Harverdska implementacija floating-point jedinice za hack računalo

Orlić, Robert

Undergraduate thesis / Završni rad

2024

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **Josip Juraj Strossmayer University of Osijek, School of Applied Mathematics and Informatics / Sveučilište Josipa Jurja Strossmayera u Osijeku, Fakultet primijenjene matematike i informatike**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:126:578305

Rights / Prava: In copyright / Zaštićeno autorskim pravom.

Download date / Datum preuzimanja: **2024-10-06**

Repository / Repozitorij:

Repository of School of Applied Mathematics and Computer Science

Sveučilište J. J. Strossmayera u Osijeku

Fakultet primijenjene matematike i informatike

Sveučilišni preddiplomski studij Matematika i računarstvo

Hardverska implementacija Floating-Point jedinice za Hack računalo

Završni rad

Mentor:

izv. prof. dr. sc. Domagoj Matijević

Komentor: Kandidat:

dr. sc. Luka Borozan **Robert Orlić**

Osijek 2024.

Sažetak

U ovom radu proći ćemo kroz hardversku implementaciju 16-bitne floating-point jedinice za Hack računalo. Upoznat ćemo se s osnovama HDL-a, detaljno opisati hardversku implementaciju floating-point jedinice te se naposljetku dotaknuti softverskog dijela implementacije i ukratko ga opisati.

Ključne riječi

float, hardver, FPU, HDL, strojni jezik, asembler, Hack, nand2tetris, Jack

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

Summary

In this paper we will go through the hardware implementation of a 16-bit floating-point unit for a Hack computer. We will get to know the basics of HDL, thoroughly describe the hardware implementation of the floating point-unit and eventually say something about the software implementation and describe it briefly.

Keywords

float, hardware, FPU, HDL, machine language, assembley, Hack, nand2tetris, Jack

Sadržaj

1. Uvod

U sklopu kolegija Moderni računalni sustavi, slušanog u prvom semestru druge godine preddiplomskog studija Matematika i računarstvo izradili smo 16-bitno računalo pod nazivom HACK. Proces izrade ovog računala započeo je opisivanjem njegovih hardverskih komponenti u HDL-u, nastavio se implementiranjem parsera za asembler i jezik virtualnog stroja te završio implementacijom kompajlera za objektno orijentirani jezik Jack. Kroz izradu tog računala detaljno smo se upoznali s njegovom građom i načinom rada na svim razinama, što je bilo iznimno korisno jer se to znanje lako primjenjuje na moderne 32 i 64-bitne sustave. Tom računalu nedostajala je samo Floating-point jedinica koja mu omogućuje rad s brojevima s pomičnim zarezom pa je cilj ovog projekta bio dizajn i implementacija 16-bitne floating-point jedinice za to računalo.

U ovom radu upoznat ćemo se sa standardom za zapisivanje brojeva s pomičnim zarezom, floating-point aritmetikom, objasniti građu Hack računala i detaljno opisati dizajn i implementaciju floating-point jedinice te na kraju ukratko opisati softverski dio implementacije.

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

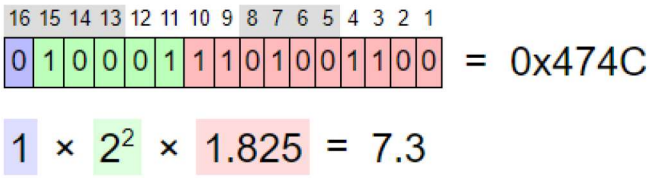
2. IEEE 754 Standard

IEEE 754 standard je način za računalno zapisivanje racionalnih brojeva nastao 1985. godine. Standard obuhvaća nekoliko različitih formata: 64-bitni format odnosno „double-precision“, 32-bitni format odnosno „single-precision“ te 16-bitni format odnosno „half-precision“. Kako u radu govorimo o 16-bitnom računalu taj ćemo format i objasniti.

Po IEEE 754 standardu prikaz 16-bitnog broja s pomičnim zarezom, slijeva na desno izgleda ovako: 1 bit za predznak (0 za pozitivan, a 1 za negativan broj), 5 bitova za eksponent (kada tih 5 bitova prebacimo u bazu 10 od dobivenog broja oduzmemo 15 kako bi dobili pravi eksponent), 1 skriveni bit mantise (on se nigdje ne sprema, ali se koristi u aritmetičkim operacijama, 0 ukoliko je eksponent „00000“, 1 inače) te 10 bitova mantise.

U ovom standardu razlikujemo dvije vrste brojeva i nekoliko specijalnih vrijednosti. Brojevi se dijele na subnormalne (skriveni bit 0) i normalne (skriveni bit 1), a aritmetičke operacije se drugačije izvršavaju u ovisnosti o vrsti broja. U slučaju kada je eksponent maksimalan („11111“) i mantisa 0 broj se čita kao beskonačno. Zbog načina određivanja predznaka u standardu razlikujemo pozitivnu i negativnu beskonačnost, ali i pozitivnu i negativnu nulu. Kada je eksponent maksimalan a mantisa različita od 0 broj se čita kao „NaN“ vrijednost odnosno „nije broj“.

U idućem poglavlju ćemo govoriti o floating-point aritmetici definiranoj u ovom standardu koja nam je bila potrebna za dizajn i implementaciju floating-point jedinice za Hack računalo.



Slika 1. Primjer zapisa broja 7.3 koristeći IEEE 754 Standard

3. Floating-point aritmetika

U IEEE 754 standardu detaljno su definirane i objašnjene aritmetičke operacije nad brojevima sa pomičnim zarezom. Kako bi mogli dizajnirati i implementirati floating-point jedinicu za Hack računalo potrebno ih je poznavati i u potpunosti razumjeti. Za implementaciju su nam bile potrebne operacije zbrajanja, oduzimanja i množenja.

3.1 Zbrajanje i oduzimanje

Operacije zbrajanje i oduzimanja se u ovom standardu obavljaju na vrlo sličan način. Prvo odredimo veći od dva broja po apsolutnoj vrijednosti. Ukoliko se eksponenti brojeva razlikuju, mantisu broja s manjim eksponentom pomičemo u desno onoliko puta koliko se eksponenti razlikuju. Tako dobivene mantise zbrojimo (uključujući i skriveni bit), a eksponent normaliziramo ukoliko je potrebno. Normalizacija eksponenta provodi se ukoliko smo prilikom zbrajanja mantisa s lijeve strane decimalne točke dobili više od jednog bita, ukoliko se to dogodilo decimalnu točku pomičemo u lijevo dok ne dobijemo jedan bit s lijeve strane, a eksponent povećamo za onoliko koliko smo puta pomaknuli decimalnu točku. Oduzimanje se obavlja na gotovo identičan način, jedina razlika je što se mantise oduzimaju umjesto zbrajanja. Konačni rezultat dobijemo kombinacijom predznaka većeg broja po apsolutnoj vrijednosti, normaliziranog eksponenta i rezultat zbrajanja odnosno oduzimanja mantisa (bez skrivenog bita). Primjerice uzmemo li broj 2.5 koji zapisan u IEEE 754 standardu izgleda ovako: „0100000100000000“ i zbrojimo ga sa samim sobom provedba algoritma zbrajanja izgledala bi ovako:

1. Provjerimo razlikuju li se eksponenti broja, kako se radi o istim brojevima oni se ne razlikuju (oba su „10000“) pa ne moramo raditi nikakve prilagodbe
2. Zbrojimo mantise, koje čine posljednjih 10 znamenki svakog broja plus skriveni bit, u ovom slučaju „1.0100000000“ i „1.0100000000“. Rezultat zbrajanja mantisa je „10.1000000000“. Kako s lijeve strane decimalne točke smijemo imati samo jedan bit decimalnu točku pomičemo jedno mjesto u lijevo te dobivamo konačnu mantisu „1.0100000000“.
3. Provjerimo treba li normalizirati eksponent, kako smo u prošlom koraku pomicali mantisu u lijevo to znači da ga moramo normalizirati odnosno povećati ga za onoliko koliko smo puta pomakli mantisu. Originalnom eksponentu „10000“ dodajemo "00001" te dobivamo konačni eksponent „10001“.
4. Spajamo predznak većeg od 2 broja po apsolutnoj vrijednosti, u ovom slučaju su oba pozitivna pa je to „0“, eksponent iz 3. koraka i 10 znamenki mantise koje se nalaze desno od decimalne točke u 2. koraku. Tim postupkom dobivamo konačni rezultat „0100010100000000“, odnosno broj 5 zapisan u IEEE 754 standardu.

3.2 Množenje

Kod operacije množenja nije nam bitno koji broj je veći. Prvo odaberemo predznak na temelju „XOR“ tablice istinitosti (isti predznaci daju 0, različiti 1), zatim zbrojimo eksponente i od njih oduzmemo „bias“ (u slučaju 16-bitnog zapisa to je 15) nakon toga pomnožimo mantise uključujući i skriveni bit te naposljetku rezultat normaliziramo ukoliko je to potrebno.

A	B	Q
0	0	0
0	1	1
1	0	1
1	1	0

Slika 2. XOR tablica istinitosti

Ukoliko primjerice uzmemo brojeve 2.5 i -2.5 odnosno „0100000100000000“ i „1100000100000000“ te ih pomnožimo provedba algoritma množenja bi izgledala ovako:

1. Odabir predznaka koristeći XOR tablicu istinitosti, predznaci su „0“ i „1“ pa odabiremo „1“.
2. Zbrojimo eksponente i oduzmemo bias, zbrajanjem „10000“ i „10000“ dobijemo „100000“ i oduzimanjem od toga „01111“ dobijemo „10001“.
3. Pomnožimo mantise, množenjem „1.0100000000“ i „1.0100000000“ dobijemo „1.1001000000“. Lijevo od decimalne točke imamo samo jedan bit pa rezultat ne moramo normalizirati.
4. Provjerimo je li potrebna normalizacija eksponenta, kako u prošlom koraku nije bilo potrebe za normalizacijom tako ni eksponent nije potrebno normalizirati.
5. Spojimo sve komponente kako bi dobili traženi rezultat, „1100011001000000“, odnosno 6.25 zapisan u IEEE 754 standardu.

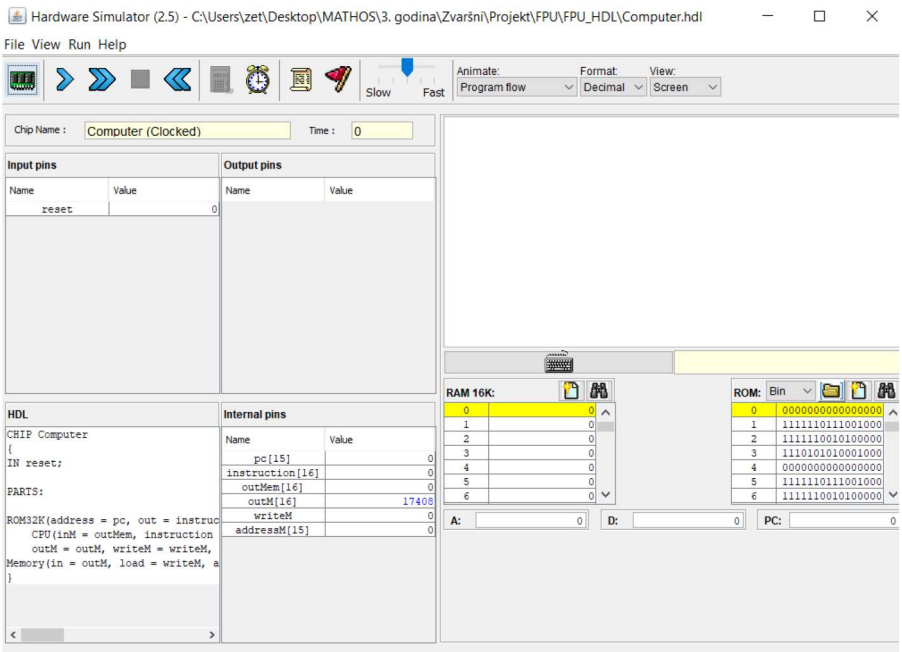
U idućem poglavlju upoznat ćemo se s HDL-om i detaljno objasniti hardversku implementaciju same floating-point jedinice.

4. Floating-point jedinica

Hack računalo izrađeno na kolegiju Moderni računalni sustavi, a prateći „Nand2Tetris“ projekt sastoji se od memorije (RAM i ROM) i CPU-a unutar kojeg se nalazi aritmetičko logička jedinica (ALU). Aritmetičko logička jedinica omogućuje izvršavanje aritmetičkih i logičkih operacija nad cijelim brojevima zapisanim u binarnom zapisu. Kako bi rad Hack računala proširili na brojeve s pomičnim zarezom bilo je potrebno dizajnirati i implementirati floating-point jedinicu (FPU), po uzoru na aritmetičke operacije opisane u prethodnom poglavlju.

4.1 HDL

Cjelokupna hardverska implementacija Hack računala odrađena je u HDL-u. HDL (Hardware decription language) je specijalizirani jezik koji se koristi za opisivanje strukture i ponašanja elektroničkih čipova i logičkih vrata. Cijelo Hack računalo izgrađeno je od velikog broja čipova implementiranih u HDL-u. Važan dio u implementaciji ovih čipova je i njihovo testiranje koje se provodi u specijaliziranom softveru naziva „Hardware simulator“ dostupnom u [6]. Sam simulator prikazuje RAM i ROM, HDL kod pokrenutog čipa, unutarnje vrijednosti čipa, ulazne i izlazne podatke te ekran računala. Simulator također omogućuje pokretanje HACK datoteka koje sadrže instrukcije za računalo zapisane u strojnom jeziku.



Slika 3. Primjer pokretanja Hack računala u Hardware simulatoru

4.2. Implementacija

Sama implementacija može se podijeliti na tri glavna djela, prvi dio je implementacija same floating-point jedinice i njene generalne logike, drugi dio predstavljao bi implementaciju čipa

koji odrađuje operacije zbrajanja i oduzimanje, a treći obuhvaća implementaciju čipa koji odrađuje operaciju množenja. Prvo smo dizajnirali i implementirali FPU po uzoru na ALU. FPU mora imati mogućnost vračanja nekoliko različitih rezultata, ako s X i Y označimo 16-bitne ulaze koji predstavljaju brojeve s pomičnim zarezom FPU treba biti u mogućnosti vratiti sljedeće rezultate na izlazu: X, Y, X+Y, X-Y, Y-X, 0, -X, -Y i X*Y. Osim ulaza X i Y FPU prima 6 bitova na ulazu koje nazivamo „zastavice“, o njihovoj kombinaciji ovisi koja će se operacija izvršiti i koji će rezultat biti na izlazu FPU.

Prve dvije zastavice određuju hoće li se ulazi X i Y postaviti na nula, stoga ih prikladno nazivamo „zx“ i „zy“ (ako je zastavica 1 postavljamo ulaz na 0, inače ne radimo ništa). Zatim provjeravamo trebaju li se ulazi negirati pomoću prikladno nazvanih zastavica „nx“ i „ny“ (ako je zastavica 1 negiramo ulaz, inače ne radimo ništa). Nakon što smo dobili potrebne ulaze na FPU se izvršavaju i operacija zbrajanja/oduzimanja i operacija množenja, nakon što se izvrše obje operacije na temelju pete zastavice „f“ određujemo koji od ta dva rezultata uzimamo (zbrajanje/oduzimanje ako je zastavica1, množenje inače). Operacije zbrajanja/oduzimanja odvijaju se na čipu „FloatAdd“, a operacija množenja na čipu „MultHalf“ čije ćemo implementacije opisati kasnije. Nakon odabira rezultata između te dvije operacije provjeravamo posljednju zastavicu „no“ kako bi negirali izlaz ukoliko je to potrebno (ukoliko je zastavica 1 negiramo izlaz).

	zx	zy	nx	ny	f	no
X	0	1	0	0	1	0
Y	1	0	0	0	1	0
(-X)	0	1	1	0	1	0
(-Y)	1	0	0	1	1	0
0	1	1	0	0	1	0
X+Y	0	0	0	0	1	0
X-Y	0	0	0	1	1	0
Y-X	0	0	1	0	1	0
X*Y	0	0	0	0	0	0

Slika 4. Prikaz kombinacija zastavica i pripadnih operacija koje će se izvršiti

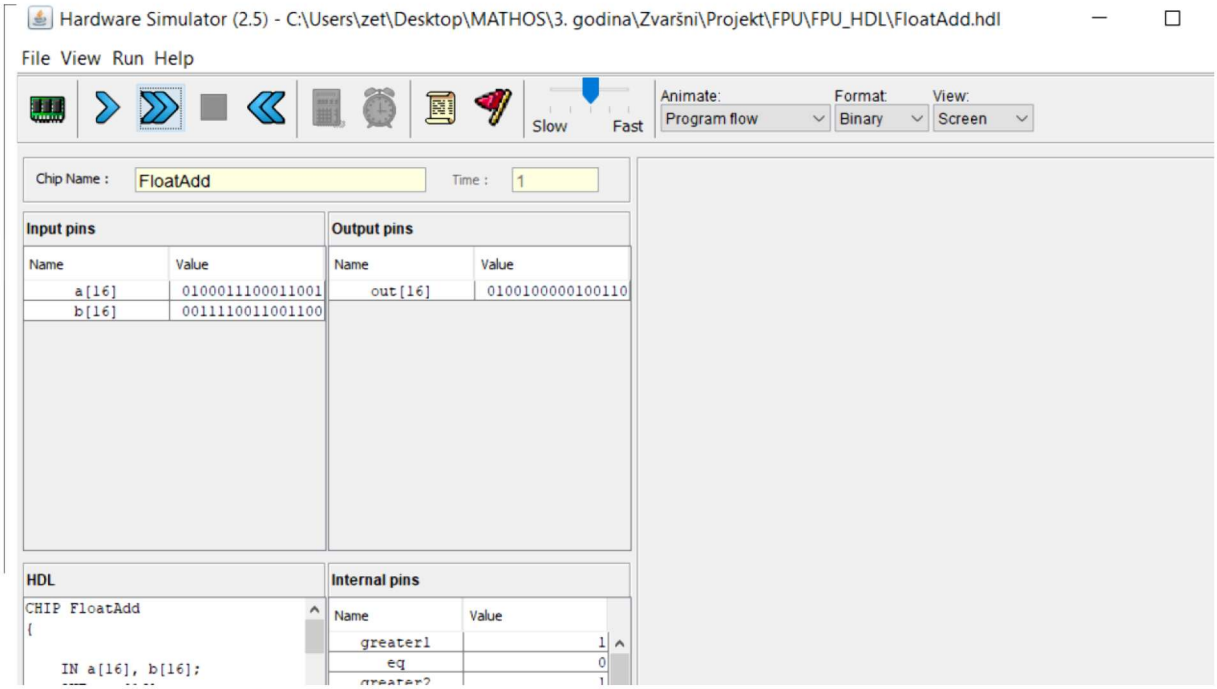
```
CHIP FPU
{
    IN
        x[16], y[16],   // 16 bitni ulazi
        zx,             // postavljanje prvog ulaza na 0?
        nx,             // negiranje prvog ulaza (promjena sign bita)?
        zy,             // postavljanje drugog ulaza na 0?
        ny,             // negiranje drugog ulaza (promjena sign bita)?
        f,              // računanje izlaza, x+y ako je f=1, x*y ako je f=0
        no;             // negiranje izlaza

    OUT
        out[16],        // 16 bitni izlaz
        zr,             // 1 ako je izlaz jednak 0, inače 0
        ng;             // 1 ako je izlaz manji od 0, inače 0 -

    PARTS:

    // Postavljanje ulaza na 0
    Mux16(a=x, b=false, sel=zx, out=zerox);
    Mux16(a=y, b=false, sel=zy, out=zeroy);

    // Negiranje ulaza
    NotFloat(a=zerox, out=notx);
    NotFloat(a=zeroy, out=noty);

    // Odabir varijabli
    Mux16(a = zerox, b = notx, sel = nx, out = finalx);
    Mux16(a = zeroy, b = noty, sel = ny, out = finaly);

    // Zbrajanje odnosno oduzimanje i množenje
    FloatAdd(a=finalx, b=finaly, out=addxy);
    MultHalf(a = finalx, b = finaly, out = multxy);

    // Odabir operacije.
    Mux16(a = multxy, b = addxy, sel = f, out = op);

    // Negacija operacije.
    NotFloat(a = op, out = notop);

    // Odabir originalne ili negirane operacije.
    Mux16(a = op, b = notop, sel = no, out[0..7] = res1, out[8..14] = res2, out[15] = res3);

    // Je li rezultat jednak nuli?
    Or8Way(in = res1, out = or1);
    Or8Way(in[0..6] = res2, in[7] = res3, out = or2);
    Or(a = or1, b = or2, out = or);

    Or16(a[0..7] = res1, a[8..14] = res2, a[15] = res3, b = false, out = out);
    Not(in = or, out = zr);
    Or(a = res3, b = false, out = ng);
}
```

Slika 5. Prikaz implementacije floating-point jedinice u HDL-u

4.2.1 Zbrajanje i oduzimanje

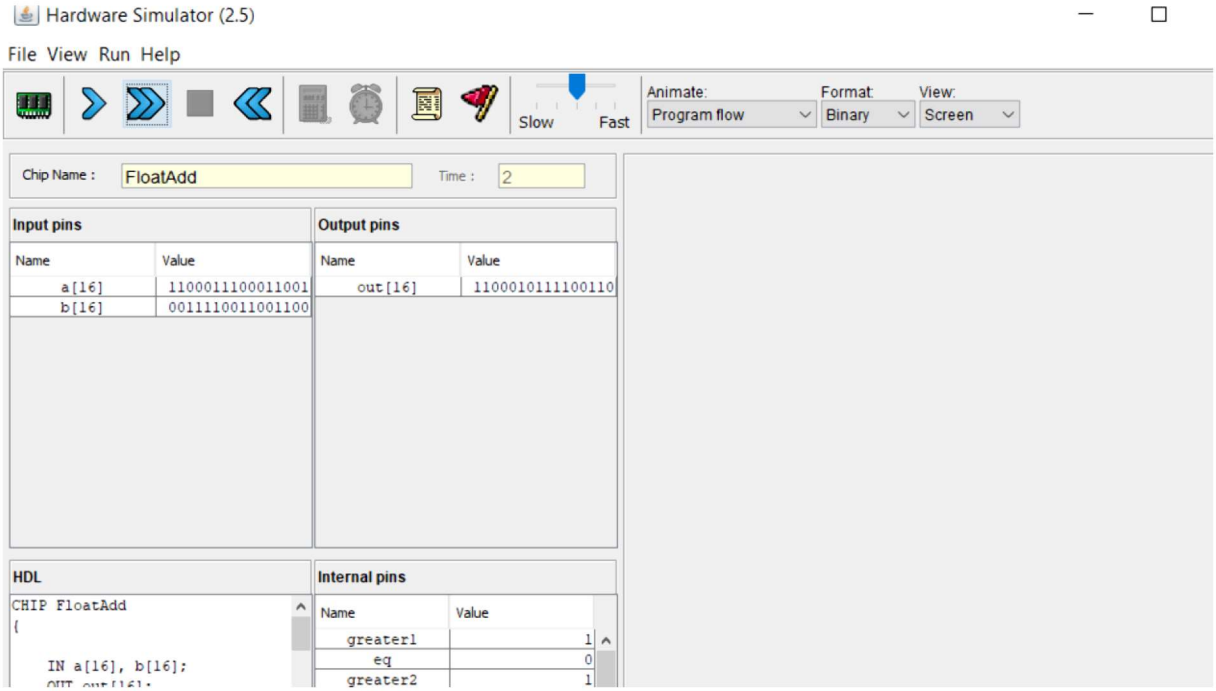
Operacije zbrajanje i oduzimanja izvršavaju se na istom čipu „FloatAdd“ vidljivom na slici 5. Te operacije implementirane su točno onako kako su opisane u 3. poglavlju. Prvo određujemo koji od dva ulaza je veći. Veći ulaz uvijek postavimo kao prvi, ukoliko je drugi ulaz veći obavljamo zamjenu. Nakon što smo odredili koji ulaz je veći računamo razliku eksponenata kako bi mogli pomaknuti mantisu manjeg toliko puta udesno. Kako je u HDL-u teško odnosno nemoguće implementirati petlje moramo izvršiti sva pomicanja te zatim na temelju razlike eksponenata odabrati ono koje nam odgovara. Iako je najveća moguća razlika eksponenata 29 primijetimo da se mantisa sastoji od samo 10 bitova te bi svako pomicanje nakon 10-tog dalo isti rezultat, sve 0 u mantisi, iz tog razloga spremamo samo 10 pomicanja i odabiremo između njih.

Nakon što smo odabrali odgovarajuće mantise zbrojimo ih odnosno oduzmemo, ponovno zbog nemogućnosti implementiranja petlji moramo obaviti obje operacije te odabrati jedan od rezultata. Ovaj put rezultat odabiremo na temelju predznaka ulaza, ukoliko su predznaci jednaki odabiremo rezultat zbrajanja, inače odabiremo rezultat oduzimanja. Nakon i tijekom implementacije bilo je potrebno testirati ovaj čip, to smo radili pokretanjem istog u Hardware simulatoru i unošenjem 16-bitnih ulaza, kako simulator nije namijenjen za korištenje s brojevima s pomičnim zarezom svaki broj smo morali unijeti u binarnoj reprezentaciji.



Slika 6. Primjer testiranja čipa „FloatAdd“ u Hardware simulatoru

Na slici 6 vidimo primjer zbrajanja dva broja pomoću čipa „FloatAdd“ koji je pokrenut u Hardware simulatoru. Brojevi su uneseni u svojim binarnim reprezentacijama, ali ako njih prevedemo u bazu 10 vidimo da se radi o brojevima 7.1 i 1.2, s desne strane vidimo rezultat koji je čip vratio. Nakon što njega prevedemo u bazu 10 vidimo da se radi o broju 8.3 što je uistinu rezultat zbrajanja dva broja na ulazu.



Slika 7. Primjer testiranja čipa „FloatAdd“

Kada usporedimo sliku 6 i sliku 7 vidimo da se ulazi razlikuju samo u predznaku prvog broja, no rezultat s desne strane je značajno različit. To je iz razloga što smo na slici 6 imali ulaze sa istim predznakom pa je rezultat bio rezultat zbrajanja brojeva na ulazu. Na slici 7 pak vidimo ulaze sa različitim predznacima, -7.1 i 1.2, pa je rezultat s desne strane rezultat oduzimanja manjeg broja od većeg s predznakom onog većeg. U ovom slučaju taj rezultat je -5.9.

4.2.2 Množenje

Osim aritmetičkih operacija zbrajanja i oduzimanja nad brojevima s pomičnim zarezom implementirali smo i operaciju množenja. Nju smo implementirali kroz čip „MultHalf“. Ponovno smo pratili operaciju množenja opisanu u 3. poglavlju. Za razliku od zbrajanja i oduzimanja ovdje nam nije bilo bitno koji ulaz je veći. Predznak smo odabrali pomoću XOR čipa, zatim smo zbrojili eksponente i od toga oduzeli unaprijed određen bias, koji je za ovaj format jednak 15. Naposlijetku množimo mantise i normaliziramo rezultat ukoliko je to potrebno. Kao što se moglo vidjeti u prijašnjem opisu ove operacije ona je bila malo jednostavnija za implementaciju.

```
CHIP MultHalf{
    IN a[16], b[16];
    OUT out[16];

    PARTS:

    //predznak
    Xor(a = a[15], b = b[15], out = out[15]);

    //eksponent
    Add16(a[0..4] = a[10..14], b[0..4] = b[10..14], out = hp);
    TwosCompl(in[0..3] = true, in[5..15] = true, out = hp2);
    Add16(a = hp, b = hp2, out[0..4] = hpe);

    //decimalni dio
    Mult(a[0..4] = a[5..9], a[5] = true, b[0..4] = b[5..9], b[5] = true,
         out = hp3);

    Mult(a[0..4] = a[0..4], a[5] = true, b[0..4] = b[0..4], b[5] = true,
         out = hp4);

    BitShiftR(in = hp4, out = hp5);
    BitShiftR(in = hp5, out = hp6);
    BitShiftR(in = hp6, out = hp7);
    BitShiftR(in = hp7, out = hp8);
    BitShiftR(in = hp8, out = hp9);
    And16(a[0..4] = true, b = hp9, out = hp10);
    Add16(a = hp3, b = hp10, out[0..9] = hpd, out[10..15] = hp11);

    Or16(a = false, b[0..5] = hp11, out[1] = hpe3);

    Add16(a[0..4] = hpe, b[0] = true, out = hpe2);

    BitShiftR(in[0..9] = hpd, out = hpd2);

    Mux16(a[0..4] = hpe, b = hpe2, sel = hpe3 , out[0..4] = out[10..14]);
    Mux16(a[0..9] = hpd, b = hpd2, sel = hpe3, out[0..9] = out[0..9]);


}
```

Slika 8. Implementacija čipa „MultHalf“ u HDL-u

Na slici 8 možemo vidjeti kompletnu implementaciju čipa „MultHalf“ u HDL-u. Primijetimo kako je u implementaciji množenje mantisa podijeljeno u dva dijela, razlog tomu je povećanje preciznosti kod množenja.

4.3 Integraciju u CPU

Nakon same implementacije FPU je bilo potrebno integrirati u CPU kako bi Hack računalo doista moglo izvršavati operacije na brojevima s pomičnim zarezom. CPU prima 16-bitne instrukcije, zapisane u strojnom jeziku, koje određuju koja će se operacija izvršiti. Te instrukcije mogu biti A-instrukcije ili C-instrukcije. Razlika između te dvije instrukcije nalazi se u prvom bitu, A-instrukcije uvijek započinju nulom, a C-instrukcije jedinicom. A-instrukcije služe za dodavanje numeričkih konstanti dok se C-instrukcije izvršavaju na ALU ili FPU te se njihov rezultat može također spremiti u A-registar. Kako bi integrirali FPU u CPU morali smo odrediti koje ćemo instrukcije slati na FPU, a koje na ALU. Za

implementaciju tog dijela služit će nam prva tri bita C-instrukcija. Oni će određivati radi li se o instrukciji koja se treba izvršiti na ALU ili instrukciji koja se treba izvršiti na FPU. Instrukcije koje se šalju na ALU započinju s „111“, dok instrukcije koje se šalju na FPU započinju s „110“. Nakon ta prva 3 bita u C-instrukcijama dolazi 6 bitova koji određuju koja će se točno operacija izvršiti (zastavice spomenute u implementaciji FPU). Preostalih 6 bitova određuju hoće li se rezultat izvršene operacije spremiti i gdje te hoćemo li skočiti na određenu instrukciju.

U idućem poglavlju reći ćemo nešto o softverskoj implementaciji FPU, proći ćemo kroz asembleru i parser koji ga prevodi u strojni jezik. Kratko ćemo se dotaknuti i virtualnog stroja te pripadnog parsera za jezik virtualnog stroja, a upoznat ćemo se i s osnovama Jack jezika te implementacijom pripadnog kompajlera.

5. Softverska implementacija

5.1 Asembler

Svaka naredba u asembleru prevodi se i odgovara točno jednoj naredbi u strojnom jeziku. Asemblerski jezik lakše je razumjeti od strojnog jezika, ali i dalje nije praktičan za pisanje velikih programa. Kako svaka naredba odgovara točno jednoj instrukciji u strojnom jeziku i u asebmleru moramo razlikovati A i C-instrukcije. U asembleru A-instrukcije započinju znakom „@“ iza kojeg slijedi numerička konstanta. C-instrukcije su oblika „*dest = comp ; jmp*“, gdje „comp“ predstavlja operaciju koja se izvršava, „dest“ registar u koji će se rezultat operacije zapisati i „jmp“ predstavlja skok. Jedini obvezan dio C-instrukcije je „comp“. U Hack strojnom jeziku razlikujemo tri različita registra: D (za pohranu podataka) , A (za pohranu i adresiranje) i M (trenutno adresirani registar). U „jmp“ djelu nalazi se logički izraz, ukoliko je on istinit izvršava se skok na instrukciju ROM[A].

5.2 Parser za asembler

Parsiranje asemblera obavlja se liniju po liniju i svaka naredba prevodi se u točno jednu instrukciju strojnog jezika. Cijeli parser za asembler implementiran je u jeziku Python i na ulazu prima ASM datoteku dok na izlazu vraća HACK datoteku s 16-bitnim instrukcijama u strojnom jeziku. Prolaskom kroz svaku liniju prvo provjeravamo predstavlja li ta linija A-instrukciju, odnosno započinje li linija znakom „@“. Ukoliko linija započinje znakom „@“ znamo da iza njega slijedi numerička konstanta koja može biti cijeli broj ili broj s pomičnim zarezom. Ukoliko smo naišli na cijeli broj njega pretvaramo u binarni i nadopunjujemo s nulama do 16 bitova, ako smo pak naišli na broj s pomičnim zarezom njega pretvaramo u binarni prema IEE 754 standardu. Tako dobivenu A-instrukciju zapisujemo u HACK datoteku.

```
from converter import float_to_IEEE754_16bit

def _parse_command(self, line, b, c):
    # A instrukcija (@844)
    if line[0] == "@":
       res = str(line[1:]).split(".")
       if len(res) > 1:
            return float_to_IEEE754_16bit(float(line[1:]))
       else :
           num = "{0:b}".format(int(line[1:]))
           return "0" * (16 - len(num)) + num
```

Slika 9. Dio koda ASM parsera koji parsira A-instrukcije

Ukoliko smo naišli na C-instrukciju nju dekodiramo pomoću unaprijed inicijaliziranih rječnika za operacije, destinacije i skokove. Kako bi mogli izvršavati operacije nad brojevima s pomičnim zarezom i njih slati na FPU moramo ih nekako razlikovati od operacija koje se trebaju izvršavati na ALU. Postojeći parser proširili smo novim rječnikom koji sadrži instrukcije namijenjene izvršavanju na FPU. Te instrukcije posebno su naznačene znakom „*“. Kada u C-instrukciji naiđemo na „*“ bit koji određuje gdje će se instrukcija izvršiti postavljamo na 0 kako bi e ona izvršila na FPU. Konačnu 16-bitnu C-instrukciju u strojnom jeziku dobijemo na sljedeći način: „11“, 1 bit koji označava šalje li se instrukcija na FPU ili

ALU, 6 bitova koji označavaju koja se operacija treba izvršiti (dekodirano pomoću rječnika), 3 bita za destinaciju (dekodirano pomoću rječnika) i 3 bita za skok (dekodirani pomoću rječnika). Nakon što dobijemo odgovarajuću 16-bitnu instrukciju nju zapisujemo u HACK datoteku. Osim A i C-instrukcija u ASM-u možemo pisati i komentare označene s „//“. Linije koje započinju s „//“ prilikom parsiranja se ignoriraju i ne prevode ni na koji način. Nakon što smo na ovaj način prošli kroz cijelu ASM datoteku dobijemo pripadnu HACK datoteku koja se sastoji od niza instrukcija u strojnom jeziku koju zatim možemo pokretati u Hack računalu.

```
126         self._float_op = {
127             "*0": "0101010",
128             "*D": "0001010",
129             "*A": "0100010",
130             "*-D": "0011010",
131             "*-A": "0100110",
132             "*D+A": "0000010",
133             "*A+D": "0000010",
134             "*D-A": "0000110",
135             "*A-D": "0010010",
136             "*M": "1100010",
137             "*-M": "1100110",
138             "*D+M": "1000010",
139             "*M+D": "1000010",
140             "*D-M": "1000110",
141             "*M-D": "1010010"
142         }
143         self._jmp = {
144             "" : "000",
145             "JGT": "001",
146             "JEQ": "010",
147             "JGE": "011",
148             "JLT": "100",
149             "JNE": "101",
150             "JLE": "110",
151             "JMP": "111"
152         }
153         self._dest = {
154             "" : "000",
155             "M" : "001",
156             "D" : "010",
157             "MD" : "011",
158             "A" : "100",
159             "AM" : "101",
160             "AD" : "110",
161             "AMD" : "111"
162         }
```

Slika 10. Inicijalizacija rječnika za operacije nad brojevima s pomičnim zarezom, skokove i destinacije

```
61  //push local 1
62  @1
63  D=A
64  @LCL
65  A=D+M
66  D=M
67  @SP
68  M=M+1
69  A=M-1
70  M=D
71  //floatgt
72  @SP
73  AM=M-1
74  D=M
75  A=A-1
76  D=*M-D
77  @LAB0
78  D;JGT
79  @LAB1
80  D=0;JMP
```

```
51  1111110111001000
52  1111110010100000
53  1110001100001000
54  0000000000000001
55  1110110000010000
56  0000000000000001
57  1111000010100000
58  1111110000010000
59  0000000000000000
60  1111110111001000
61  1111110010100000
62  1110001100001000
63  0000000000000000
64  1111110010101000
65  1111110000010000
66  1110110010100000
67  1101010010010000
68  0000000001000111
69  1110001100000001
70  0000000001001000
```

Slika 11. Primjer ASM datoteke i pripadne HACK datoteke nakon parsiranja

5.3 Virtualni stroj i parser

Virtualni stroj omogućava emulaciju računalnih sustava i služi za pokretanje programa i operacijskih sustava. On predstavlja razinu iznad asemblera i u njemu je lakše pisati veće programe. Svaka linija VM koda predstavlja nekoliko linija ASM koda. Kako bi se jezik virtualnog stroja mogao prevesti u ASM morali smo implementirati parser za isti i doraditi ga kako bi omogućili pisanje programa koji manipuliraju brojevima s pomičnim zarezom. Virtualni stroj koristi apstrakciju memorije pomoću stoga te sukladno tome koristi naredbe kao što su „pop“, „push“ i ostale. Aritmetičke operacije označene su ključnim riječima kao što su „add“, „sub“, itd. Kako bi ove operacije razlikovali od operacija koje se trebaju izvršavati nad brojevima s pomičnim zarezom ispred njih smo dodali ključnu riječ „float“. Tako je operacija zbrajanja nad brojevima s pomičnim zarezom označena s „floatadd“. Svaka od tih operacija prevodi se u više linija ASM koda što je vidljivo na slici 12. Parser za jezik virtualnog stroja također je implementiran u Pythonu i sličan je ASM parseru.

```
235     def _comm(self, comm, n):
236         if comm == "add":
237             l = "@SP\nAM=M-1\nD=M\nA=A-1\nM=M+D"
238         elif comm == "floatadd":
239             l = "@SP\nAM=M-1\nD=M\nA=A-1\nM=*M+D"
240         elif comm == "sub":
241             l = "@SP\nAM=M-1\nD=M\nA=A-1\nM=M-D"
242         elif comm == "floatsub":
243             l = "@SP\nAM=M-1\nD=M\nA=A-1\nM=*M-D"
244         elif comm == "neg":
245             l = "@SP\nA=M-1\nM=-M"
246         elif comm == "floatneg":
247             l = "@SP\nA=M-1\nM=*-M"
248         elif comm == "and":
249             l = "@SP\nAM=M-1\nD=M\nA=A-1\nM=M&D"
250         elif comm == "or":
251             l = "@SP\nAM=M-1\nD=M\nA=A-1\nM=M|D"
252         elif comm == "not":
253             l = "@SP\nA=M-1\nM=!M"
```

Slika 12. Dio koda VM parsera koji prevodi operacije u više linija ASM koda

5.4 Jack jezik i kompajler

Jack jezik je objektno orijentiran jezik posebno dizajniran za Hack računalo. Sav kod neke Jack datoteke mora se nalaziti u jednoj klasi koja je istog naziva kao i datoteka. Jack jezik zadnja je i najviša razina u kojoj smo omogućili korištenje brojeva s pomičnim zarezom. Dopunjavanjem gramatike Jack jezika, koju možemo vidjeti na slici 13, tipom „float“ proširili smo jezik tako da mu omogućimo rad s brojevima s pomičnim zarezom. Nadopunili smo postojeći kompajler kako bi mogao prepoznati radi li se o operacijama nad brojevima s pomičnim zarezom ili cijelim brojevima. Za razliku od prošle dvije razine ovdje razliku ne detektiramo posebnom oznakom već se ona određuje s obzirom na tipove varijabli. Ukoliko se operacija izvršava na dvije varijable tipa „float“ ona će se prevesti kao „float“ operaciju u jeziku virtualnog stroja. Tako primjerice ukoliko se radi o zbrajanju dva podataka tipa „float“ u Jacku, pripadna operacija u jeziku virtualnog stroja bit će „floatadd“.

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 13. Potpuna gramatika Jack jezika

Kao i prethodna dva parsera, Jack kompajler je u potpunosti implementiran u Pythonu. Njegovu cijelu implementaciju, kao i implementaciju svakog od parsera i HDL implementaciju floating-point jedinice i pripadnog Hack računala moguće je pronaći na linku projekta koji se nalazi u prvom poglavlju.

6. Zaključak

Prvo smo se kroz kolegij Moderni računalni sustavi upoznali s građom i načinom rada računala na primjeru 16-bitnog računala. Kroz ovaj rad smo to znanje nadogradili i proširili kroz proces potpune implementacije floating-point jedinice. Dizajnirali smo i odradili hardversku implementaciju floating-point jedinice. Kroz tu implementaciju upoznali smo se s IEEE 754 standardom za zapisivanje brojeva s pomičnim zarezom, floating-point aritmetikom definiranom u tom standardu kao i s osnovama HDL-a. Nakon same hardverske implementacije dotakli smo se i one softverske, objasnili A i C-instrukcije u strojnom jeziku, asemblerski jezik te prošli kroz implementaciju parsera za asembler. Kratko smo se dotakli i onih najviših razina implementacije što su virtualni stroj i Jack jezik. Naposlijetku smo dobili prošireno Hack računalo koje ima mogućnost zapisivanja i manipuliranja brojevima s pomičnim zarezom.

Nastavak ovog rada i projekta mogla bi biti nadogradnja alata za testiranje svih pojedinih dijelova implementacije kako bi se olakšalo testiranje i rad s brojevima s pomičnim zarezom na svim razinama. Nadalje bi to bila eventualna optimizacija hardverskog djela implementacije i nadopuna Jack jezika posebnim klasama za zapisivanje, manipuliranje i prikaz brojeva s pomičnim zarezom.

7. 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] Grune, D. & Jacobs, C. J. (1990), *Parsing Techniques - A Practical Guide*, Ellis Horwood, Chichester, England .

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

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

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