би се отишло на адресу означену лабелом.
* ***out operand* -** Овом инструкцијом се вредност дата операндом *operand* поставља на порт *DATA*. За адресирање се користи непосредно адресирање, описано у оквиру описа првог система.
* ***itr labela* -** Извршавањем ове инструкције се поставља адреса на коју ће се отићи при обради прекида пристиглог преко порта *INT*. При извршавању инструкције, у листу глобалних променљивих додаје се променљива са именом *INTADDR,* чија се вредност поставља на адресу дату лабелом*labela.* Адреса се као локација на којој се налази инструкција означена лабелом.
* ***halt*** *-* Ова инструкција зауставља извршавање програма.

Поред процесора и меморије, реализоване су и две компоненте општег типа. Прва компонента је стандардни седмосегментни дисплеј. Код којим је дефинисано исцртавање ове компоненте је дат на слици 5.1.7.

var value **=** component**.**GetPort**(**"DSP"**).**Val**;**

component**.**Height **=** 52**;**

component**.**Width **=** 50**;**

Rectangle rectangle **=** **new** Rectangle**(**5**,** 0**,** 45**,** 48**);**

graphics**.**FillRectangle**(new** SolidBrush**(**Color**.**White**),** rectangle**);**

graphics**.**DrawRectangle**(**component**.**DefaultPen**,** rectangle**);**

component**.**GetPin**(**"DSP0"**).**Location **=** **new** Point**(**0**,** 1**);**

component**.**GetPin**(**"DSP1"**).**Location **=** **new** Point**(**0**,** 7**);**

component**.**GetPin**(**"DSP2"**).**Location **=** **new** Point**(**0**,** 13**);**

component**.**GetPin**(**"DSP3"**).**Location **=** **new** Point**(**0**,** 19**);**

component**.**GetPin**(**"DSP4"**).**Location **=** **new** Point**(**0**,** 25**);**

component**.**GetPin**(**"DSP5"**).**Location **=** **new** Point**(**0**,** 31**);**

component**.**GetPin**(**"DSP6"**).**Location **=** **new** Point**(**0**,** 37**);**

component**.**GetPin**(**"DSP7"**).**Location **=** **new** Point**(**0**,** 43**);**

string**[]** chars **=** **new** string**[**11**];**

chars**[**0**]** **=** " "**;**

chars**[**1**]** **=** **((**value **&** 0x01**)** **>** 0**)** **?** "\_" **:** " "**;**

chars**[**2**]** **=** " "**;**

chars**[**3**]** **=** "\n"**;**

chars**[**4**]** **=** **((**value **&** 0x20**)** **>** 0**)** **?** "|" **:** " "**;**

chars**[**5**]** **=** **((**value **&** 0x40**)** **>** 0**)** **?** "\_" **:** " "**;**

chars**[**6**]** **=** **((**value **&** 0x02**)** **>** 0**)** **?** "|" **:** " "**;**

chars**[**7**]** **=** "\n"**;**

chars**[**8**]** **=** **((**value **&** 0x10**)** **>** 0**)** **?** "|" **:** " "**;**

chars**[**9**]** **=** **((**value **&** 0x08**)** **>** 0**)** **?** "\_" **:** " "**;**

chars**[**10**]** **=** **((**value **&** 0x04**)** **>** 0**)** **?** "|" **:** " "**;**

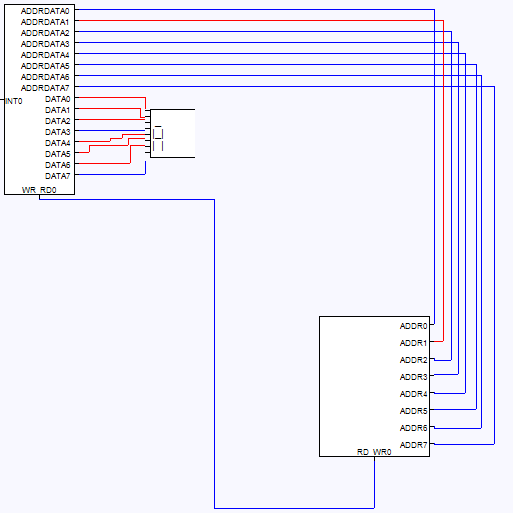
string s **=** string**.**Join**(**string**.**Empty**,** chars**);**

graphics**.**DrawString**(**s**,** **new** Font**(new** FontFamily**(**"Arial"**),** 8**),** Brushes**.**Black**,** **new** Point**(**5**,** 5**));**

Слика 5.1.7. Код којим се описује исцртавање седмосегментно дисплеја

Друга компонента је контролер прекида, која периодично шаље захтеве за прекид процесору. Процесор ослушкује вредност порта *INT*, и када се на њему појави позитивна вредност, чита вредност променљиве *INTADDR* из листе променљивих и уписује је у бројач корака.

Овај пројекат тестиран је са два система. Први систем је приказан на слици 5.1.8.



Слика 5.1.8. Тест систем број 4

Систем се састоји из процесора и меморије, с тим што је сада на процесор повезан и седмосегментни дисплеј. Код који се извршава на овом систему приказан је на слици 5.1.9.

out #77h

out #06h

out #7Fh

out #77h

out #06h

out #7Fh

out #77h

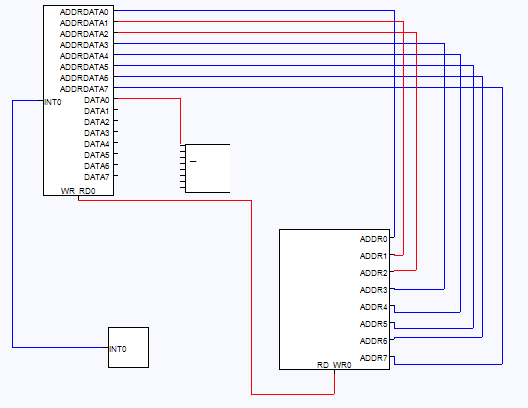
out #06h

out #7Fh

Слика 5.1.9. Код за тест систем број 4

Овај код тестира рад седмосегментног дисплеја, тако што исписује карактере {A, I, B} на њему, понављајући то три пута.

Последњи тест систем је уједно и најкомплекснији. Овај систем се састоји од процесора и меморије, повезаних на стандардни начин. На порт *INT* процесора повезан је контролер прекида, а први пин порта *DATA* повезан је на први пин седмосегментног дисплеја. Овај тест систем приказан је на слици 5.1.10.



Слика 5.1.10. Тест систем број 5

Овај систем тестиран је са кодом приказаним на слици 5.1.11.

itr lab

lab1: jmp lab1

lab: out #1

halt

Слика 5.1.11. Код за тест систем број 5

На почетку, постави се адреса прекидне рутине на адресу инструкције означену лабелом *lab.* Затим се извршава бесконачна петља, чекајући да стигне прекид. Када стигне прекид, поставља се вредност 1 на порт *DATA*, чиме се горњи сегмент седмосегментног дисплеја пали.

У табели 5.1.1. приказани су резултати добијени тестирањем.

Табела 5.1.1. Преглед резултата тестирања

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Редни број** | **Прецизност** | **Конзистентност** | **Време отварања пројекта [ms]** | **Време извршавања програма [ms]** |
| 1 | Да | Да | 6828 | 13360 |
| 2 | Да | Да | 7656 | 53310 |
| 3 | Да | Да | 9019 | 14759 |
| 4 | Да | Да | 7379 | 7503 |
| 5 | Да | Да | 8072 | 4692 |

На основу резултата приказаних у претходној табели, може се видети да се апликација у свим случајевима понаша прецизно и конзистентно. Време отварања пројекта благо варира, али се добијају резултати приближно истог реда величине. Разлике настају са повећањем броја компоненти чије је исцртавање дефинисано кроз графички интерфејс, јер је тада потребно више времена за динамичко учитавање слика, које траје нешто више од 2 секунде. Са друге стране, време извршавања програма јако варира и углавном зависи од броја компоненти, али и од количине компоненти чије је исцртавање компликовано. У свим случајевима осим у трећем, није било оваквих компоненти и програм се извршавао за приближно исто време, док је у другом случају било три компоненте чије је исцртавање компликовано, што је доста повећало време извршавања.

Пети тест систем највише одговара реалним системима. Показало се да овај систем има најбоље перформансе, што је и логично, с обзиром да је његов код најкраћи. Међутим, изузетно кратко време извршавања говори и о томе да, и поред компликованости целог система, перформансе нису трпеле, па се може рећи да дефинисање овако компликованог система не квари перформансе целог система.

1. Закључак

Циљ рада је био развој апликације која кориснику пружа могућност дефинисања асемблерског језика и преводиоца за тај језик у дефинисани машински језик, дефинисање архитектуре, понашања и изгледа сваке од рачунарских компоненти, дефинисање рачунарског система који се састоји од овако дефинисаних компоненти и веза између њих и на крају симулирање извршавања програма на овако дефинисаном систему, уз праћење рада стања целог система и сваке од компоненти и динамичко мењање овог стања ради евалуирања рада система.

Реализована апликација омогућује кориснику да у потпуности дефинише програмски језик који се извршава и машински језик у који се програмски језик преводи, уз ограничење да дефинисани језик мора бити асемблерски. Ово се имплицитно дешава дефинисањем компоненте типа процесора. Дефиниција програмског језика и машинског језика обавља се ван апликације, кроз *XML* датотеку и низ датотека са екстензијом *.cs* које се након тога учитавају. Перформансе учитавања су задовољавајуће, с обзиром да се не чека више од неколико секунди за креирање преводиоца на овај начин, иако је пријатност коришћења ове функционалности слабија. Након дефинисања програмског језика, може се дефинисати и произвољан број рачунарских компоненти, на начин сличан као што је то урађено за преводилац. Свака компонента идентификује се кроз *XML* датотеку, која описује архитектуру компоненте и *.cs* датотеке, које описују понашање компоненте. Имајући у виду да се на учитавање компоненте троши исто времена колико на учитавање језика, и овај део апликације показује добре перформансе. Предности овако имплементираног начина дефинисања архитектуре процесора и осталих рачунарских компоненти су перформансе рада апликације. Мана се огледа у томе што овај начин дефинисања архитектуре није толико пријатан за корисника као што би био уколико би постојао графички интерфејс. Дефинисање читавог система одвија се кроз графички интерфејс, додавањем и превлачењем компоненти и повезивањем пинова компоненти помоћу сигнала и магистрала. Овај начин се показао као добар у погледу перформанси, јер се додавање компоненти и веза између њих одиграва готово тренутно. Такође, једноставност коришћења је на високом нивоу, јер су обезбеђени начини кроз разне интуитивне пречице за лако додавање нових компоненти. Симулирање се обавља кроз графички интерфејс. Корисник може пратити стање свих компоненти система и вредности пинова. Може заустављати извршавање у сваком тренутку, извршавати код инструкцију по инструкцију или чак такт по такт, тако да се може донети закључак да су могућности велике. Мана овог дела система јесу перформансе, које драстично опадају када се користи велики број компоненти чије је исцртавање компликовано и захтева учитавање већег броја слика.

Као што је показано у претходном поглављу, реализовано је окружење које је потпуно конфигурабилно и чији је рад конзистентан и прецизан, али оставља могућности за нека проширења и унапређења. У претходном делу текста, показано је да је реализовано окружење које задовољава циљеве постављене на почетку рада, али и уводи нека ограничења. Окружење се за сада може покретати само под оперативним системом *Microsoft Windows* и да би било подржано и од стране других оперативних система, потребно је направити измене које нису велике, али ипак захтевају одређено време да би биле реализоване на прави начин. Ово подразумева адаптацију кода, који користи *.NET* радни оквир, али и мењање неких битних делова који се тичу покретања екстерних апликација, а који су специфични за сваки оперативни систем. Овај део апликације је рализован модуларно, па се може параметризовати, тако да се екстерне апликације увек отварају на начин погодан за оперативни систем на коме се цела апликација покреће. Као што се могло видети, тренутно је могуће додати само један процесор у систем и извршавати програме на њему. Ово доноси доста ограничења, с обзиром да данас велики број система има више процесора. Ово је једна од могућности за проширење које није тешко реализовати, с обзиром да су промене неопходне само у графичком интерфејсу, како би се омогућило више едитора кода. Требало би реализовати едитор који подржава картице, или у коме се из падајућег менија бира за који процесор се пише код. На тај начин, сваки процесор би имао свој машински код који извршава независно од другог процесора. Овде остаје отворено питање како реализовати дебаговање апликације, али се и то може решити аналогно комерцијалним преводиоцима. Дефинисање архитектура и понашања процесора и других компоненти се углавном одвија кроз екстерне апликације, што врло често није пријатно за кориснике. У овом делу апликацију је могуће доста унапређивати. У ту сврху, намеће се као логичан извор креирање графичког интерфејса посебно за сврху дефинисања архитектуре рачунарских компоненти. Ова унапређења су једноставна, али ипак не тако мала, па за њихову имплементацију треба више времена. У погледу перформанси, реализована апликација се показала прилично добро, осим уколико се дода већи број компоненти чије је исцртавање компликовано. У овом делу постоје могућности за проширења, али њих није тако лако реализовати. Ове промене би уједно биле и најскупље, али би допринеле целокупном утиску о апликацији.

Коначно, може се извести закључак да је реализовано решење у потпуности задовољило циљеве постављене на почетку, посебно оне који се тичу конфигурабилности. Реализована апликација представља озбиљан симулатор, који би, уколико се спроведу нека од могућих проширења и унапређења, могао лако да доживи комерцијалну употребу.

Литература

1. Бојан Јелача, *Преводилац и симулатор за више архитектура рачунара*, дипломски рад, септембар 2014.
2. *Microsoft Developer Network* (<https://msdn.microsoft.com>)
3. *Stack Overflow* (<http://stackoverflow.com>)
4. *Labcenter Electronics* (<http://www.labcenter.com>)
5. *VisualSim Architect* (<http://mirabilisdesign.com/new/visualsim/>)
6. *CPU OS Simulator* (<http://www.teach-sim.com/>)
7. *Reflection Introduction* (<http://csharp.net-tutorials.com/reflection/introduction/>)

Списак слика

[Слика 2.1.1. Дефинисање шеме система у алату *Proteus Design Suite* 3](#_Toc460377722)

[Слика 2.2.1. Основни екран окружења *VisualSim Architect* 4](#_Toc460377723)

[Слика 2.3.1. Изглед екрана окружења *Computer OS Simulator* 6](#_Toc460377724)

[Слика 3.3.1. Графички интерфеjс за дефинисање изгледа компоненти 14](#_Toc460377725)

[Слика 3.3.2. Пример компоненте чији је изглед дефинисан кроз графички интерфејс 15](#_Toc460377726)

[Слика 3.3.3. Пример дефинисања изгледа компоненте кроз код 16](#_Toc460377727)

[Слика 3.4.1. Прозор за пројектовање система 16](#_Toc460377728)

[Слика 3.4.2. Пример система 17](#_Toc460377729)

[Слика 4.1.1. Дијаграм класа за имплементацију графичког дела дефинисања изгледа компоненти 21](#_Toc460377730)

[Слика 4.1.2. Проналажење граница компоненте приликом исцртавања 23](#_Toc460377731)

[Слика 4.1.3. Дијаграм класа за реализацију графичког дела пројектовања система и симулирања рада система 26](#_Toc460377732)

[Слика 4.2.1. Дијаграм класа од значаја за дефинисање архитектуре и рада компоненти 30](#_Toc460377733)

[Слика 4.2.2. Дијаграм класа значајних за дефинисање изгледа компоненти 33](#_Toc460377734)

[Слика 4.2.3. Дијаграм класа значајних за пројектовање система 35](#_Toc460377735)

[Слика 4.2.4. Спајање сигнала по имену 37](#_Toc460377736)

[Слика 4.2.5. Дијаграм класа од значаја за симулирање рада система 39](#_Toc460377737)

[Слика 5.1.1. Опис регистарског директног начина адресирања 42](#_Toc460377738)

[Слика 5.1.2. Опис непосредног начина адресирања 42](#_Toc460377739)

[Слика 5.1.3. Код за тест системе 1, 2 и 3 43](#_Toc460377740)

[Слика 5.1.4. Тест систем број 1 43](#_Toc460377741)

[Слика 5.1.5. Тест систем број 2 44](#_Toc460377742)

[Слика 5.1.6. Тест систем број 3 44](#_Toc460377743)

[Слика 5.1.7. Код којим се описује исцртавање седмосегментно дисплеја 46](#_Toc460377744)

[Слика 5.1.8. Тест систем број 4 46](#_Toc460377745)

[Слика 5.1.9. Код за тест систем број 4 47](#_Toc460377746)

[Слика 5.1.10. Тест систем број 5 47](#_Toc460377747)

[Слика 5.1.11. Код за тест систем број 5 47](#_Toc460377748)

Списак табела

[Табела 2.4.1. Пресек постојећих решења 7](#_Toc460377249)

[Табела 4.1.1. Преглед најважнијих догађаја класе *Designer* 22](#_Toc460377250)

[Табела 4.1.2. Преглед најважнијих догађаја класе *DragAndDropPanel* 24](#_Toc460377251)

[Табела 4.1.3. Преглед најважнијих догађаја класе *Clipboard* 27](#_Toc460377252)

[Табела 5.1.1. Преглед резултата тестирања 48](#_Toc460377253)