Softverska implementacija Floating - Point jedinice za Hack računalo

Kordić, Vedrana

Sveučilište J. J. Strossmayera u Osijeku

Fakultet primijenjene matematike i informatike

Sveučilišni preddiplomski studij Matematika i računarstvo

Softverska implementacija Floating-Point jedinice za Hack računalo

Završni rad

Mentor:

izv. prof. dr. sc. Domagoj Matijević

Komentor: Kandidat:

dr. sc. Luka Borozan **Vedrana Kordić**

Osijek 2023.

Sažetak

U ovom radu ćemo promatrati softverski dio izrade 16-bitnog računala u sklopu „Nand2Tetris“ projekta te doradu istoga kako bi mogao zapisati i manipulirati brojevima s pomičnim zarezom. Upoznat ćemo se s osnovama rada hardvera, asemblera te cjelokupnom sintaksom jezika virtualnog stroja te programskog jezika Jack te izraditi kompajler za jezik Jack.

Ključne riječi

float, softver, virtualni stroj, asembler, nand2tetris, kompajler, Jack

Software implementation of the Floating-Point Unit for the Hack computer

Summary

In this project, we will be focusing on the software aspect of building a 16-bit computer as part of the "Nand2Tetris" project and enhancing it to be able to handle floating-point numbers. We will get familiar with the basics of hardware operation, assembly language, the overall syntax of the virtual machine language, and the Jack programming language. Additionally, we will create a compiler for the Jack language.

Keywords

float, software, virtual machine, assembler, nand2tetris, compiler, Jack

Sadržaj

1. Uvod

Na drugoj godini preddiplomskog studija Matematika i računarstvo susreli smo se kolegijem Moderni računalni sustavi unutar kojeg smo pratili „Nand to Tetris“ projekt izrade 16-bitnog Hack računala od najniže razine koristeći logičke sklopove pa sve do programskog jezika visoke razine i pisanja kompajlera. Generalno građa Hack računala sastoji se od hardverske platforme, asemblera, virtualnog stroja te kompajlera za programski jezik Jack. Popratno su implementirani parseri za asmebler i jezik virtualnog stroja. Korist ovog projekta je što pokazuje koliko je računalo ustvari moćan alat današnjice te omogućuje uvid u rad računala na što jednostavniji način. Također omogućuje shvaćanje kako svaki tip podataka funkcionira u računalu prateći ga od najviše razine u programskom jeziku Jack pa sve do binarnog zapisa. Doradom projekta u sve dijelove Hack računala uspjeli smo integrirati mogućnost zapisa i rada s brojevima s pomičnim zarezom. Preciznije, implementirana je hardverska platforma koja ima mogućnost rada s brojevima s pomičnim zarezom, unaprijeđeni parseri za asembler i virtualni stroj te poboljšan kompajler kako bi mogao prevoditi brojeve s pomičnim zarezom.

Kroz ovaj rad opisati ćemo standard korišten za zapis brojeva s pomičnim zarezom, ukratko objasniti građu Hack računala te podrobnije opisati jezik virtualnog stroja, programski jezik Jack te na posljetku opisati kompajler koji dodatno može prevesti i brojeve s pomičnim zarezom.

Potpun projekt se može naći u GitHub repozitoriju na sljedećem linku :
https://github.com/vedranakordic/Floating-Point-Unit-for-the-Hack-computer.git

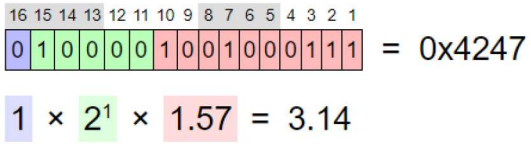
2. IEEE 754 Standard

Upotrebom koncepta poznatog kao "znanstveni zapis brojeva", moguće je zapisivati brojeve na način da se prvo napišu vodeće značajne znamenke broja, a zatim se koristi faktor izražen kao eksponent povezan s određenom bazom. U slučaju binarnog zapisa brojeva, rezultat je dobiven množenjem eksponenta, mantise i potencije broja 2 podignute na taj eksponent.

IEEE 754 standard je način za računalno zapisivanje racionalnih brojeva nastao 1985. godine. Standard razlikuje dva formata zapisivanja brojeva: jednostruke preciznost, tj. koristeći 32-bitnu reprezentaciju te dvostruke preciznosti koristeći 64-bitnu. Također postoji varijanta i za 16-bitnu reprezentaciju.

Korištenjem IEEE 754 standarda, brojevi se prikazuju na sljedeći način: prvi bit označava predznak, 1 za negativan te 0 za pozitivan broj. U slučaju 32-bitne reprezentacije, sljedećih 8 bitova predstavlja eksponent. Eksponent se računa kao zbroj eksponenta originalnog broja i fiksne konstante, koja je ovdje 127. U 16-bitnoj reprezentaciji kao što je vidljivo i na Slici 1., eksponent je prikazan sa 5 bitova i računa se na isti način s konstantom 15. Zadnjih 23 bita u 32-bitnoj reprezentaciji (ili 10 bitova u 16-bitnoj reprezentaciji) predstavljaju mantisu. Mantisa se računa tako da se uzmu svi brojevi iza decimalne točke u znanstvenom zapisu originalnog broja, a zatim se dopuni s nulama do potrebne duljine (23 bita za 32-bitnu, odnosno 10 bitova za 16-bitnu reprezentaciju). U slučaju da je prvi bit mantise 1, njega nazivamo skriveni bit te ga ne spremamo, ali ga koristimo za izračunavanje mantise kod aritmetičkih operacija.

Glavni cilj korištenja ovog standarda jest zapis brojeva na način da se osigurava određena preciznost, mogućnost provođenja računskih operacija uz relativno minimalne pogreške, tj. u našem slučaju rezultati se razlikuju do na treću decimalu za sve implementirane operacije izuzev množenja. Također omogućavanje zapisa specijalnih brojeva poput beskonačnosti, „NaN“(nije broj).



Slika 1. Primjer zapisa broja s pomičnim zarezom koristeći IEEE 754 Standard

3. Građa HACK računala

"Nand to Tetris" je obrazovni projekt usmjeren na izgradnju vlastitog funkcionalnog računala, počevši od osnovnih logičkog komponenti pa sve do stvaranja operativnog sustava i aplikacija. Izgradnja vlastitog računala obuhvaća izradu hardverskog djela 16-bitnog Hack računala, parsere za jezik asemblera, jezik virtualnog stroja te naposljetku kompajler. Hack računalo koje je posljedica tog projekta je u mogućnosti raditi samo s cijelim brojevima, stoga smo odabrali unaprijediti Hack računalo kako bi imalo mogućnost zapisa i rada s brojevima s pomičnim zarezom.

Za zapis brojeva s pomičnim zarezom korišten je IEEE 754 standard za 16-bitnu reprezentaciju. Zbog toga što se i sam zapis brojeva razlikuje jer za zapis cijelih brojeva koristimo klasičnu binarnu reprezentaciju cijelih brojeva, a za brojeve s pomičnim zarezom IEEE 754 Standard bilo je potrebno izraditi FPU, Floating Point Unit. FPU je zadužena za rad s brojevima s pomičnim zarezom, dok je za rad s cijelim brojevima zadužena ALU, Arithmetic Logic Unit. Unutar FPU definirane su operacije zbrajanja, oduzimanja, veće od, manje od za brojeve s pomičnim zarezom, a ostale operacije poput izjednačavanja, logičko i, logičko ili itd. su jednake kao i u ALU. Operacije su kreirane koristeći osnovne logičke komponente te pomoćne čipove koji su već kreirani u HDL-u, Hardware description language. Za rad s podatcima u procesoru koriste se A-instrukcije koje služe za adresiranje registra u memoriji i C-instrukcije kojima obavljamo željene operacije nad brojevima. Stoga je bilo nužno razlikovati C-instrukcije koje rade s cijelim brojevima te one koje rade s brojevima s pomičnim zarezom na način da prva dva bita binarne reprezentacije instrukcije određuju hoće li naredbu odraditi ALU ili FPU. U slučaju da su prva dva bita 11, naredbu obavlja ALU, a ukoliko su 10 FPU. Slijedom toga razlikovale su se naredbe i u asembleru, odnosno definirane su nove naredbe za rad s brojevima s pomičnim zarezom koje se razlikuju s znakom „*“, npr. „D = *M“. Kasnije će se naredbe za brojeve s pomičnim zarezom u „.vm“ dokumentima zvati „floatadd“, „floatsub“, „floatgt“, „floatlt“ te će im biti pridružene asemblerske naredbe sa znakom „*“. U jeziku Jack takve razlike neće postojati već će se po tipu podataka nad kojim se vrši operacija razlikovati te prilikom prevođenja u jezik virtualnog stroja odlučiti radi li se o „float“ naredbi ili ne. Svaki dokument napisan u HDL-u, asembleru, jeziku virtualnog stroja ili u programskom jeziku Jack ili preveden koristeći neki od prevoditelja je moguće testirati u pomoćnim alatima i emulatorima koji su dostupni na „Nand to Tetris“ web stranici.

4. Virtualni stroj

Virtualni stroj je softver za virtualizaciju i emulaciju računalnih sustava koji omogućava funkcionalnosti fizičkog računala te služi za pokretanje programa i operacijskih sustava. Razlikuju se dvije vrste virtualni strojeva: sistemski virtualni strojevi koji se još nazivaju i virtualnim strojevima za punu virtualizaciju te procesni virtualni strojevi koji su dizajnirani za izvršavanje računalnih programa neovisno o platformi.

U slučaju jezika virtualnog stroja računala Hack te i samog virtualnog stroja nema nikakvih posebnih zahtjeva i standarda za pisanje programa u istom za razliku od Jave i sl. Koristi se apstrakcija memorije pomoću stoga i memorijskih segmenata te se programi pišu u jeziku virtualnog stroja .

4. 1. Sintaksa jezika virtualnog stroja

Sintaksa jezika virtualnog stroja je relativno jednostavna jer se koristi apstrakcija stoga pa zbog toga razlikujemo neke osnovne naredbe kao što su pop, push te dodatno call, return, function, label, goto, if-goto, oznake za računske operacije itd. Naredbama push i pop dodajemo, odnosno uklanjamo prvi element sa vrha stoga, a ostale operacije obavljamo tako da uklonimo njihove argumente sa stoga te postavimo rezultat na vrh. Push i pop naredbe se pišu kao „push(pop) segment indeks“ naredba, gdje indeks predstavlja vrijednost koja se dodaje ili uklanja sa stoga. Segment dio označava radi li se o „local“, „argument“, „static“, „constant“, „this“, „that“, „pointer“ ili „temp“, a to nam je bitno kako bismo znali „scope“ varijable, argumenta, tj. tip onoga što želimo dodati ili ukloniti sa stoga te također kako bismo znali kako se alocira u memoriji i koja je inicijalna vrijednost. „Local“ predstavlja lokalne varijable koje se nalaze unutar neke funkcije te su one inicijalno 0, argument sadrži argumente funkcije te se dinamički alocira, „static“ sadrži sve statičke varijable koje se nalaze unutar jednog „.vm“ dokumenta te je statički alociran, „constant“ predstavlja segment koji sadrži sve konstante koje su dopuštene, tj. sve konstante koje računalo može zapisati, što je u ovom slučaju od 0 pa do 32767. „This“ i „that“ su segmenti opće namjene i predstavljaju objekt, odnosno polje s kojim trenutno manipuliramo, „temp“ sadrži osam memorijskih adresa te služi za interne potrebe kompajlera te „pointer“ koji sadrži bazne adrese od „this“ i „that“ segmenata. U ovom dijelu je bilo nužno unaprijediti parser za jezik virtualnog stroja te kasnije i kompajler što je vidljivo u poglavlju 6 Kompajler kako bi imali mogućnost rada s „float“ brojevima. Parser za jezik virtualnog stroja prevodi „.vm“ dokumente u „.asm“ dokumente stoga smo morali naći način razlikovanja zapisa cijelih brojeva u asembleru te brojeva s pomičnim zarezom što je učinjeno koristeći znak „*“. Točnije, za svake asemblersku naredbu iza znaka „=“ je slijedio asterisk.

Za pozivanje funkcije, odnosno naredbu „call function“ je bitno da su argumenti funkcije koja se poziva već na stogu. Kada se funkcija krene izvršavati, lokalne varijable se alociraju te inicijalno postavljaju na 0, static segment koji je pozvao funkciju se postavlja na static segment točno određen za taj dokument te je stog u tom trenutku prazan. Prije vraćanja rezultata funkcije rezultat se postavlja na vrh stoga. Nakon toga segmenti argument, local, static, this, that i pointer su postavljeni na vrijednosti prije poziva funkcije, a segment temp sam virtualni stroj ne koristi.

```
1  function fibonacci 1
2  push argument 0
3  push constant 2
4  lt
5  if-goto BaseCase
6  push argument 0
7  push constant 2
8  sub
9  call fibonacci.fibonacci 1
10 push argument 0
11 push constant 1
12 sub
13 call fibonacci.fibonacci 1
14 add
15 return
16
17 label BaseCase
18 push argument 0
19 return
```

Slika 2. Primjer VM programa za izračun Fibbonacijevih brojeva

Oznake definiramo ključnom riječju label te one označavaju trenutnu lokaciju u kodu. One služe za goto i if-goto naredbe jer se jedino na označene dijelove koda može „skočiti“. Oznake su vidljive samo unutar funkcije u kojoj su definirane, a nazivi oznaka su nizovi slova, brojeva, znaka „_“, točke, dvotočke te ne smiju započinjati znamenkom. Kao što vidimo na Slici 2. gdje imamo primjer funkcije koja računa n-ti Fibbonacijev broj, postoji oznaka BaseCase koja je definirana u 17. liniji. Također vidimo i ostale već spomenuto naredbe poput call, push, pop itd.

Goto i if-goto predstavljaju bezuvjetne, odnosno uvjetne skokove. Izvršavanjem goto naredbe izvršavanje programa prelazi na označeni dio koda, tj. oznaku koja mora biti definirana unutar iste funkcije te se prolazak kroz kod nastavlja od tog djela. If-goto naredba također predstavlja „skok“ na određeni dio koda, ali uz uvjet. Ovisno o vrijednosti na vrhu stoga koja ovisi je li uvjet ispunjen, tj. ako je vrijednost različita od nule, izvršavanje koda se nastavlja od određene oznake, a ukoliko je vrijednost nula, prelazi se na sljedeću naredbu koja se nalazi odmah nakon. Kao što je vidljivo na Slici 3. prvo odlazimo na oznaku x gdje se nalaze neke naredbe te zatim ako je ispunjen uvjet program prelazi na oznaku, a ukoliko ne izvršavaju se naredbe koje se nalaze na liniji 5. Nakon što se te naredbe provedu dolazimo do oznake y gdje

se opet nalaze neke naredbe te opet ukoliko je ispunjen uvjet program prelazi na oznaku z, inače program nastavlja od linije 9.

```
1  ......
2  label x
3      command
4      if condition goto y
5      command
6  label y
7      command
8      if condition goto z
9      command
10 label z
11     command
12     goto x
13 ......
```

Slika 3. Generalni primjer uvjetnih i bezuvjetnih skokova

Također kao što je ranije rečeno, razlikujemo oznake za neke osnovne operacije poput zbrajanja, oduzimanja, množenja, dijeljenja, veće od, manje od, jednako te osnovnih logičkih operacija poput negacije, i te ili za brojeve s pomičnim zarezom. Sintaksa operacija je jednostavna jer se koriste samo engleski nazivi istih, no razlikujemo operacije za cijele brojeve i brojeve s pomičnim zarezom. Operacije za brojeve s pomičnim zarezom imaju prefiks „float“ te se prilikom prevođenja u „.asm“ dokument koristeći već spomenute razlike koristeći asterisk kao što je vidljivo na Slici 4.

```
71 //floatadd
72 @SP
73 AM=M-1
74 D=M
75 A=A-1
76 M=*M+D
```

Slika 4. Primjer „floatadd“ naredba koja zbraja dva broja s pomičnim zarezom

Razlika između operacija postoji zbog prevođenja u asemblerski kod te kasnije u strojni jezik jer naredbe za cijele brojeve obavlja ALU, a za brojeve s pomičnim zarezom FPU pa moramo biti u mogućnosti znati gdje „poslati“ naredbu. Sama razlika je u zbrajanju, oduzimanju, množenju, veće od i manje od, dok su ostale operacije izuzev dijeljenja jednake jer dijeljenje za brojeve s pomičnim zarezom nije definirano. Operacije nad različitim tipovima podataka nisu moguće.

5. Programski jezik Jack

5.1 Struktura programa

Programski jezik Jack koji nastaje u Nand2Tetris projektu je objektno orijentirani jezik visoke razine sa relativno jednostavnom i intuitivnom sintaksom sa nekoliko ugrađenih klasa.

Sintaksa je slična mnogim objektno orijentiranim jezicima poput C++, C#, Jave itd., stoga je bitno naglasiti da unutar jednog dokumenta sve mora biti u jednoj klasi kao i u navedenim jezicima. Ta klasa se mora zvati identično kao i dokument te se sav kod treba nalaziti unutar vitičastih zagrada. Generalno, program napisan u Jacku možemo raščlaniti na niz tokena sa proizvoljnim brojem razmaka i komentara. Razlikujemo četiri vrste tokena: „symbols“, „reserved words“, „constants“ i „identifiers“. Pod „symbols„ smatramo sve vrste zagrada, interpunkcijske znakove te znakove računskih operacija. „reserved words“ su podijeljene u šest kategorija: „program components“ (class, constructor, method, function), „primitive types“ (int, boolean, char, void), „variable declarations“(var, static, field) , „statements“ (let, do, if, else, while, return), „constant values“(true,false, null) i „object reference“ (this). „constants“ su pozitivni cijeli i decimalni brojevi, dok negativne brojeve promatramo kao izraz simbola „-“ i pozitivne konstante. Također postoje boolean konstante true i false, string konstante i „null“ konstanta. „identifiers“ su nazivi funkcija, varijabli i sl. koji se mogu sastojati od malih i veliki slova engleske abecede, brojeva i znaka „_“ te ne smiju započinjati brojkom.

5.2 Varijable

U jednom dokumentu, odnosno klasi mogu biti metode, funkcije, konstruktori, varijable itd. Kod tipova varijabli razlikujemo primitivne tipove podataka, klase te dodatno „float“. Također razlikujemo „static“, „var“ i „field“ varijable gdje „static“ varijable označavaju statične varijable, „var“ lokalne, a „field“ varijable koje su dostupne samo objektu, tj.instanci neke klase. Također imamo „this“ i „that“ kao referencu na objekt klase te referencu na polje koje trenutno koristimo. Konverzija tipova ovisi o samome kompajleru, ali konverzija tipova na Slici 4., Slici 5. i Slici 6. mora biti dopuštena u svakom kompajleru.

```
1  var char c;
2  var String s;
3  let c = 33; // 'A'
4  // Ekvivalentni zapis
5  let s = "A";
6  let c = s.charAt(0);
```

Slika 5. Prikaz dva zapisa znaka koristeći njegovu ASCII vrijednost

```
1  var Array a;
2  let a = 2000;
3  let a[100] = 13; // Memorijska adresa 2000 je postavljena na 13
```

Slika 6. Prikaz postavljanja memorijske adrese na konstantu 13

```
1  var Complex c;
2  var Array a;
3  let a = Array.new(2);
4  let a[0] = 2;
5  let a[1] = 4;
6  let c = a; // c==Complex(2,4)
```

Slika 7. Prikaz postavljanja vrijednosti koristeći indeksiranje polja

5.3 Funkcije

Također bitno je razlikovati funkciju i metodu. Funkcije imaju ulogu statičnih metoda dok su metode funkcije koje se referiraju na objekt neke klase. Tipovi funkcija su kao i u svakom programskom jeziku su void, int, char, float te klasa koje imamo implementirane. Dodatno postoje tipovi string i array koji se nalaze unutar istoimenih klasa koje su ugrađene u sustav. Ostale funkcije ili metode koje pozivamo unutar neke funkcije pozivamo s narednom „do“ te pozivamo je zbog njezinog učinka na varijable, objekte, polja itd. Osim funkcija i metoda, od ostalih „statements“ potrebno je spomenuti naredbu let kojom postavljamo vrijednost neke varijable. Naredba „if“ koji predstavlja klasično if grananje, no bitno je da se naredbe koje se nalaze unutar „if i „else“ dijela su unutar vitičastih zagrada te while kod koje su također obavezne zagrade. I naredba „return“ koja je obavezna u svakoj funkciji te se pojavljuje u dva oblika : „return“ i „return izraz“ gdje izraz mora biti tipa kojeg je dana funkcija. Također svaki konstruktor mora vraćati objekt „this“. I slično kao i C++ svaka naredba unutar funkcije mora završavati znakom „ ; “.

```
1  class Main
2  {
3      function void main()
4      {
5          do Output.printString("Hello world!");
6          do Output.println();
7          return;
8      }
9  }
```

Slika 8. „Hello World“ program napisan u programskom jeziku Jack

Na slici 8. vidimo klasični „Hello World“ program koji prikazuje osnovna svojstva jezika Jack, tj. vidimo kako treba izgledati sintaksa svake funkcije te primjer korištenja već spomenutih ugrađenih klasa koje se nalaze unutar operativnog sustava, u ovom slučaju klase „Output“ te dviju funkcija koje ispisuju izraz „Hello World!“ i novi red.

Razlikujemo tri vrste komentara u Jacku, komentari koji započinju znakom „//“ nazivamo jednolinijski komentari i komentar se piše do kraja reda, zatim „/*“ komentar je višelinijski te završava kada se zatvori znakom „*/“ te „/**“ su komentari koji sadrže API dokumentaciju te završavaju jednako kao višelinijski komentari.

Kada je riječ o obavljanju operacija poput zbrajanja, oduzimanja, množenja, dijeljenja, veće od, manje od te logičnih operacija poput negacije, logičko i te logičko ili postoje neka pravila. Redoslijed operacija unutar jedne naredbe se odvija po redu, odnosno ne vrijede pravila o redoslijedu računskih operacija, već se operacije odvijaju s lijeva na desno. Kako bi se uspostavio uvriježeni prioritet računskih operacija nužne su zagrade.. Kod operacije jednako bitno je znati da operator „=“ ovisno o kontekstu označava i izjednačavanje, ali i pridruživanje. Također operator koji u ostalim programskim jezicima postoji kao „!=“ se zapisuje tako da prvo zapišemo izraz izjednačavanja u zagradi, a zatim ga negiramo. Generalno, negacija se zapisuje pomoću znaka tilde.

6. Kompajler

Kompajler je računalni program koji prevodi računalni kod napisan u nekom programskom jeziku u drugi jezik. Uobičajeno, ova transformacija uključuje prevođenje koda napisanog na višem programskom jeziku u jezik niže razine, kao što je asembler ili strojni jezik. Postoje različite vrste kompajlera te neki od njih mogu pripadati u više kategorija ovisno o svojim funkcionalnostima i karakteristikama.

Dvije glavne kategorije su „Single-Pass“ i „Multi-pass“ kompajleri koji kako im i naziv kaže prolaze kroz kod samo jednom, odnosno više puta. „Single-pass“ kompajleri su jednostavniji, ali nisu uvijek optimalni, dok „Multi-Pass“ čitaju kod više puta kako bi optimizacija koda bila što bolje, ali su zato kompleksniji. Također, postoje „Cross“ kompajleri koji generiraju kod za različite platforme od one na kojoj rade. Oni su korisni za razvoj aplikacija za različite sustave. Nasuprot tome, „Native“ kompajleri generiraju kod za istu platformu na kojoj rade i koriste se za razvoj općenitih aplikacija.

Kompajler za jezik Jack je „Single-Pass“ kompajler koji radi na principu leksičke i semantičke analize, parsiranja i generiranja koda. Jack program se sastoji od niza tokena razdvojenih proizvoljnim brojem razmaka i komentara te tokeni mogu biti simboli, posebne riječi, konstante i identifikatori. Svaki tip tokena se posebno prevodi te postoje određena pravila kod prevođenja.

6.1 Leksička analiza

Prva faza kompajliranja, tj. prevođenja koda jest leksička analiza koja obuhvaća prolazak kroz cijeli program te ga raščlanjuje na različite vrste tokena. Program se raščlanjuje na najmanje smislene jedinice, poput simbola, operatora, interpunkcijskih znakova, identifikatora, ključnih riječi i sl. Kao što je vidljivo na slici jednostavan primjer „while“ petlje je tokeniziran na određene vrste tokena te za svaki od njih postoje određena pravila prevođenja.

```
1  ...
2  while (count <= 100)
3  {
4  count++;
5  }
6  ...
```

```
1  ...
2  while
3  (
4  count
5  <=
6  100
7  )
8  {
9  count
10 ++
11 ;
12 }
13 ...
```

Slika 9. Primjer leksičke analize

6.2. Semantička analiza

Sljedeći korak jest semantička analiza čija su pravila prikazana na Slici 10. koja određene vrste tokena svrstava u određene kategorije. Te kategorije su: „lexical elements“, „program structure“, „statements“ i „expressions“. Dodatno, doradom projekta kako bi programski jezik Jack prihvaćao i decimalne brojeve, u kategoriji „lexical elements“, kod „keywords“ možemo pronaći „float“, u kategoriji „program structure“, „type“ je također „float“ i kod „constants“ postoji „floatConstant“.

Lexical elements:	The Jack language includes five types of terminal elements (tokens):
keyword:	**'class'** | **'constructor'** | **'function'** | **'method'** | **'field'** | **'static'** | **'var'** | **'int'** | **'char'** | **'boolean'** | **'void'** | **'true'** | **'false'** | **'null'** | **'this'** | **'let'** | **'do'** | **'if'** | **'else'** | **'while'** | **'return'**
symbol:	'{' | '}' | '(' | ')' | '[' | ']' | '.' | ',' | ';' | '+' | '-' | '*' | '/' | '&' | '|' | '<' | '>' | '=' | '~'
integerConstant:	A decimal number in the range 0 .. 32767.
StringConstant	'"' A sequence of Unicode characters not including double quote or newline '"'
identifier:	A sequence of letters, digits, and underscore ('_') not starting with a digit.
Program structure:	A Jack program is a collection of classes, each appearing in a separate file. The compilation unit is a class. A class is a sequence of tokens structured according to the following context free syntax:
class:	**'class'** className '{' classVarDec* subroutineDec* '}'
classVarDec:	(**'static'** | **'field'**) type varName (',' varName)* ';'
type:	**'int'** | **'char'** | **'boolean'** | className
subroutineDec:	(**'constructor'** | **'function'** | **'method'**) (**'void'** | type) subroutineName '(' parameterList ')' subroutineBody
parameterList:	((type varName) (',' type varName)*)?
subroutineBody:	'{' varDec* statements '}'
varDec:	**'var'** type varName (',' varName)* ';'
className:	identifier
subroutineName:	identifier
varName:	identifier
Statements:	
statements:	statement*
statement:	letStatement | ifStatement | whileStatement | doStatement | returnStatement
letStatement:	**'let'** varName ('[' expression ']')? '=' expression ';'
ifStatement:	**'if'** '(' expression ')' '{' statements '}' (**'else'** '{' statements '}')?
whileStatement:	**'while'** '(' expression ')' '{' statements '}'
doStatement:	**'do'** subroutineCall ';'
ReturnStatement	**'return'** expression? ';'
Expressions:	
expression:	term (op term)*
term:	integerConstant | stringConstant | keywordConstant | varName | varName '[' expression ']' | subroutineCall | '(' expression ')' | unaryOp term
subroutineCall:	subroutineName '(' expressionList ')' | (className | varName) '.' subroutineName '(' expressionList ')'
expressionList:	(expression (',' expression)*)?
op:	'+' | '-' | '*' | '/' | '&' | '|' | '<' | '>' | '='
unaryOp:	'-' | '~'
KeywordConstant:	**'true'** | **'false'** | **'null'** | **'this'**

Slika 10. Gramatika jezika Jack

Provjera odgovara li, točnije „prihvaća“ li gramatika ulaz naziva se parsiranje. Provjerava se je li ulaz validan, gdje se nalazi u gramatici te koja pravila vrijede za njega. Postoje različiti algoritmi parsiranja, ali najčešće se provodi rekurzivno parsiranje čiji rezultat je binarno stablo tokena. Na primjeru Slike 9. vidljiva je izjava i to while izjava. Sintaksa while izjave je ključna riječ while pa slijedi izraz u zagradama te opet izjava. Izraz je u ovom slučaju „count <= 100“, a izjava koja ide nakon je podijeljena u niz izjava gdje se nalazi izjava „count++“,

zatim slijedi znak „;“ te opet niz izjava koje bi slijedile u programu. Sljedeća faza je generiranje koda o kojoj ćemo čuti u sljedećem podnaslovu.

6.3 Generiranje koda

Zadnja faza kompajliranja je generiranje koda, ali prije toga reći ćemo nešto o strukturi kompajlera kako bi nam bilo jasno gdje se koji dio prevodi. Kompajler se sastoji od nekoliko pomoćnih dokumenta, a oni su: „JackCompiler“ koji upravlja ostalim dijelovima kompajlera, „JackTokenizer“ koji obavlja raščlambu programa na tokene, „SymbolTable“ gdje se nalazi tablica simbola, „VMWriter“ koji generira VM kod te „CompilationEngine“.

Prilikom generiranja koda koji je u našem slučaju „.vm“ dokument, bit će nam bitni tipovi varijabli i njihove vrste, tj. “kind“ varijabli jer se svaka varijabla mora mapirati u ekvivalentan zapis u jeziku virtualnog stroja. Kada govorimo o prevođenju identifikatora prvo smo morali kreirati „SymbolTable“. „SymbolTable“ sadrži funkcije kojima je moguće kreirati i koristiti se tablicom simbola. Unutar tablice simbola za svaku vrstu podatka postoji određeni cijeli broj koji ga reprezentira unutar njegovog opsega počevši od 0 te povećavajući se za 1 za svaki novi simboli. Taj broj predstavlja indeks koji se resetira svaki put kada je u pitanju novi „scope“, odnosno opseg varijable. Razlikujemo četiri vrste varijabli, statične i member varijable čiji su opseg cijela klasa, argument i lokalne varijable čiji su opseg neki „potprogram“, tj. neka metoda, funkcija ili konstruktor. Generalno u tablici simbola se nalaze svi identifikatori, njihovi tipovi i vrste te trenutni indeks. Za izraze („expressions“) generiranje koda ovisi o načinu parsiranja. U našem slučaju kada prevodimo u jezik virtualnog stroja, parsiranje je rekurzivno te je rezultat istoga binarno stablo. Tada ovisno o kakvom je izrazu riječ vrijede određena pravila. Prvo se na stog postavljaju varijable, a zatim se pozivaju funkcije i obavljaju operacije s lijeva na desno. Za prevođenje if, while i sl. dijelova programa bitno je znati da u jeziku virtualnog stroja postoje samo „goto“, odnosno bezuvjetni „skok“ te „if-goto“, odnosno uvjetni „skok“ na određeni dio koda. Zbog toga if petlje se prevode koristeći „if-goto“ i „goto“ naredbe, a za while petlje koristi se višestruko generiranje i korištenje jedinstvenih naziva oznaka.

Osim prepoznavanja i parsiranja brojeva s pomičnim zarezom, bilo je bitno i parsiranje identifikatora. Kompajler je u svakome trenutku morao znati radi li sa „int“ identifikatorom ili „float“ identifikatorom što je postignuto sa pomoćnom zastavicom koja je prilikom parsiranja identifikatora postavljena na 0 ukoliko je tip identifikatora „int“, a u slučaju „float“. Za parsiranje samih brojeva s pomičnim zarezom korištena je nova funkcija koja je gledala „string“ verziju broja ili izraza, u slučaju računskih operacija, te provjeravala radi li se validnom broju s pomičnim zarezom ili o validnoj računskoj operaciji nad dva broja s pomičnim zarezom.

Kod generiranja koda za računske operacije tu smo morali znati nad kojim tipovima varijabli obavljamo određene operacije jer postoji razlika u operacijama nad cijelim brojevima i brojevima s pomičnim zarezom u jeziku virtualnog stroja. Također operacije nad različitim tipovima podataka nisu moguće. No, problem razlike između operacija nad cijelim brojevima i brojevima s pomičnim zarezom riješen je na način vidljiv na Slici 11. Prilikom parsiranja bitan nam je operator i tip podatka s kojim radimo. Tip podatka dobivamo koristeći već spomenuti „JackTokenizer“, te dodatno kada je u pitanju identifikator pazimo radi li se o „int“ ili „float“ identifikatoru. S tim prikupljenim podatcima kompajler može prepoznati o kojoj računskoj ili logičkoj operaciji je riječ te zatim poziva pripadnu funkciju iz „VMWriter“ dokumenta.

```
750
751             if operator == '+' and (tokenType == 4 or self.flag == 0 or tokenType == 2):
752                 self.vmWriter.WriteArithmetic(OP_ADD)
753             elif operator == '+' and (tokenType == 6 or self.flag == 1 or tokenType == 2):
754                 self.vmWriter.WriteArithmetic(OP_FLOATADD)
755             elif operator == '-' and (tokenType == 4 or self.flag == 0 or tokenType == 2):
756                 self.vmWriter.WriteArithmetic(OP_SUB)
757             elif operator == '-' and (tokenType == 6 or self.flag == 1 or tokenType == 2):
758                 self.vmWriter.WriteArithmetic(OP_FLOATSUB)
759             elif operator == '*':
760                 self.vmWriter.WriteCall('Math.multiply', 2)
761             elif operator == '/':
762                 self.vmWriter.WriteCall('Math.divide', 2)
763             elif operator == '&':
764                 self.vmWriter.WriteArithmetic(OP_AND)
765             elif operator == '|':
766                 self.vmWriter.WriteArithmetic(OP_OR)
767             elif operator == '<' and (tokenType == 4 or tokenType == 2 or self.flag == 0):
768                 self.vmWriter.WriteArithmetic(OP_LT)
769             elif operator == '>' and (tokenType == 4 or tokenType == 2 or self.flag == 0):
770                 self.vmWriter.WriteArithmetic(OP_GT)
771             elif operator == '<' and  (tokenType == 6 or tokenType == 2 or self.flag == 1) :
772                 self.vmWriter.WriteArithmetic(OP_FLOATLT)
773             elif operator == '>' and (tokenType == 6 or tokenType == 2 or self.flag == 1):
774                  self.vmWriter.WriteArithmetic(OP_FLOATGT)
775             elif operator == '=':
776                 self.vmWriter.WriteArithmetic(OP_EQ)
```

Slika 11. Prikaz Python koda u „JackCompile“ dokumentu za parsiranje operacija

7. Zaključak

Svaki aspekt izrade ovog projekta pokazao nam je na relativno jednostavan način kako funkcionira 16-bitno računalo. Preciznije, kako od jednostavnih logičkih komponenti može nastati funkcionalna hardverska platforma koja može izvršavati osnovne aritmetičke operacije nad cijelim te dodatno brojevima s pomičnim zarezom. Posebice kako je moguće implementirati izvođenje aritmetičkih operacija nad brojevima s pomičnim zarezom koristeći IEEE-754 Standard, tj. moguć je zapis i manipulacija predznaka, eksponenta i mantise. Nadalje kako funkcionira asembler, kako pomoću znaka „*“ razlikujemo asemblerske naredbe koje nakon prevođenja obavlja ALU, a koje FPU. Također kako virtualni stroj koristi apstrakciju stroga te sve naredbe obavlja koristeći isti te kako je opet vidljiva razlika između naredbi za cijele brojeve te za brojeve s pomičnim zarezom za što se brine parser. I konačno integracija brojeva s pomičnim zarezom u programski jezik Jack, mogućnost obavljanja aritmetičkih operacija zbrajanja, oduzimanja, množenja te logičkih „and“ i „or“. Izradom kompajlera za jezik Jack upoznali smo se sa radom kompajlera, fazama kompajliranja, raščlambom samog koda na logičke cjeline, izradom gramatike, pojmom parsiranja te vrstama parsiranja te na kraju generiranjem koda u ovom slučaju u jeziku virtualnog stroja. Također je opet primjetna razlika između dvije vrste brojeva s kojima radimo, iako u jeziku Jack ne postoji određene naredbe za jedne, odnosno druge već ih sam kompajler prepoznaje te prevodi kasnije. Kompajler je dodatno morao znati prepoznati brojeve s pomičnim zarezom, razlikovati vrste „identifiers“, tj. ukoliko imamo ime neke varijable, znati njezin tip u svakom trenutku te znati prepoznati operacije koje smije obavljati nad brojevima s pomičnim zarezom. U konačnici smo dobili funkcionalno 16-bitno računalo koje može raditi sa svim tipovima podataka s ograničenom preciznošću, no shvatili smo koncept rada jednog računala te dodali dodatan tip podatka kako bi računalo bilo još funkcionalnije.

8. Literatura

[1] Nisan, N. & Schocken S., (2005) , *The elements of computing systems*, Massachusetts: Massachusetts Institute of Technology

[2] IEEE Computer Society (2019), *IEEE Standard for Floating-Point Arithmetic*, New York: IEEE Computer Society

[3] Yadav A. ,(2017), *International Journal of Engineering Research and Applications*, New Delhi: IJERA

[4] Jack Reference : https://classes.engineering.wustl.edu/cse365/jack.php

[5] Nand2Tetris : https://www.nand2tetris.org/