

УНИВЕРЗИТЕТ У БЕОГРАДУ
ЕЛЕКТРОТЕХНИЧКИ ФАКУЛТЕТ



РЕАЛИЗАЦИЈА ИНТЕГРИСАНОГ РАЗВОЈНОГ ОКРУЖЕЊА ЗА ПИКО КОМПЈУТЕР НА ВЕБУ

Завршни рад основних академских студија

Ментор:
проф. др Јелица Протић

Кандидат:
Немања Миљковић 490/2010

Београд, Септембар 2018.

Садржај

Садржај	<i>i</i>
1. Увод	1
2. Опис архитектуре picoComputer-a	2
2.1. Хардверска организација	2
2.1.1. Меморија.....	2
2.1.2. Уређај за обраду улаза/излаза.....	3
2.1.3. Процесор	3
2.2. Синтакса	3
2.2.1. Симболи.....	4
2.2.2. Почетак програма.....	4
2.2.3. Инструкције	4
2.3. Сет инструкција	5
2.3.1. Инструкције копирања – Move.....	6
2.3.2. Аритметичке инструкције.....	7
2.3.3. Инструкције условног скока	9
2.3.4. Улазно/излазне инструкције	10
2.3.5. Инструкције за скок и повратак из потпрограма	11
2.3.6. Инструкције за заустављање извршавања.....	12
3. Функционални захтеви	13
3.1. Писање кода и парсирање	13
3.2. Симулација	13
4. Реализација парсирања, анализе, превођења и извршавања	14
4.1. Парсирање програма	14
4.1.1. Парсирање дефиниције симбола.....	15
4.1.2. Парсирање адресе почетка програма.....	16
4.1.3. Парсирање програмског кода	16
4.2. Анализа програма	16
4.2.1. Анализа симбола и почетка програма	17
4.2.2. Анализа инструкција	18
4.3. Превођење програма	19
4.4. Извршавање и дебаговање	21
4.5. Тестирање	22
4.5.1. Тестирање анализе.....	22
4.5.2. Тестирање генерисања бајткода	23
4.5.3. Тестирање извршавања	24
5. Реализација корисничког интерфејса	25
5.1. Vue.js	25

5.2. Повезивање парсера са корисничким интерфејсом	25
6. Закључак	28
<i>Литература.....</i>	<i>29</i>
<i>Списак скраћеница.....</i>	<i>30</i>
<i>Списак слика</i>	<i>31</i>
<i>Списак кода.....</i>	<i>32</i>
<i>Списак табела.....</i>	<i>33</i>
A. Упутство за подешавање развоја на окружењу	34
A.1. Развој pico-asm библиотеке	34
A.2. Развој pico-sim окружења	34
B. Упутство за коришћење	35
B.1. Писање програма.....	35
B.2. Извршавање симулације.....	37

1. Увод

picoComputer је архитектура дизајнирана од стране др Јозо Дујмовића 1989. године са циљем да олакша учење асемблерског језика [1]. Уз дизајн архитектуре, др Дујмовић је направио DOS апликацију "pC Assembler and Simulator" краће названу pCAS.

picoComputer архитектура је остала релевантна током година. Модерне архитектуре подржавају већи број инструкција, имају више компоненти и раде брже али деле основе и принципе дизајна.

За лакше учење picoComputer асемблера направљено је десктоп развојно окружење "MessyLab" [2] налик на модерна графичка окружења. "MessyLab" олакшава писање програма и дебаговање - подржава синтаксну анализу, извршавање инструкција корак по корак, праћење променљивих, заустављање симулације на неки део кода итд.

Реализацијом окружења за развој picoComputer програма на веб-у омогућава се програмирање и без преузимања десктоп апликације. Могуће га је покренути на мобилном телефону или таблету. Није потребно преузимати посебну апликацију већ довољно посетити веб страницу симулатора. Веб окружење ради на свим оперативним системима што није ограничавајућа околност као код апликације "MessyLab".

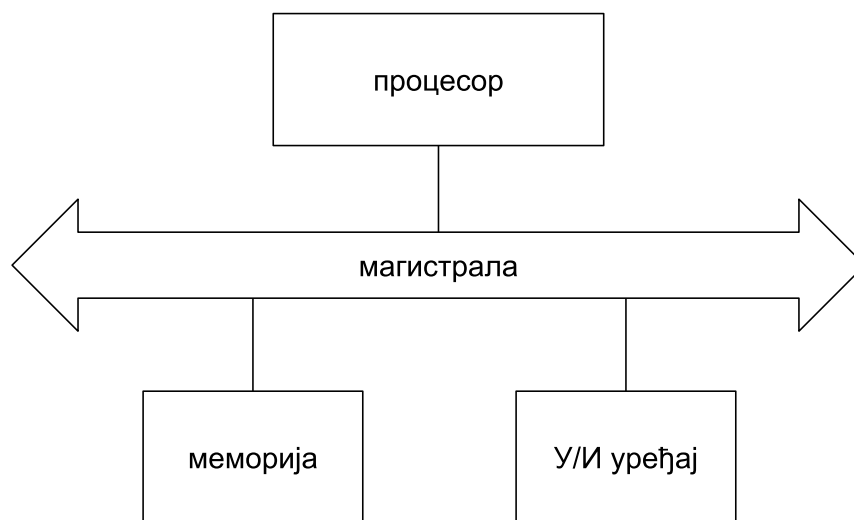
У наставку рада су описани:

- Поглавље 2 описује архитектуру picoComputer-а. Описан је сет инструкција које picoComputer подржава и варијацијама
- Поглавље 3 садржи функционалне захтеве picoComputer развојног окружења на веб-у
- Поглавље 4 описује реализацију парсирања, анализе и извршавања програма и процес верификације исправности
- Поглавље 5 описује реализацију корисничког интерфејса
- Поглавље 6 дискутује решење и могућа унапређења
- Додатак А садржи упутство за подешавање развоја на окружењу

2. ОПИС АРХИТЕКТУРЕ PICO COMPUTER-A

2.1. Хардверска организација

picoComputer је једноставна архитектура која се састоји од процесора, меморије и уређаја за обраду улазно/излазних података (Слика 2.1.1).



Слика 2.1.1 Хардверска организација picoComputer архитектуре

2.1.1. Меморија

Меморија picoComputer-а садржи највише 65536 16-то битних речи. Меморијске адресе су такође 16-то битне. Меморија је подељена у две логичне целине (Слика 2.1.2).



Слика 2.1.2 Подела меморије picoComputer-а

Фиксна зона меморије FDA (*Fixed Data Area*) се састоји од првих 8 меморијских локација и оне имају улогу регистара опште намене који се могу наћи у другим архитектурама.

Остатак меморије се може слободно користити али је потребно водити рачуна о показивачу на стек SP (*Stack Pointer*) који почиње од последње меморијске локације и расте на доле. Не треба користити локације које могу бити пребрисане стеком.

2.1.2. Уређај за обраду улаза/излаза

Улазно/излазни уређај се понаша као тастатура и екран. Симулатор може да прими бројеве и екран може приказати неку од вредности из меморије picoComputer-а.

Позиви за улаз/излаз су блокирајући и нема паралелног извршавања.

picoComputer не подржава реаговање на прекиде. Једини прекид који picoComputer генерише је прекид на неодговарајући улаз. Прекиди обустављају извршавање програма.

2.1.3. Процесор

Процесор има бројач наредби PC (*Program Counter*) и регистар показивача на стек SP (*Stack Pointer*). Они се не могу користити као операнди у инструкцији.

Вредност регистра PC се инкрементира након читавања инструкције. Инструкције условног скока и инструкције скока и повратка из потпрограма могу променити регистар PC приликом извршавања. Почетну вредност регистра PC је могуће дефинисати директивом *org*.

Вредност регистра SP се мења извршавањем инструкција за скок или повратак из потпрограма. Стеку није могуће приступити кроз остале инструкције, али је могуће приступити меморијским локацијама које стек користи што се не препоручује.

Постоје регистри које процесор користи али их је немогуће користити из писаних програма и нису битни за симулацију извршавања.

2.2. Синтакса

Структура picoComputer програма се састоји из:

- 1) дефиниција симбола – није обавезно
- 2) адресе почетка програма – мора бити дефинисана
- 3) програмског кода – мора садржати барем једну инструкцију

Код 2.2.1 приказује пример програма који учита два броја, сабере их и испише резултат.

```

; Symbols
a = 1
b = 2
c = 3

; Origin
org 8

; Instructions
start: in a          ; input a
       in b          ; input b
       add c, a, b    ; c = a + b
       out c          ; output c

       in a          ; input a
       beq a, 0, end   ; if a=0 goto end
       beq a, a, start ; goto start
end:    stop          ; stop execution

```

Код 2.2.1 Програм који сабира два броја

2.2.1. Символи

Символи представљају симболичке ознаке меморијских локација и омогућавају читљивије писање кода. Могуће је дефинисати више симбола са истом вредношћу, али није могуће дефинисати више симбола са истим именом.

Символи се користе само током превођења и нису присутни у генерисаном коду већ се адресе уписују директно као операнди инструкција.

Лабеле су врста симбола који приликом превођења имају вредност меморијске локације пратеће инструкције. Могу се користити у инструкцијама условног скока и скока у потпрограме.

2.2.2. Почетак програма

Почетак програма се дефинише директивом *org*. Адреса почетка мора бити меморијска локација у опсегу [8..65535], тј. не сме показивати на меморијске локације резервисане за FDA и мора бити у оквиру расположиве меморије.

2.2.3. Инструкције

Инструкције у picoComputer архитектури су дефинисане симболичким именом и листом операнада. Постоје четири типа операнада дефинисаних у picoComputer архитектури (Табела 2.2.1).

Табела 2.2.1 Типови операнда

формат	пример	тип адресирања	опис
симбол	a	Директно	Адреса меморијске локације вредности операнда је вредност симбола.
(симбол)	(a)	Индијектно	Адреса меморијске локације вредности операнда је у меморији. Адреса меморијске локације која садржи меморијску локацију операнда је вредност симбола.
#симбол	#a	Непосредно	Вредност операнда је вредност симбола.
константа	5	Непосредно	Вредност операнда је константа.

2.3. Сет инструкција

рiсoComputer дефинише 6 различитих типова инструкција:

- 1) Инструкције копирања
- 2) Аритметичке инструкције
- 3) Инструкције условног скока
- 4) Улазно/излазне инструкције
- 5) Инструкције за скок и повратак из потпрограма
- 6) Инструкције за заустављање извршавања

рiсoComputer инструкције имају до највише 3 операнда. Инструкција има једну или две речи у зависности од тога да ли је један операнд константа који се чува у другој речи. Табела 2.3.1 приказује формат инструкција.

Табела 2.3.1 Формат инструкција

Код операције				i1	a1			i2	a2			i3	a3		
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Битови 15 до 12 означавају код операције док се остали користе за кодирање начина адресирања и вредности операнда. Зато што се 4 бита користе за код операције, максимални број подржаних инструкција може бити 16.

Сваки операнд се кодира са 4 бита. Бит *i* означава тип адресирања, а бити *a* означавају адресу у FDA. Приступ меморијским локацијама ван FDA се постиже коришћењем индијектног адресирања.

Неке инструкције одступају од дефинисаног формата и могу користити делове резервисане за операнде у друге сврхе.

2.3.1. Инструкције копирања – Move

Инструкције копирања се користе за копирање вредности из једне меморијске локације у другу. Табела 2.3.2 приказује код операција.

Табела 2.3.2 Кодови операција инструкција копирања

	Код операције	Симболичко име
Копирај	0000	MOV

i) Тип 1

Ако су $i3$ и $a3$ једнаки 0, вредност другог операнда у копира се у операнд x . Код 2.3.1 приказује пример коришћења инструкције. Табела 2.3.3 приказује формат инструкција.

MOV x, y

Код 2.3.1 Коришћење инструкције копирања типа 1

Табела 2.3.3 Формат инструкције копирања типа 1

Код операције				$i1$	$a1$			$i2$	$a2$			$i3$	$a3$		
0	0	0	0	x	x	x	x	y	y	y	y	0	0	0	0

ii) Тип 2

Ако је $i3$ једнак 1 и $a3$ једнак 0, вредност константе c у другој речи копира се у операнд x . Код 2.3.2 приказује пример коришћења инструкције. Табела 2.3.4 приказује формат инструкција.

MOV $x, 5$
MOV (x), $\#a$

Код 2.3.2 Коришћење инструкције копирања типа 2

Табела 2.3.4 Формат инструкције копирања типа 2

Код операције				$i1$	$a1$			$i2$	$a2$			$i3$	$a3$		
0	0	0	0	x	x	x	x	?	?	?	?	1	0	0	0
c	c	c	c	c	c	c	c	c	c	c	c	c	c	c	c

iii) Тип 3

Ако је $i3$ једнак 1 и $a3$ једнак 7 (бинарно 111), вредност низа са почетном адресом операнда у копира се у низ чија је адреса дата операндом x . Дужина низа n се налази у другој речи инструкције. Код 2.3.3 приказује пример коришћења инструкције. Табела 2.3.5 приказује формат инструкција.

```
MOV x, y, 5
MOV (x), (y), #a
```

Код 2.3.3 Коришћење инструкције копирања типа 3

Табела 2.3.5 Формат инструкције копирања типа 3

Код операције				i1	a1			i2	a2			i3	a3		
0	0	0	0	x	x	x	x	y	y	y	y	1	1	1	1
n	n	n	n	n	n	n	n	n	n	n	n	n	n	n	n

iv) *Tun 4*

Ако је *i3* једнак 0 и *a3* није једнак 0, вредност низа са почетном адресом операнда у копира се у низ чија је адреса дата операндом *x*. Код 2.3.4 приказује пример коришћења инструкције. Табела 2.3.6 приказује формат инструкција.

```
MOV x, y, n
MOV (x), (y), n
```

Код 2.3.4 Коришћење инструкције копирања типа 4

Табела 2.3.6 Формат инструкције копирања типа 4

Код операције				i1	a1			i2	a2			i3	a3		
0	0	0	0	x	x	x	x	y	y	y	y	0	n	n	n

2.3.2. Аритметичке инструкције

Аритметичке инструкције се користе за основно рачунање. picoComputer подржава сабирање, одузимање, множење и дељење. Свака рачунска операција има два кода операције али исто симболичко име. Табела 2.3.7 приказује кодове операција инструкција које се деле на:

- 1) Аритметичке операције без константе
- 2) Аритметичке операције са константом

Табела 2.3.7 Кодови операција аритметичких инструкција

	Код операције	Симболичко име	
Сабирање	0001	1001	ADD
Одузимање	0010	1010	SUB
Множење	0011	1011	MUL
Дељење	0100	1100	DIV
	Без константе	Са константом	

i) Без константе

Табела 2.3.8 приказује формат аритметичких инструкција ако ниједан операнд није константа. Код 2.3.5 приказује пример коришћења аритметичких инструкција са различитим врстама операнда.

```
ADD x, y, z
SUB (x), y, z
MUL x, (y), z
DIV (x), (y), (z)
```

Код 2.3.5 Коришћење аритметичких инструкција без константе

Табела 2.3.8 Формат аритметичких инструкција без константе

Код операције				i1	a1			i2	a2			i3	a3		
0	оп	оп	оп	x	x	x	x	y	y	y	y	z	z	z	z

ii) Са константом

Аритметичке операције дозвољавају коришћење једног операнда као константу – константа може бити други или трећи операнд. Први операнд не сме бити константа јер мора бити меморијска локација на коју се уписује резултат.

Операнд који је константа се означава тако што *i* и *a* буду 0. Константа *c* се чува у другој речи инструкције.

Табела 2.3.9 приказује формат аритметичких инструкција када је други операнд константа. Код 2.3.6 приказује пример коришћења инструкција.

```
ADD x, 5, z
SUB x, #c, z
```

Код 2.3.6 Коришћење аритметичких инструкција са константом – други операнд

Табела 2.3.9 Аритметичка инструкција са константом – трећи операнд

Код операције				i1	a1			i2	a2			i3	a3		
1	оп	оп	оп	х	х	х	х	0	0	0	0	z	z	z	z
с	с	с	с	с	с	с	с	с	с	с	с	с	с	с	с

Табела 2.3.10 приказује формат инструкција када је трећи операнд константа. Код 2.3.7 приказује пример коришћења инструкција. Из разлога што се 0 користи за означавање константог операнда, адресирање меморијске локације 0 у FDA није дозвољено.

ADD x, y 5
SUB x, y, #с

Код 2.3.7 Коришћење аритметичких инструкција са константом – трећи операнд

Табела 2.3.10 Аритметичка инструкција са константом – трећи операнд

Код операције				i1	a1			i2	a2			i3	a3		
1	оп	оп	оп	х	х	х	х	у	у	у	у	0	0	0	0
с	с	с	с	с	с	с	с	с	с	с	с	с	с	с	с

2.3.3. Инструкције условног скока

Инструкције условног скока се користе за контролу извршавања програма. Оне мењају регистар PC уколико се услов задат инструкцијом испуни.

pcsoComputer има две инструкције условног скока (Табела 2.3.11).

Табела 2.3.11 Кодови операција инструкција условног скока

	Код операције	Симболичко име
Скочи ако су једнаки	0101	BEQ
Скочи уколико је већи	0110	BGT

Инструкције имају 2 формата који се разликују у начину кодирања адресе за скок.

i) Тип 1 – адреса је константа

Табела 2.3.12 приказује формат инструкција када се константа адреса налази у другој речи инструкције. Адреса је тада задата лабелом и означава меморијску локацију инструкције на коју треба скочити. Код 2.3.8 приказује пример коришћења инструкција са лабелама *is_eq* и *is_gt*.

```
BEQ x, y, is_eq
BGT x, y, is_gt
```

Код 2.3.8 Коришћење инструкција условног скока са лабелом

Табела 2.3.12 Формат инструкција условног скока са константним адресом – скок на лабелу

Код операције				i1	a1			i2	a2			i3	a3		
оп	оп	оп	оп	x	x	x	x	y	y	y	y	1	0	0	0
с	с	с	с	с	с	с	с	с	с	с	с	с	с	с	с

ii) Тип 2 – адреса је меморијска локација

Када је адреса скока меморијска локација тада се трећим операндом кодира адреса меморијске локације (Табела 2.3.13). У питању је индиректно адресирање али је *i3* једнако 0, на шта треба обратити пажњу. Код 2.3.9 приказује пример коришћења инструкција.

```
BEQ x, y, (z)
BGT x, y, (z)
```

Код 2.3.9 Коришћење инструкција са скоком на адресу која се налази у меморији

Табела 2.3.13 Формат инструкција са скоком на адресу која се налази у меморији

Код операције				i1	a1			i2	a2			i3	a3		
оп	оп	оп	оп	x	x	x	x	y	y	y	y	0	z	z	z

Први и други операнд се кодирају по стандардним правилима али постоји посебан случај за поређење са нулом. Ако су *i* и *a* једнаки нули, сматра се да је тај операнд константа нула, самим тим није могуће адресирати меморијску локацију 0.

2.3.4. Улазно/излазне инструкције

Инструкције примају 2 операнда. Први операнд је дестинација или извор податка у зависности да ли је инструкција улаза или излаза. Други операнд је дужина низа у који се пише или чита. Приликом писања програма није неопходно дати дужину и у том случају преводилац користи подразумевану вредност (1). Табела 2.3.14 приказује кодове операција.

Табела 2.3.14 Кодови операција У/И инструкција

	Код операције	Симболичко име
Улаз	0111	IN
Излаз	1000	OUT

Ако је *i2* једнак јединици а *a2* једнако нули, тада се користи трећи операнд за дужину низа. Табела 2.3.15 приказује формат инструкција.

Табела 2.3.15 Формат У/И инструкција када је дужина низа меморијска локација

Код операције				i1	a1				i2	a2				i3	a3			
оп	оп	оп	оп	х	х	х	х	х	0	0	0	0	0	п	п	п	п	п

Ако је $i2$ једнак нули тада се користе $a2$, $i3$ и $a3$ за чување константе која представља дужину низа. Табела 2.3.16 приказује формат инструкција. За разлику од других инструкција које чувају константу као другу реч инструкције овде се чувају у нижих 7 бита. Због тога константа c не може бити већа од 127.

Табела 2.3.16 Формат У/И инструкција када је дужина низа константа

Код операције				i1	a1				i2	a2				i3	a3			
оп	оп	оп	оп	х	х	х	х	х	1	с	с	с	с	с	с	с	с	с

2.3.5. Инструкције за скок и повратак из потпрограма

Инструкција JCP се користи за скок у потпрограм, и за разлику од инструкција условног скока чува повратну адресу на стеку. Инструкцијом RTS је после могуће наставити извршавање од инструкције након позива потпрограма. Табела 2.3.17 приказује кодове операција.

Табела 2.3.17 Кодови операција инструкција скока и повратка из потпрограма

	Код операције	Симболичко име
Скок у потпрограм	1101	JSR
Повратак из потпрограма	1110	RTS

Табела 2.3.18 приказује формат инструкција за скок у потпрограм. Адреса потпрограма се чува у другој речи инструкције. Код 2.3.10 показује пример скока на лабелу *sub*.

JSR sub

Код 2.3.10 Коришћење скока на потпрограм дефинисаног лабелом *sub*

Табела 2.3.18 Формат инструкција за скок на потпрограм када је адреса скока константа

Код операције				i1	a1				i2	a2				i3	a3			
1	1	0	1	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
с	с	с	с	с	с	с	с	с	с	с	с	с	с	с	с	с	с	с

Табела 2.3.19 приказује формат инструкције за повратак из потпрограма. Инструкција не прима аргументе већ чита повратну адресу са стека. Код 2.3.11 приказује пример коришћења инструкције.

RTS

Код 2.3.11 Коришћење инструкције за повратак из потпрограма

Табела 2.3.19 Формат инструкције за повратак из потпрограма

Код операције				i1	a1				i2	a2				i3	a3			
1	1	1	0	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?

2.3.6. Инструкције за заустављање извршавања

Инструкција STOP се користи за заустављање извршавања програма. Инструкција може примити до 3 операнда које ће послати на излаз. Табела 2.3.20 описује формат инструкција. Недефинисани аргументи се кодирају са нулом и зато постоји ограничење да није могуће исписати вредност меморијске локације 0. Код 2.3.12 приказује пример коришћења инструкције за заустављање програма са различитим врстама операнда.

STOP
 STOP x
 STOP x, (y)
 STOP (x), y, (z)

Код 2.3.12 Коришћење инструкције за заустављање извршавања програма

Табела 2.3.20 Формат инструкције за заустављање извршавања програма

Код операције				i1	a1				i2	a2				i3	a3			
1	1	1	0	x	x	x	x	x	y	y	y	y	z	z	z	z	z	z

3. ФУНКЦИОНАЛНИ ЗАХТЕВИ

Развојно окружење мора покрити све кораке од уношења кода до његовог извршавања и омогућити лак увид кориснику у проблеме настале приликом писања програма како би се лако отклонили.

3.1. Писање кода и парсирање

Потребно је омогућити писање кода у едитору налик на модерна графичка окружења. Кључне речи би требало назначити другој бојом. Уколико код није синтаксички исправан треба то назначити у едитору на месту где се налази грешка повлачењем спорног дела кода и бојењем позадине. Исто важи за семантички неисправне програме. Поред приказа у едитору, треба приказати списак свих грешака на једном месту у листи испод едитора.

Са леве стране кода треба приказати бројеве линија кода и оставити места за приказ зауставних тачки. Кликом на број линије би требало назначити да је на тој линији постављена зауставна тачка. На празне линије не треба бити могуће постављати зауставне тачке. Уколико се обрише линија на којој је била постављена зауставна тачка она треба нестати. Постављање зауставних тачки на линије кода пре директиве *org* нема ефекта.

3.2. Симулација

Када је написан код синтаксички и семантички исправан треба омогућити покретање симулације, са и без подршке за дебаговање.

Покретање без подршке за дебаговање треба да изврши симулацију без заустављања на постављеним зауставним тачкама. Тиме корисник не мора да склања постављене зауставне тачке ако само жели да изврши цео програм.

Покретање са подршком за дебаговање треба да започне извршавање симулације и да се заустави пре извршавања инструкције на којој је постављена зауставна тачка. Инструкције на којима се извршавање заустави у том случају требају бити јасно обележене.

У случају заустављања симулације, треба постојати могућност да се симулација настави или у потпуности прекине.

Током извршавања је потребно пратити вредности меморијских локација у FDA. Симулатор може да препозна дефинисане симболе и да приказује вредности меморијских локација на које симболи означавају, уколико се оне налазе у FDA.

Симулација се може зауставити и у случају да је захтеван улаз од стране корисника. Не треба дозволити наставак извршавања док корисник не унесе податак али је дозвољено прекидање симулације.

Све У/И операције морају да се прикажу у листи која ће садржати адресу у/из које је писано/читано и вредност.

У случају грешака током извршавања програма, треба их исписати у истој листи где су записане УИ операције. Пример грешке која је могућа у току извршавања програма и није је могуће спречити је дељење са нулом.

4. РЕАЛИЗАЦИЈА ПАРСИРАЊА, АНАЛИЗЕ, ПРЕВОЂЕЊА И ИЗВРШАВАЊА

Пакет `pico-asm` је написан као JavaScript библиотека која у себи садржи модуле за парсирање, анализу, превођење и извршавање `picoComputer` програма.

`pico-asm` користи ANTLR (ANother Tool for Language Recognition) [3] парсер генератор. Предност коришћења ANTLR-а је једноставност коришћења. ANTLR сам генерише лексер за токенизацију програма, прави модел синтаксног стабла и даје лаку могућност обиласка стабла уз помоћ Visitor шаблона [4]. Могуће је искористити граматику за генерисање парсера за више различитих програмских језика тако да је у будућности могуће искористити написану граматику у друге сврхе ван JavaScript-а [5].

4.1. Парсирање програма

Структура `picoComputer` програма се састоји из:

- 1) дефиниција симбола – није обавезно
- 2) адресе почетка програма – мора бити дефинисана
- 3) програмског кода – мора садржати барем једну инструкцију

Код 4.1.1 приказује опис граматике структуре програма.

```
grammar Pico;

program
: EOL* symbols origin instructions
;

symbols
: symbolDeclLine*
;

origin
: ORG constant EOL
;

instructions
: line+
;
```

Код 4.1.1 Граматика структуре програма

Генерисање парсера захтева покретање ANTLR генератора над граматиком и потребно је покренути команду за генерисање приликом промене граматике (Код 4.1.2).

Приликом генерисања добијају се класе *PicoLexer*, *PicoParser*, *PicoListener* и *PicoVisitor*. Додатне контекстне класе постоје за правила која парсер користи и ANTLR генерише имена на основу самог правила: *SymbolsContext*, *OriginContext*, *InstructionsContext* итд.

PicoVisitor генерисан од стране ANTLR-а је класа из које се могу извести посетиоци стабла који могу форматирати код, извршити анализу и генерисати бајткод (Код 4.1.3).

```
#!/usr/bin/env bash

VERSION="4.7.1"
FILENAME="antlr-$VERSION-complete.jar"

if [ ! -f "$FILENAME" ]; then
    curl "http://www.antlr.org/download/$FILENAME" --output
"$FILENAME"
fi

java -jar "$FILENAME" -Dlanguage=JavaScript -visitor Pico.g4
```

Код 4.1.2 Генерисање парсера (src/parser/compile.sh)

```
// Visit a parse tree produced by PicoParser#symbols.
PicoVisitor.prototype.visitSymbols = function(ctx) {
    return this.visitChildren(ctx);
};
// Visit a parse tree produced by PicoParser#origin.
PicoVisitor.prototype.visitOrigin = function(ctx) {
    return this.visitChildren(ctx);
};
```

Код 4.1.3 Пример генерисаних метода

4.1.1. Парсирање дефиниције симбола

Симболи се дефинишу именом и вредношћу. Вредност може бити било који 16-то битни број са знаком (Код 4.1.4).

```
a = 1
N = -100
```

Код 4.1.4 Исправно дефинисани симболи

Правило symbolDecl дефинише правила парсирања симбола (Код 4.1.5).

```
symbolDecl
: identifier '=' constant
;

identifier
: IDENTIFIER
;

constant
: SIGN? NUMBER
;

IDENTIFIER
: [a-zA-Z][a-zA-Z0-9_]*
;

NUMBER
: [0-9]+
;

SIGN
: '+' | '-'
;
```

Код 4.1.5 Граматика дефинисања симбола

Правило *identifier* је направљено са циљем јединственог начина обраде имена симбола и лабела. Иако је исправно написати *IDENTIFIER* унутар *symbolDecl*, писањем наизглед непотребних правила говори генератору да направи методе за обилазак идентификатора. ANTLR неће направити методе за обилазак терминалних чворова као што су *IDENTIFIER*, *NUMBER* и *SIGN*.

4.1.2. Парсирање адресе почетка програма

Дефинисање адресе почетка програма је покривено правилом *origin* (Код 4.1.6). Иако почетак програма мора да се налази у одређеном распону, исправност вредности се не проверава приликом парсирања већ приликом семантичке анализе.

```
origin
: ORG constant EOL
;
```

Код 4.1.6 Правило за парсирање адресе почетка програма

4.1.3. Парсирање програмског кода

Парсирање програмског кода се своди на парсирање листе инструкција раздвојених терминатором са опционом лабелом испред (Код 4.1.7).

```
line
: label? (instruction?) EOL
;

instruction
: moveInstr
| arithmeticInstr
| branchInstr
| ioInstr
| callInstr
| returnInstr
| stopInstr
;
```

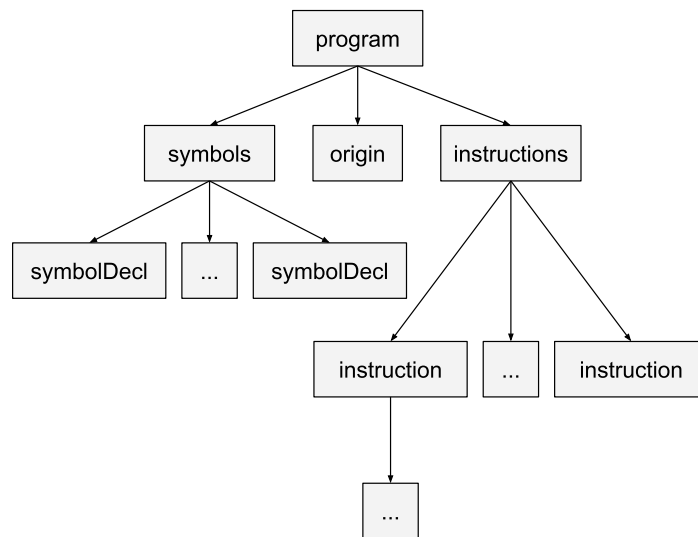
Код 4.1.7 Правило за парсирање линије

Правила за парсирање појединачних инструкција садрже назив инструкције и листу аргумената. Сами типови аргумената се углавном не проверавају за исправност приликом парсирања већ је то део семантичке анализе. Дозвољене комбинације аргумената у аритметичким инструкцијама би било тешко и нечитљиво покрити парсирањем.

4.2. Анализа програма

Након успешног парсирања програма потребно је анализирати стабло да би се откриле неправилности у дефинисању симбола, коришћењу лабела, типова аргумената и неправилном коришћењу инструкција. Класа *AnalysisVisitor* се користи за потребе анализе.

Слика 4.2.1 приказује пример синтаксног стабла парсираног програма. Обилазак стабла се ради по реду.



Слика 4.2.1 Пример генерисаног синтаксног стабла

Неправилности које се откривају током анализе се чувају и враћају на крају посете стабла. Контекстне класе које ANTLR генерише садрже информације о линији и позицији правила парсера и терминалних чворова које се користе за детаљније поруке о грешци, јер су те информације битне кориснику и корисничком интерфејсу.

4.2.1. Анализа симбола и почетка програма

Обиласком стабла је могуће у правилу *symbolDecl* добити име симбола и његову вредност. Користи се празна мапа симбола сваки пут кад се започне обилазак стабла програма и у њој се чувају вредности симбола. Потребно је проверити да ли је вредност симбола унутар дозвољеног опсега јер је дужина речи 16 бита (Код 4.2.1).

```

// Handle symbol declaration:
// * symbol must not exist
// * symbol value cannot be negative
AnalysisVisitor.prototype.visitSymbolDecl = function (ctx) {
  const identifier = ctx.identifier();
  const constant = ctx.constant();
  const name = identifier.accept(this);
  let value = constant.accept(this);
  // The symbol must not exist beforehand.
  if (this.hasSymbol(name)) {
    this.newError(identifier,
      errorMessages.symbolAlreadyDefined(name)
    );
    return;
  }
  // Symbol's value must be inside bounds.
  if (value < minConstant || value > maxConstant) {
    this.newError(constant,
      errorMessages.symbolValueOutOfRange(name, value)
    );
    value = 1;
  }
  this.setSymbol(name, value);
};

```

Код 4.2.1 Провера и чувања вредности симбола

Након анализе симбола проверава се да ли је директивом *org* задата исправна адреса.

4.2.2. Анализа инструкција

У зависности од типа инструкције потребно је проверити исправност аргумената као што је описано у поглављу 2.3 *Сет инструкција*.

Све инструкције проверавају по сличном шаблону:

- 1) Посете се аргументи да би се добили њихови типови
- 2) Провери се да ли типови аргумената одговарају тренутном типу инструкције
- 3) Провери се да ли вредност аргумената одговара опсегу који подржава тип инструкције

Слика 4.2.2 приказује редукован UML класни дијаграм посетиоца [6]. Анализа инструкција добија од посећених аргумената унификован тип *Argument* над којим се лако може проверити да ли је одређеног типа и забележити грешка.

Редослед обиласка стабла је следећи:

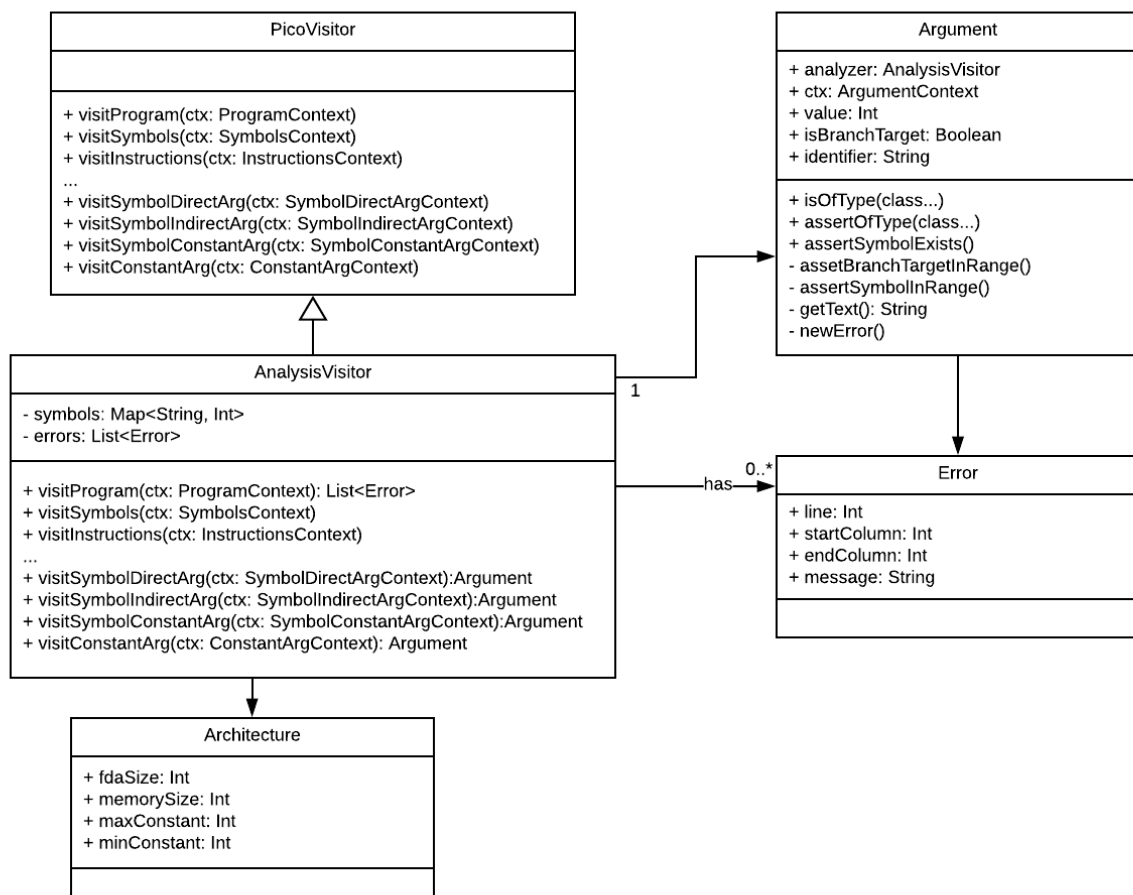
- 1) Анализа декларације симбола
- 2) Анализа *origin*-а
- 3) Анализа лабела
- 4) Анализа инструкција

Лабеле је могуће дефинисати након њиховог коришћења (Код 4.2.2) тако да се прво морају анализирати све лабеле, а након тога инструкције.

Лабеле се третирају као симболи тако да се не дозвољава да постоје две лабеле са истим именом или да лабела и симбол деле име.

```
bgt a, b, labela
labela: stop a
```

Код 4.2.2 Случај када се лабела користи пре дефинисања



Слика 4.2.2 UML класни дијаграм анализе

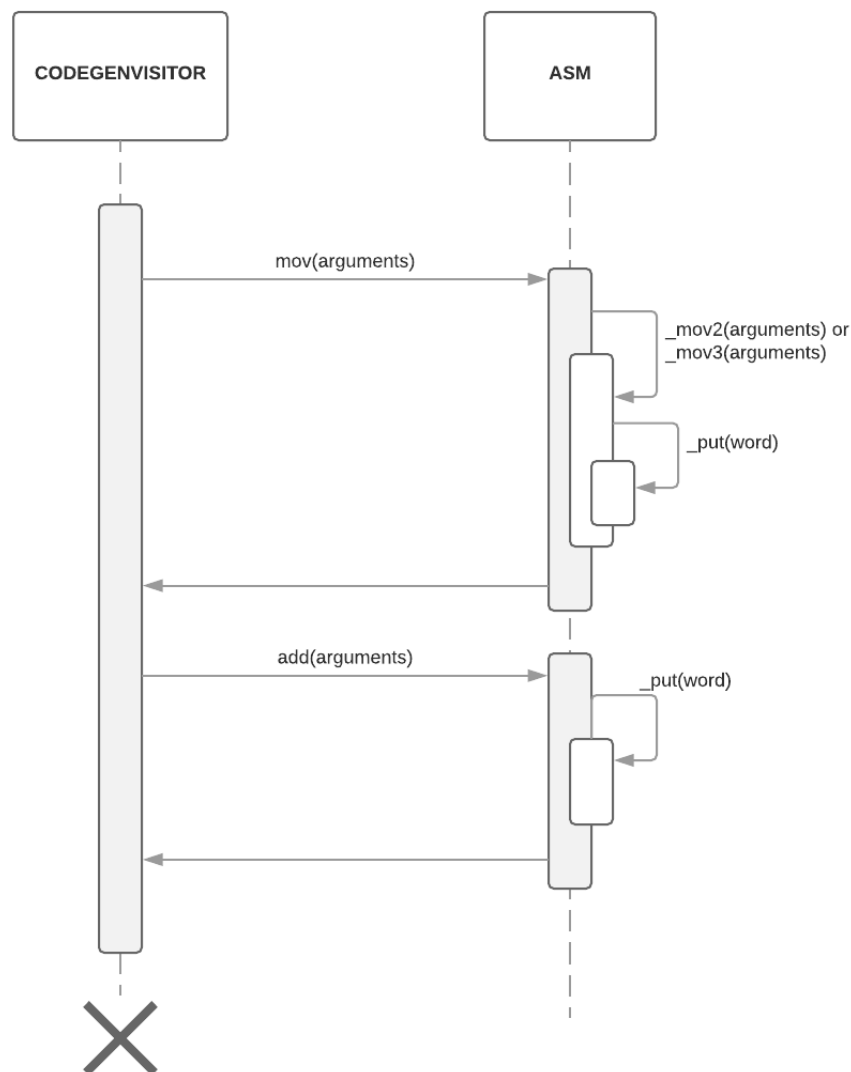
4.3. Преводјење програма

У случају да је написан програм семантички исправан могуће је генерисати бајткод који ће виртуелна машина извршавати. За те потребе је могуће користити *CodegenVisitor* класу.

CodegenVisitor обилази стабло по сличном принципу као посетилац у случају анализе. Прво је потребно проћи кроз дефинисане симболе и запамтити њихове вредности. Вредност почетка програма се такође памти јер је битна за чување вредности лабела обзиром да су адресе скока апсолутне вредности.

Користи се помоћна класа *Asm* која чува низ генерисаних бајтова и има методе за генерисање бајткода за сваку инструкцију. Методе знају да на основу типа аргумената препознају код операције и да генеришу једну или две речи инструкције и да исправно кодирају операнде у случају да инструкција кодира операнде на посебан начин. Слика 4.3.1 приказује дијаграм секвенце између посетиоца и бајткод генератора.

На крају обиласка стабла потребан је један пролаз кроз све инструкције скока да би се покрили случајеви када су лабеле дефинисане након коришћења. Приликом генерисања бајткода за инструкције скока оставља се празно место за адресу скока и оно се попуњава на крају када вредности буду познате.



Слика 4.3.1 Дијаграм секвенце приликом генерисања кода за 2 инструкције

Написан је и *DebuggableCodegenVisitor* који се користи у случајевима када је потребно омогућити дебаговање кода и постављање зауставних тачака. За сваку посећену линију чува се мапа линије кода на показивач РС (Табела 4.3.1, Код 4.3.1). Функционалност омогућује корисничком интерфејсу да зада команду за заустављање извршавања на линији изворног кода, а дебагер уме да мапира на адресу у меморији која представља инструкцију. Додатно се чува и мапа симбола и њихових адреса за време превођења тако да кориснички интерфејс може да прикаже промену вредности симбола.

```

1  ; Symbols
2  a = 1
3  b = 2
4  c = 3
5
6  ; Origin
7  org 8 ; Must start at 8 or above
8
9  ; Instructions
10 start:
11 in a
12 in b
13 add c, a, b
14 out c
15
16 in a
17 beq a, 0, end
18 beq a, a, start
19 end:
20 stop

```

Код 4.3.1 Сабирање два броја

Табела 4.3.1 Мапирање изворног кода на бајткод

Линија	Код	PC	Бајткод (хексадецимално)
11	in a	8	7101
12	in b	9	7201
13	add c, a, b	10	1312
14	out c	11	8301
16	in a	12	7101
17	beq a, 0, end	13	5118 0011
18	beq a, a, start	15	5118 0008
20	stop	17	f000

4.4. Извршавање и дебаговање

Виртуелна машина може да учита генерисан бајткод и почетну адресу генерисаног кода. Дозвољава се извршавање целе симулације или извршавање инструкција корак по корак.

Извршавање пролази кроз кораке:

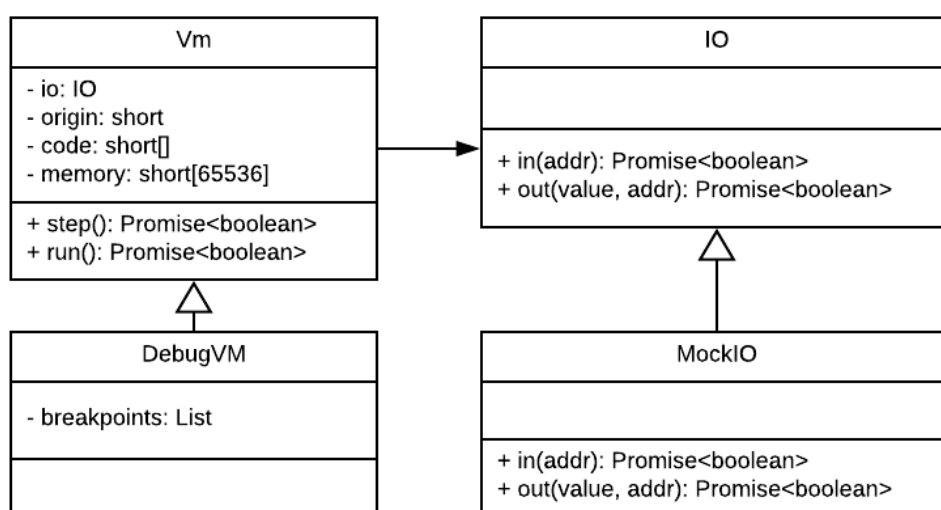
- 1) Читање и декодовање инструкције
- 2) Читање вредности или адреса операнда, у зависности од инструкције
- 3) Израчунавање резултата и уписивање резултата у меморију (уколико је потребно)

Свако читање инструкције инкрементира РС. Након декодовања инструкције у делу читања вредности операнда се додатно чита следећа реч инструкције (константа) уколико тип инструкције и операнди то налажу.

У/И инструкције заустављају извршавање док се захтевана вредност или више вредности не доставе виртуелној машини и не дозвољава се даље извршавање инструкција.

Приликом израчунавања вредности аритметичких израза се води рачуна о резултатима који излазе из граница дозвољених вредности јер су речи 16-то битне.

Слика 4.4.1 приказује класни UML дијаграм виртуелне машине. Виртуелна машина дати код учита у меморију на почетној адреси и користећи *step* и *run* је могуће извршавати симулацију корак по корак или целу одједном. *MockIO* постоји зарад тестирања и служи као УИ уређај који не захтева интеракцију са корисником.



Слика 4.4.1 UML класни дијаграм виртуелне машине

4.5. Тестирање

rico-asm има написан сет аутоматизованих тестова који тестирају функционалности парсирања, анализе, генерисања кода и извршавања. Тестови функционишу по принципу сачуваних резултата који се морају поклапати у сваком покретању тестова. Многи програмски језици тестирају своје компајлере на сличан начин [7].

4.5.1. Тестирање анализе

Тестирање анализе је подељено у тестове:

- 1) Анализе аритметичких инструкција
- 2) Анализе инструкција условног скока
- 3) Анализе У/И инструкција
- 4) Анализе инструкција копирања
- 5) Анализе зауставних инструкција
- 6) Анализе дефиниције симбола

У свим наведеним случајева анализирају се програми који садрже инструкције које су исправне и инструкције које у својој дефиницији имају аргументе који нису правилно дефинисани. Након анализирања се очекује да програми не буду исправни и да су се исправне поруке за грешку приказале. Код 4.5.1 приказује пример за тестирање зауставних инструкција. Код 4.5.2 приказује листу очекиваних грешака приликом анализе.

Приликом покретања тестова, уколико се понашање анализе промени од оног што је очекивано тестови ће пријавити грешку и лако се може установити разлика у понашању.

```
1  a=5
2  b=2
3  c=8
4  x=0
5  org 8
6  ; valid cases
7  stop a
8  stop b, (b)
9  ; invalid cases
10 stop c, (c)
11 stop #a, 5
12 stop x
13 stop d
```

Код 4.5.1 Програм за тестирање анализе зауставних инструкција

```
10:5:5:Address error in argument 'c'. Legal address [1..7].
10:8:10:Address error in argument '(c)'. Legal address [1..7].
11:5:6:Invalid argument type, expected one of {SymbolDirect,
SymbolIndirect}
11:9:9:Invalid argument type, expected one of {SymbolDirect,
SymbolIndirect}
12:5:5:Address error in argument 'x'. Legal address [1..7].
13:5:5:Symbol 'd' has not been defined.
```

Код 4.5.2 Очекиване грешке приликом анализе

4.5.2. Тестирање генерисања бајткода

Тестирање ради по сличном принципу као анализа кода, али се увек тестирају исправни програми и очекиван резултат је генерисан бајткод. Код 4.5.3 приказује пример кода за тестирање зауставних инструкција. Код 4.5.4 приказује очекиван бајткод генерисан од стране програма.

```
a = 1
b = 2
org 8
stop
stop a
stop a, (b)
stop a, (a), b
```

Код 4.5.3 Програм за тестирање генерисања бајткода зауставних инструкција

```
0xf000 1111 0000 0000 0000
0xf100 1111 0001 0000 0000
0xf1a0 1111 0001 1010 0000
0xf192 1111 0001 1001 0010
```

Код 4.5.4 Очекиван бајткод од стране програма Код 4.5.3

4.5.3. Тестирање извршавања

За тестирање извршавања се користе примери програма са предмета “Програмирање 1” и збирке задатака [8]. Програмима се зада неки улаз и очекује тачан излаз и настоји се да се покрије што већи број функционалности виртуелне машине. Код 4.5.5 приказује програм који сабира 2 задата броја. Код 4.5.6 приказује задати улаз и очекиван излаз програма.

```
; Symbols
a = 1
b = 2
c = 3

; Origin
org 8 ; Must start at 8 or above

; Instructions
start:
    in a          ; input a
    in b          ; input b
    add c, a, b    ; c = a + b
    out c         ; output c

    in a          ; input a
    beq a, 0, end  ; if a=0 goto end
    beq a, a, start ; goto start
end:
    stop ; stop execution
```

Код 4.5.5 Програм који сабира 2 задата броја

```
{
  "inputs": [
    1,
    3,
    1,
    2,
    3,
    0
  ],
  "outputs": [
    4,
    5
  ]
}
```

Код 4.5.6 Очекиван улаз и излаз програма

Виртуелној машини је могуће задати било који улаз без интеракције корисника користећи *MockIO* описан у секцији 4.4 *Извршавање и дебаговање* и описан механизам се користи за покретање тестова.

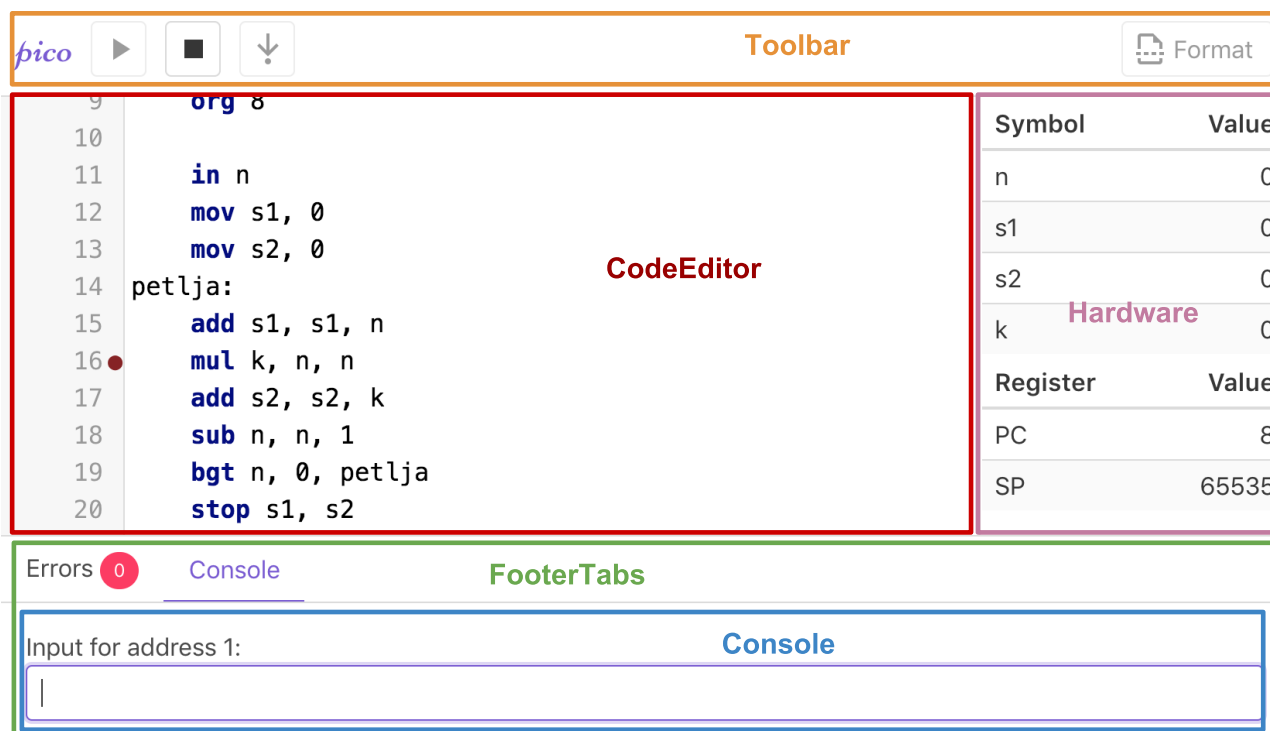
5. РЕАЛИЗАЦИЈА КОРИСНИЧКОГ ИНТЕРФЕЈСА

Кориснички интерфејс је заснован на *create-vue-app* и користи библиотеке *Vue.js*, *Vuex*, *Buefy* и *CodeMirror*.

5.1. *Vue.js*

Vue.js је једноставна библиотека која олакшава развој великих апликација. Могуће је развијати независне компоненте и њих комбиновати тако да се на крају добије повезана апликација (**Error! Reference source not found.**). Сами елементи корисничког интерфејса као што су дугмићи, иконе и табеле су увезени из пакета *Buefy*. *Buefy* је пакет компоненти написаних за *Vue.js* који користе пакет *Bulma* за стилове и изглед апликације.

CodeMirror је библиотека која се користи за едитор кода. Лако се интегрише у пројекат и подржава подешавање изгледа и интеракције са едитором и био је логичан избор за интеграцију у развојно окружење. Препознавање кључних речи и коментара у самом едитору је функционалност библиотеке и користи посебно написан мод [9].

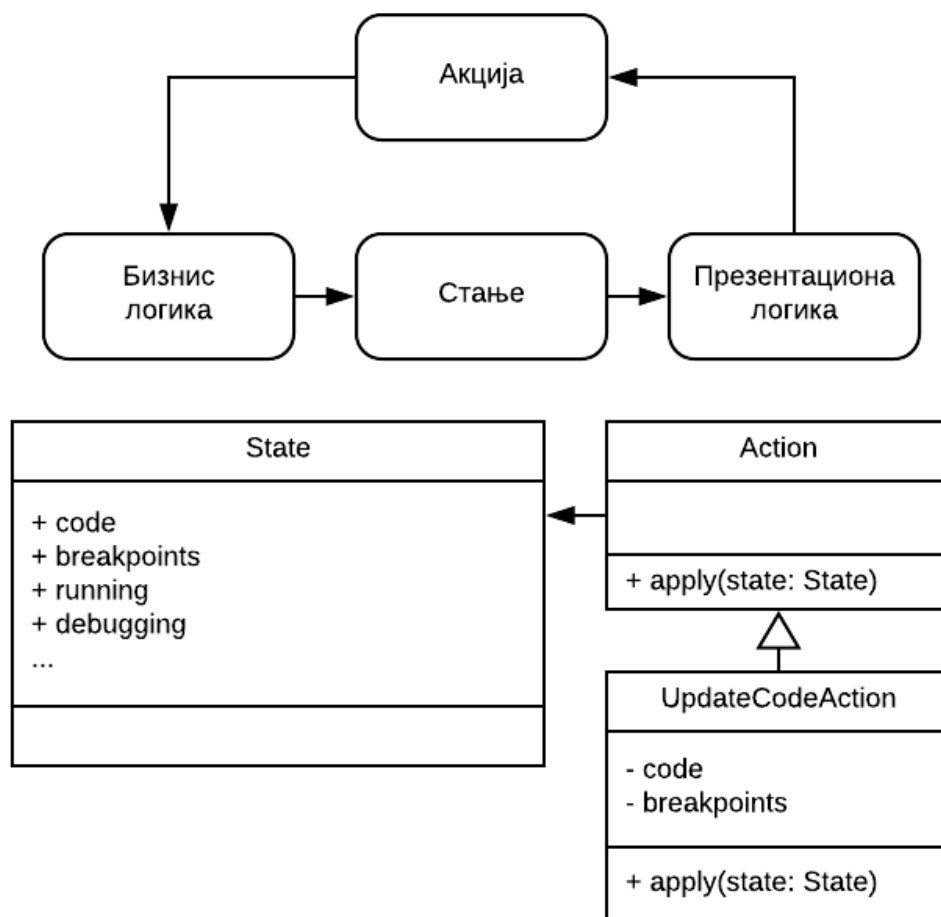


Слика 5.1.1 Преглед компоненти корисничког интерфејса

5.2. Повезивање парсера са корисничким интерфејсом

Vue.js дозвољава раздвајање презентационе и бизнис логике користећи библиотеку *Vuex*. Компоненте имају свој приказ преко ког шаљу акције проузроковане корисничким интеракцијама. Послате акције мењају глобално стање, и тада се то стање враћа компонентама које мењају свој приказ (Слика 5.2.1).

У тренутном стању апликације се чувају подаци који су битни за приказ извршавања или захтевања акције од корисника. Акција *updateCode* (Слика 5.2.1) прима код који треба парсирати и зауставне тачке, и та акција треба да промени стање – у случају акције *updateCode* то подразумева само мењање тренутно записаног кода и зауставних тачки. Мењање кода на сачуваном стању проузрокује поновно парсирање и тиме компонента *ErrorList* добија листу грешака коју треба да прикаже. Уколико нема грешака, интерфејс зна да треба да омогући почетак симулације кориснику.



Слика 5.2.1 Дијаграм стања слања акција и упрошћени класни дијаграм

Код 5.2.1 приказује пример компоненте *Toolbar* који се везује помоћу *Vuex-a* и прати извршавање симулације док постоји листа акција које је могуће позвати.

Овај приступ доста упрошћава писање компоненти јер постоји један извор истине (Single Source of Truth). Друге компоненте могу на исти начин да добију информацију о грешкама, да ли је могуће покренути симулацију, да ли је симулација тренутно у току итд.

```

import {mapActions, mapGetters, mapState} from "vuex";

export default {
  name: "Toolbar",
  computed: {
    ...mapGetters(["errors", "canRun", "canContinue"]),
    ...mapState(["running"]),
  },
  methods: {
    ...mapActions([
      "debug",
      "run",
      "stop",
      "step",
      "format",
    ]),
    ...mapActions({
      "continueExecution": "continue"
    })
  },
};

```

Код 5.2.1 Компонента Toolbar – JavaScript без HTML-a

6. ЗАКЉУЧАК

Развојно окружење за picoComputer на веб-у би требало да помогне студентима и осталим заинтересованима да се на лак начин упознају са picoComputer архитектуром и пишу програме које је могуће извршавати и дебаговати корак по корак.

Написан сет тестова ће помоћи осталима који у будућности буду желели да наставе рад на развојном окружењу као и онима који буду желели да напишу симулатор у неком другом облику.

Даљи развој окружења би ишао у правцу покривања напреднијих функционалности:

- Чување и учитавање програма – тренутно није могуће сачувати написан програм у фајл и учитати га поново или поделити са другим особама.
- Имплементација *Step Over* и *Step Out* дебаговања – развојно окружење подржава само *Step In*.
- Праћење произвољних меморијских локација – омогућено је праћење вредности у FDA, али није могуће пратити индиректна меморијска адресирања и елементе низова.
- Дефинисање зауставних тачака за приступ меморијским локацијама – “MessyLab” подржава заустављање извршавања уколико инструкција приступи дефинисаној меморијској локацији.
- Извршавање бајткода – иако виртуелна машина извршава бајткод, кориснички интерфејс нема подршку за извршавање унетог корисничког бајткода.

Развојно окружење је доступно на адреси <https://picosim.app> [10] за коришћење. Изворни код је јавно доступан на GitHub-у под MIT лиценцом како би се охрабрио наставак развоја.

Литература

- [1] J. Dujmović, *Programski jezici i metode programiranja*, Akademska misao, 2004
- [2] *picoComputer – MessyLab* [Online]. Available: <http://messylab.com/pico/> (19.09.2018.)
- [3] T. Parr, *The Definitive ANTLR 4 Reference*, The Pragmatic Programmers, 2013
- [4] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994
- [5] *Runtime Libraries and Code Generation Targets* [Online]. Available: <https://github.com/antlr/antlr4/blob/master/doc/targets.md> (19.09.2018.)
- [6] G. Bosch, J. Rumbaugh, I. Jacobson, *The Unified Modeling Language User Guide* 2nd edition, Addison-Wesley, 1999
- [7] *Go compiler tests* [Online]. Available: <https://github.com/golang/go/tree/master/test> (21.09.2018.)
- [8] L. Kraus, *Zbirka zadataka iz programskih jezika 1*, Akademska misao, 2006
- [9] *CodeMirror: User Manual* [Online]. Available: <https://codemirror.net/doc/manual.html#modeapi> (21.09.2018.)
- [10] *picoComputer Web IDE* [Online]. Available <https://picosim.app/> (21.09.2018.)

Списак скраћеница

pCAS	pC Assembler and Simulator
FDA	Fixed Data Area
SP	Stack Pointer
PC	Program Counter
ANTLR	ANother Tool for Language Recognition

Списак слика

Слика 2.1.1 Хардверска организација picoComputer архитектуре	2
Слика 2.1.2 Подела меморије picoComputer-а	2
Слика 4.2.1 Пример генерисаног синтаксног стабла	17
Слика 4.2.2 UML класни дијаграм анализе	19
Слика 4.3.1 Дијаграм секвенце приликом генерисања кода за 2 инструкције	20
Слика 4.4.1 UML класни дијаграм виртуелне машине	22
Слика 5.1.1 Преглед компоненти корисничког интерфејса	25
Слика 5.2.1 Дијаграм стања слања акција и упрошћени класни дијаграм	26

Списак кода

Код 2.2.1 Програм који сабира два броја	4
Код 2.3.1 Коришћење инструкције копирања типа 1	6
Код 2.3.2 Коришћење инструкције копирања типа 2	6
Код 2.3.3 Коришћење инструкције копирања типа 3	7
Код 2.3.4 Коришћење инструкције копирања типа 4	7
Код 2.3.5 Коришћење аритметичких инструкција без константе	8
Код 2.3.6 Коришћење аритметичких инструкција са константом – други операнд	8
Код 2.3.7 Коришћење аритметичких инструкција са константом – трећи операнд	9
Код 2.3.8 Коришћење инструкција условног скока са лабелом	10
Код 2.3.9 Коришћење инструкција са скоком на адресу која се налази у меморији	10
Код 2.3.10 Коришћење скока на потпрограм дефинисаног лабелом <i>sub</i>	11
Код 2.3.11 Коришћење инструкције за повратак из потпрограма	12
Код 2.3.12 Коришћење инструкције за заустављање извршавања програма	12
Код 4.1.1 Граматика структуре програма	14
Код 4.1.2 Генерисање парсера (src/parser/compile.sh)	15
Код 4.1.3 Пример генерисаних метода	15
Код 4.1.4 Исправно дефинисани симболи	15
Код 4.1.5 Граматика дефинисања симбола	15
Код 4.1.6 Правило за парсирање адресе почетка програма	16
Код 4.1.7 Правило за парсирање линије	16
Код 4.2.1 Провера и чувања вредности симбола	17
Код 4.2.2 Случај када се лабела користи пре дефинисања	18
Код 4.3.1 Сабирање два броја	21
Код 4.5.1 Програм за тестирање анализе зауставних инструкција	23
Код 4.5.2 Очекиване грешке приликом анализе	23
Код 4.5.3 Програм за тестирање генерисања бајткода зауставних инструкција	23
Код 4.5.4 Очекиван бајткод од стране програма Код 4.5.3	23
Код 4.5.5 Програм који сабира 2 задата броја	24
Код 4.5.6 Очекиван улаз и излаз програма	24
Код 5.2.1 Компонента Toolbar – JavaScript без HTML-а	27

Списак табела

Табела 2.2.1 Типови операнада	5
Табела 2.3.1 Формат инструкција	5
Табела 2.3.2 Кодови операција инструкција копирања.....	6
Табела 2.3.3 Формат инструкције копирања типа 1	6
Табела 2.3.4 Формат инструкције копирања типа 2	6
Табела 2.3.5 Формат инструкције копирања типа 3	7
Табела 2.3.6 Формат инструкције копирања типа 4	7
Табела 2.3.7 Кодови операција аритметичких инструкција.....	8
Табела 2.3.8 Формат аритметичких инструкција без константе	8
Табела 2.3.9 Аритметичка инструкција са константом – трећи операнд	9
Табела 2.3.10 Аритметичка инструкција са константом – трећи операнд.....	9
Табела 2.3.11 Кодови операција инструкција условног скока	9
Табела 2.3.12 Формат инструкција условног скока са константном адресом – скок на лабелу	10
Табела 2.3.13 Формат инструкција са скоком на адресу која се налази у меморији.....	10
Табела 2.3.14 Кодови операција У/И инструкција	10
Табела 2.3.15 Формат У/И инструкција када је дужина низа меморијска локација	11
Табела 2.3.16 Формат У/И инструкција када је дужина низа константа	11
Табела 2.3.17 Кодови операција инструкција скока и повратка из потпрограма.....	11
Табела 2.3.18 Формат инструкција за скок на потпрограм када је адреса скока константа	11
Табела 2.3.19 Формат инструкције за повратак из потпрограма	12
Табела 2.3.20 Формат инструкције за заустављање извршавања програма	12
Табела 4.3.1 Мапирање изворног кода на бајткод.....	21

А. Упутство за подешавање развоја на окружењу

Развој окружења захтева инсталиран *NodeJS* и менаџер пакета *Yarn*. Препоручује се инсталирање истих користећи *nvm* (Node Version Manager). Упутства за инсталирање *nvm*-а се могу пронаћи на веб страници <https://github.com/creationix/nvm>.

А.1. РАЗВОЈ PICO-ASM БИБЛИОТЕКЕ

У директоријуму *pico-asm* је потребно инсталирати пакете:

```
yarn install
```

За покретање тестова је довољно:

```
yarn test
```

А.2. РАЗВОЈ PICO-SIM ОКРУЖЕЊА

У директоријуму *pico-sim* је потребно инсталирати пакете:

```
yarn install
```

За покретање сервера који ће се подразумевано покренути на `http://localhost:4000`:

```
yarn dev
```

Инсталирање пакета ће инсталирати верзију *pico-asm* пакета која је јавно доступна на репозиторијуму `npmjs.org`. Уколико је жеља развој са измењеном верзијом парсера:

```
yarn add file:/putanja/do/pico-asm
```

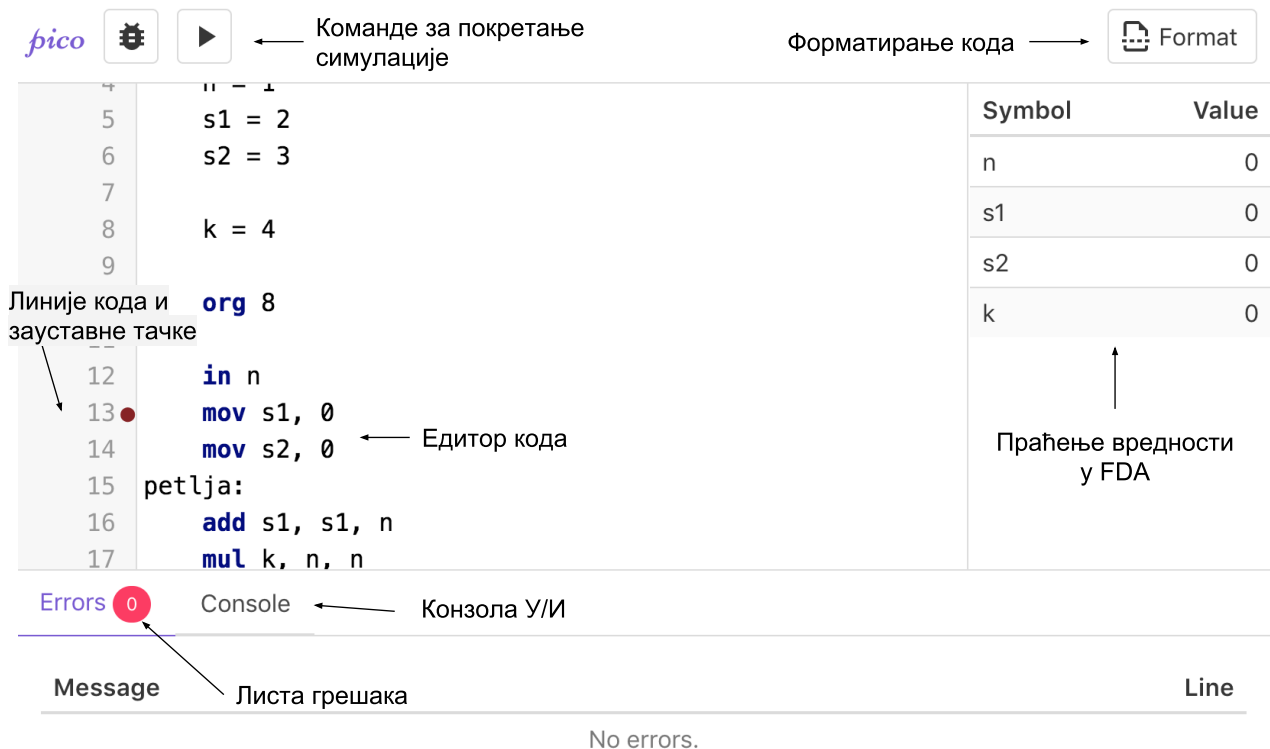
В. Упутство за коришћење

У овом додатку је описано упутство за коришћење развојног окружења на <https://picosim.app/>.

В.1. ПИСАЊЕ ПРОГРАМА

Слика В.1.1 приказује преглед развојног окружења:




- У горњем делу окружења налазе се команде за покретање симулације и форматирање кода.
- У средишњем делу се налази едитор кода и пратилац вредности меморијских локација у FDA
- У доњем делу се налазе листа грешака и конзола која даје преглед У/И операција



Слика В.1.1 Преглед развојног окружења

Током писања програма едитор води рачуна да програм буде синтаксички и семантички исправан. У случају да постоји нека грешка биће онемогућено покренути програм. Грешке ће бити обележене у самом едитору и испод едитора у листи грешака (Слика В.1.2 и Слика В.1.3).

Кључне речи су подебљане и обојене другом бојом, а исто важи и за коментаре у коду ради лакшег прегледа програма.

pico    Format




		Symbol	Value
1	a==1		
2	org 8		
3	stop a		

Symbols will appear

Errors **1** Console

Message	Line
extraneous input '=' expecting {NUMBER, SIGN}	1

Слика В.1.2 Грешка приликом парсирања приказана у листи и едитору

pico    Format

		Symbol	Value
1	a=1		
2	org 8		
3	stop b		

Symbols will appear

Errors **1** Console

Message	Line
Symbol 'b' has not been defined.	3

Слика В.1.3 Семантичка грешка приказана у листи и едитору

Кликом на дугме *Format* код се програмски форматира да постане читљивији (Слика В.1.4).

1	n = 1	1	n = 1
2	s1 = 2	2	s1 = 2
3	s2 = 3	3	s2 = 3
4	k = 4	4	k = 4
5		5	
6	org 8	6	org 8
7	in n	7	in n
8	mov s1, 0	8	mov s1, 0
9	mov s2, 0	9	mov s2, 0
10	petlja:add s1, s1, n ; add n	10	petlja:
11	mul k,n,n	11	add s1, s1, n ; add n
12	add s2, s2, k	12	mul k, n, n
13	sub n, n, 1	13	add s2, s2, k
14	bgt n, 0, petlja	14	sub n, n, 1
15	stop s1, s2	15	bgt n, 0, petlja
16		16	stop s1, s2

Слика В.1.4 Форматирање кода – пре (лево) и после (десно)

В.2. ИЗВРШАВАЊЕ СИМУЛАЦИЈЕ

Када је програм исправно написан омогућено је покретање са или без подршке за дебаговање. У едитору је могуће постављање зауставних тачака поред инструкција на којима ће се извршавање зауставити. Кликом на број линије поред инструкције се поставља или склања зауставни маркер (Слика В.2.1).

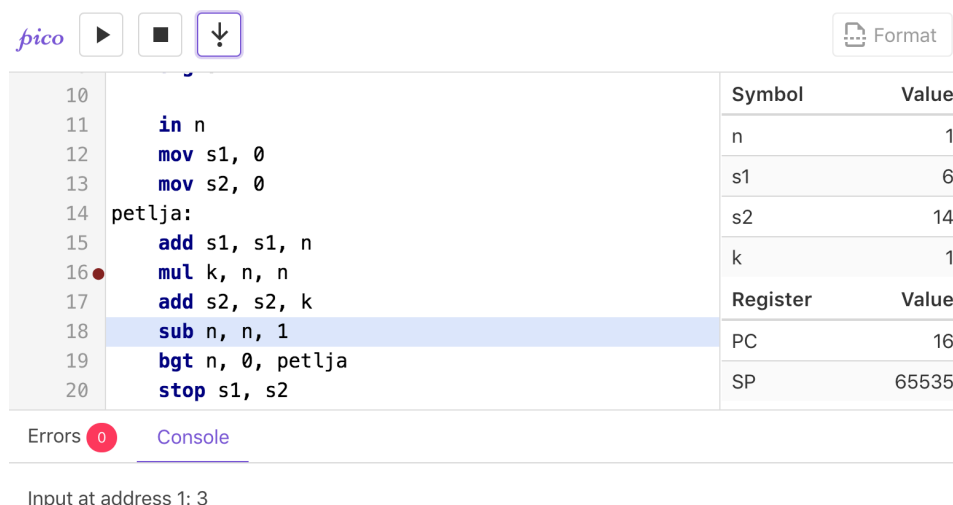
Приликом дебаговања извршавања, следећа по реду инструкција која ће се извршити је обележена (Слика В.2.2).

На десној страни симулатора је могуће пратити вредности у локацијама у FDA и регистре PC и SP (Слика В.2.2).

Приликом захтевања улаза од корисника, у одељку *Console* ће се приказати поље за унос податка (Слика В.2.3).



Слика В.2.1 Црвеним маркером обележене инструкције на којима ће се дебагер зауставити



Слика В.2.2 Обележена следећа инструкција по реду и праћење вредности симбола

Errors 0 Console

Input for address 1:

Слика В.2.3 Поље за унос податка