

Hyvönen Saku

Ajonaikaisesti käännetty ohjelmointikieli

```
6  »  »  cexp ret = compile_expr(st, begintoken+1, outvid, ac, vars, glm, dumystack);
7  »  »  ret.dummiesneeded = r.dummiesneeded;
8  »  »
9  »  »  std::cout << "set dest var " << outvid << " src var mem " << outvid << std::endl;
10 »  »  ac.pi(zvm_instr::set(1, data(zvvar(outvid)), data(zvvar(outvid)), true, zvm_type:
11 »  »
12 »  »  ret.endtoken = r.endtoken;
13 »  »  } else
14 »  »  if(bgst == "eq" || bgst == "neq" || bgst == "gt" || bgst == "lt" || bgst == "gte" ||
15 »  »  if(tokencount < begintoken+2) {ERRXIT("Not enough arguments");}
16 »  »  if(st.tokens[begin token+1].type != token::NORMAL) {ERRXIT("TODO");}
17 »  »  if(st.tokens[begin token+2].type != token::NORMAL) {ERRXIT("TODO");}
18 »  »
19 »  »  //Comparisons return 1 on true
20 »  »
21 »  »  //Arguments
22 »  »  std::cout << outvid << ", dummies < fnargs" << std::endl;
23 »  »  fnargsparse_ret r = fnargsparse_2(st, begintoken+1, 2, ac, vars, glm, outvid, dur
24 »  »  ret.dummiesneeded = r.dummiesneeded;
25 »  »
26 »  »  //Set to 1. if condition is false set to 0
27 »  »  zvm_glblm::label skip = glm.new_();
28 »  »  ac.pi(zvm_instr::compare(data(zvvar(outvid)), data(zvvar(dumystack)))); // !! fla
29 »  »  ac.pi(zvm_instr::set(1, data(zvvar(outvid)), data(zvm_instr_data_imm::_i64(1))));
30 »  »  if(bgst == "eq") {
31 »  »  »  ac.pi(zvm_instr::branch(skip, cndtype::EQ));
32 »  »  } else if(bgst == "neq") {
33 »  »  »  ac.pi(zvm_instr::branch(skip, cndtype::NEQ));
34 »  »  } else if(bgst == "gt") {
35 »  »  »  ac.pi(zvm_instr::branch(skip, cndtype::GT));
36 »  »  } else if(bgst == "lt") {
37 »  »  »  ac.pi(zvm_instr::branch(skip, cndtype::LT));
38 »  »  } else if(bgst == "gte") {
39 »  »  »  ac.pi(zvm_instr::branch(skip, cndtype::GTE));
40 »  »  } else if(bgst == "lte") {
41 »  »  »  ac.pi(zvm_instr::branch(skip, cndtype::LTE));
42 »  »  }
43 »  »  ac.pi(zvm_instr::set(1, data(zvvar(outvid)), data(zvm_instr_data_imm::_i64(0))));
44 »  »  ac.label_bind(skip);
45 »  »
46 »  »  ret.endtoken = r.endtoken;
47 »  »  } else //EXT functions
48 »  »  if(extfn_by_name(bgst) != nullptr) {
49 »  »  »  zvm_extfn* extf = extfn_by_name(bgst);
50 »  »  »  const szet argc = extf->sig.args.size();
```

Insinööri (AMK)

Tieto- ja viestintätekniikka

Kevät 2021



KAMK • University
of Applied Sciences

Tiivistelmä

Tekijä(t): Hyvönen Saku

Työn nimi: Ajonaikaisesti käännetty ohjelmointikieli

Tutkintonimike: Insinööri (AMK), tieto- ja viestintätekniikka

Asiasanat: Kääntäjä, ohjelmointikieli, x86.

Tämä opinnäytetyö on tehty aikaisemmin aloitetusta henkilökohtaisesta projektista, jossa tarkoituksena on luoda ajonaikaisesti käännetty ohjelmointikieli. Tämä ohjelmointikieli päätettiin luoda johtuen syvemmästä mielenkiinnosta ohjelmointikieliä, ja kääntäjiä kohtaan. Tämä projekti koettiin melko tärkeäksi niin, että lopputuloksesta yritettiin saada tasokas. Ohjelmointikielen nopeutta pidettiin erityisen suurena osana laatu. Ohjelmointikielien kääntämisellä prosessorille on tunnettu tehokkuusetu, joten kielestä päätettiin luoda käännetty. Tämä projekti oli liian suuri opinnäytetyöksi, joten kaikkia projektin osa-alueita on yksinkertaistettu huomattavasti. Päättävöitteeksi asetettiin toimiva kokonaisuus, jossa mahdollisimman yksinkertainen ohjelmointikieli käännetään prosessorille.

Projektin hyvin aikaisessa vaiheessa kielelle oli päätettävä tarkka toteutustapa. Ohjelmointikielen toteutustapoja aloitettiin tutkimaan. Tutkimuksessa havaittiin useita etuja sille, että kieli käännettäisiin ensin virtuaalikoneelle, ennen prosessorille kääntämistä. Virtuaalikone voidaan suunnitella alustastaan riippumattomaksi alustaksi. Tämä tarkoittaa sitä, että virtuaalikoneen ohjelmalla toteutetut kielet tukevat kaikkia alustoja, joilla virtuaalikoneen ohjelmien suoritus on tuettu. Virtuaalikoneella on paljon suurempikin etu, virtuaalikone voidaan myös suunnitella kieliriippumattomaksi niin, että sille voidaan kääntää useita erilaisia ohjelmointikieliä. Tässä projektissa tämä sallii paljon vapaamman kielten prototyyppien rakentamisen niin, että vain pieneen, kieltä lähellä oleviin osiin kääntäjässä pitää tehdä muutoksia. Näistä suurista eduista johtuen projektissa koettiin hyvin tärkeäksi kääntää kieli ensin virtuaalikoneen ohjelmaksi, joka taas sitten käännetään eteenpäin prosessorille. Tämä asetettiin tavoitteeksi.

Opinnäytetyön tuloksena syntyi tavoitteena oleva kokonaisuus, jossa omatekoinen ohjelmointikieli käännetään ensin virtuaalikoneen ohjelmaksi, ja tämän virtuaalikoneen ohjelma sitten käännetään eteenpäin x86-prosessorille Linux-ympäristössä. Tämän toimivan kokonaisuuden aikaansaaminen oli ominaisuuksia ja hienosäätöä korkeampi prioriteetti. Ominaisuuksien puutos näkyy selvästi ohjelmointikielessä. Ohjelmointikieli ei tue muuta, kuin laskutoimituksia, päätöksentekoa (if-lause), silmukoita (while-silmukka), ja konsolille tulostusta. Virtuaalikone ja kääntäjä juuri ja juuri tukevat tarvittavia ominaisuuksia tämän kielen kääntämiseen. Kääntäjä ei myöskään suorita mitään optimointeja. Tämän lisäksi toteutuksessa on jouduttu tekemään monia yksinkertaistuksia. Tässä dokumentissa on esitetty lukuisia parannusehdotuksia nykyiseen kokonaisuuteen. Kokonaisuudessa on paljon rajoitteita ja puutteita, mutta tämän rajatun opinnäytetyön tavoite ei ollutkaan luoda täydellistä ohjelmointikieltä tai kääntäjää, joten tavoitteet on saavutettu.

Tällä projektilla on käytännössä loputtomat laajennus- ja jatkokehitysmahdollisuudet. Alkuperäisenä ideana oli tehdä hyvä ja käytännöllinen kieli. Tätä projektia aiotaan jatkaa opinnäytetyön jälkeen. Ohjelmointikielen ja kääntäjän suunnittelu ovat kummatkin hyvin aikaavieviä prosesseja. Itse kielen ja tehokkaan kääntäjän lisäksi kielelle voidaan rakentaa kattava joukko kirjastoja. Ohjelmointikielelle voidaan myös rakentaa työkalut, kuten esimerkiksi kehitysympäristö.

Abstract

Author(s): Hyvönen Saku

Title of the Publication: Just-In-Time Compiled Programming Language

Degree Title: Bachelor of Engineering, Information and communication technologies

Keywords: Compiler, programming language, x86

This thesis has been made from previously started personal project, where the purpose was to create a self-made compiled programming language. The project was started because of personal interest towards compilers and programming languages. The language has been implemented as a compiled language because there is a well-known performance benefit to compiling a language. This project is too broad for it to be a good subject for thesis, so all parts of the project have been simplified. The main goal of this thesis was set to creating a functional compiler, that compiles as simple as possible programming language for a real processor.

The implementation method of the language must be decided in early stages of its development. Ways to implement compiled languages were studied. Multiple advantages were found for compiling the language into a program for virtual machine, and then compiling the virtual machine's program for a real processor. The virtual machine can be designed to be a platform independent platform, so that when a language is implemented as a program for the virtual machine, the language supports all platforms where running the virtual machine's programs are supported. The platform provided by the virtual machine can also be language independent so, that multiple different languages can be implemented for it. In the context of this project, this is a huge benefit as language prototypes can be freely developed, without doing significant changes to the compiler. Because of these discovered benefits, it was decided that a virtual machine should be used as an intermediary format for the compiler.

During this thesis, a functional compiler, and a self-made language were successfully created. Virtual machine has been used as an intermediary format for the compiler. The compiler can compile the virtual machine's programs for x86-processor under Linux-environment. All aspects of this project have been greatly simplified to get the project done in time. This is mostly visible in the created programming language. The language doesn't support anything else than simple arithmetic, conditional code (if-statement), loops (while-loop) and printing to console. Virtual machine doesn't support any more features, than what is strictly needed to support this simple language. The compiler that compiles the virtual machine's programs for the processor, doesn't support anything that is not strictly needed for this task. The compiler for example, doesn't do any optimizations. In this document, many improvement suggestions have been shown for all aspects of the project. There are many limitations in the current design, but the simple goal of compiling the language has been accomplished.

This project has practically unlimited options for future development. The original idea of the project was to create a useful programming language, and the interest towards this project has not gone down. The development of the compiler and the language is a very time-consuming process. In addition to creating an efficient programming language, and a compiler, development tools such as an integrated development environment can be created for the language.

Sisällys

1	Johdanto	1
2	Ohjelmointikielien toteutuksen teoriaa	3
2.1	Operaatioiden suoritus.....	3
2.1.1	Tavukoodi ja virtuaalikoneet.....	4
2.1.2	Tulkkauksen tehokkuus.....	4
2.1.3	Miten tulkki toimii.....	6
2.2	Ohjelmointikielen toteutuksen tavat	7
3	Motivaatio luodulle kääntäjälle.....	10
4	Tavoitteet luodulle kääntäjälle	11
5	Yleiskatsaus luodusta kääntäjästä	12
6	Virtuaalikone	14
6.1	Muuttujat	16
6.2	Käskyt	17
6.3	Viittaukset	18
6.4	Ulkoiset funktiot.....	18
6.5	Muistinhallinta	19
6.6	Sisäisten funktioiden toteutus	20
6.6.1	Nykyisen muuttujien käsittelyn ongelmat	20
6.6.2	Funktioiden ja muuttujien suhde	21
6.6.3	Rajapinta funktioille	21
6.6.4	Toteutus rajapinnalle	22
6.7	Virtuaalikone tulevaisuudessa	24
6.7.1	Ohjelman määrittäminen ja muuttujien luominen	24
6.7.2	Tietorakenteet.....	25
6.7.3	Ohjeiden parametrit.....	26
6.7.4	Käskyjen puhtaus	26
7	Matalan tason kääntäjä	28
7.1	Kääntäjän käyttäytyminen yleisesti.....	29
7.1.1	Muuttujat ja pinomuisti	29
7.1.2	Segmentointi	30

7.1.3	Binääritason standardien dokumentointi	30
7.1.4	Funktion kutsutapa	30
7.1.5	Upotettu tieto	31
7.1.6	Muistin kohdennus.....	32
7.2	Builder	33
7.2.1	Monisäikeinen koodin generointi	34
7.2.2	Ohjeiden korjaus	36
7.2.3	Builderin abstrakti formaatti	36
7.2.4	Viitejärjestelmä	37
7.2.5	Viittausjärjestelmän toteutus.....	40
7.2.6	Tavujonon luominen ohjelmalle.....	42
7.2.7	Builder tulevaisuudessa	45
7.3	Rekisterivaraaja	46
7.3.1	Rajapinta	47
7.3.2	Toteutus	49
7.3.3	Rekisterivaraaja tulevaisuudessa	51
7.4	Virtuaalikoneen käskyjen toteutus.....	52
7.5	Käännösprosessi.....	52
7.6	Liukulukujen toteutus.....	56
7.7	Matalan tason kääntäjä tulevaisuudessa	58
8	Demokieli	59
8.1	Syntaksi.....	59
8.2	Parametrien jäsenitys.....	61
8.3	Kielen tukemat operaatiot	62
8.4	Muistinhallinta	62
8.5	Teksti	62
8.6	Kääntäminen virtuaalikoneen ohjelmaksi.....	63
8.6.1	Jäsenitys	63
8.6.2	Ohjelman kääntäminen.....	64
8.6.3	Ilmaisujen kääntäminen	66
8.6.3.1	Ilmaisujen toteutukset	67
8.6.3.2	Ilmaisun kääntäjän optimoinnit	68
9	Yhteenveto	69
	Lähteet	72

Liitteet

Symboliluettelo

AOT	Ahead of time.
Builder	Matalan tason komponentti mitä käytetään kokoamaan x86-prosessorin ohjelmia.
Demokieli	Työtä varten luotu ohjelmointikieli.
Epilogi	Funktion varsinaisen kehon tai sen kutsun jälkeen luotu koodi. Pääasiassa käyttäjälle näkymätön, funktion suorituksen ympäristön asettava koodi.
FPU	Floating Point Unit. Matematiikkasuoritin. Prosessorin sisäinen komponentti, joka suorittaa laskutoimituksia liukuluvuilla.
GCC	GNU Compiler Collection.
JIT	Just In Time compiler. Ajonaikainen kääntäjä. Ajonaikainen kääntäjä kääntää ohjelman koodia samaan aikaan kun sitä suoritetaan.
JVM	Java Virtual Machine. Virtuaalikone, jota on käytetty Java-ohjelmointikielen toteutuksessa.
Kääntäjä	Kääntää lähdekielen kohdekieleksi. Tunnetaan siitä, että kääntäjää käytetään kääntämään ohjelma prosessorille.
Operandi	Nimitys prosessorin ohjeiden parametreille.
PC	Personal Computer.
Prologi	Funktion varsinaisen kehon tai sen kutsun edeltävä koodi. Pääasiassa käyttäjälle näkymätön, funktion suorituksen ympäristön asettava koodi.
Puhtaus	Esimerkiksi puhdas funktio ei riipu muusta ohjelman tilasta, kuin sille annetuista parametreista. Puhdas funktio ei myöskään muuta ohjelman tilaa sen paluuarvon ulkopuolella. Vastaa GCC:n funktioiden attribuuttia "const".
SIMD	Single instruction, multiple data. Yksi prosessorin ohje voi suorittaa laskutoimituksen suurelle määrälle arvoja kerralla. Ohjetta voidaan kutsua myös vektoriohjeeksi.

Sisäinen fn	Sisäinen funktio. Projektissa luodun virtuaalikoneen ohjelman sisällä luotu funktio. Määritetään käännettävällä kielellä.
SSE/AVX	x86-prosessorin ohjejoukkoja.
Tavukoodi	Virtuaalikoneen syöte. Tavukoodi on raakatekstiä nopeammin tulkattava muoto. Tavukoodi saattaa muistuttaa oikean prosessorin koodia.
Tokeni	Demokielen syntaksista selviävä yksittäinen sana tai merkkikokonaisuus. Tähän sanaan on lisätty luokittelevia metatietoja.
Tokenisointi	Kääntämisen vaihe, joka ottaa raakatekstin sisään, ja antaa ulos listan tokeneista.
Tulkki	Suorittaa operaatioita käsittelemällä jotakin syötettä. Käytetään suorittamaan ohjelmointikielellä luotuja ohjelmia.
Ulkoinen fn	Ulkoinen funktio. Projektissa luodun virtuaalikoneen ohjelman ulkopuolella määritetty funktio. Määritetään C++-kielellä.
VE	Var Emitter. Rekisterivaraaja. Analysoi ohjelmaa ja päättää ohjelmassa käytettyjen muuttujien rekisterit.
Vektorisointi	Prosessi, jossa jokin pala koodia luodaan hyödyntämään vektoriohjeita. Voi tapahtua joko käyttäjän tai kääntäjän toimesta.
Virtuaalikone	Ohjelmallisesti toteutettu kuvitteellinen prosessori. Tavukoodin tulkkia kutsutaan usein virtuaalikoneeksi. Käytetään toteuttamaan ohjelmointikieliä.
x86	Intel:in ja AMD:n kehittämä prosessoriarkkitehtuuri.
XED	x86 Encoder Decoder. Kirjasto, jota kääntäjä käyttää koodaamaan prosessorin ohjeita.
ZVM	Z Virtual Machine. Työssä luotu kääntäjän väliformaattina käytetty virtuaalikone, tai prosessori.
ZVJ	Z Virtual Machine JIT. Toiselta nimeltään ”matalan tason kääntäjä”. Kääntää ZVM-ohjelmat x86-prosessorille.

1 Johdanto

Tämä opinnäytetyö on tehty aikaisemmin aloitetusta henkilökohtaisesta projektista, missä on kehitetty ajonaikaisesti käännettyä ohjelmointikieltä. Projekti aloitettiin puhtaasti omasta mielenkiinnosta johtuen. Projektissa pyrittiin luomaan hyvä, käytännöllinen ja nopea ohjelmointikieli. Tämän kielen nopeutta pidettiin erityisen tärkeänä. Projektista päätettiin myöhemmin tehdä opinnäytetyö. Opinnäytetyöhön kuuluvien aikataulurajoitteiden ja projektin suuruuden takia opinnäytetyö rajattiin niin, että työn tavoitteena oli kehittää mahdollisimman yksinkertainen ohjelmointikieli, ja saada se kääntymään prosessorille. Tämän kokonaisuuden aikaansaamiseksi projekti rajattiin niin, että siinä toteutetaan vain mahdollisimman minimaalinen joukko toimintoja.

Kielen nopeutta pidettiin projektissa erityisen tärkeänä. Yleisesti ottaen ohjelmointikielen suorituskyvyn mittaaminen ei välttämättä ole kovin hyvä idea, koska ohjelmointikielen tehokkuus riippuu pääasiassa sen toteutuksesta. Näin ollen suorituskyyä arvioitaessa mitataankin kielen sijaan sen toteutuksen tehokkuutta. Koska tässä projektissa omalle kielelle haetaan toteutusta, niin yksinkertainen tehokkuusmittaus, kuten [1], antaa projektin kannalta arvokasta tietoa. Tässä testissä on kirjoitettu Mandelbrot-joukon laskennan suorittava ohjelma useilla kielillä. Testissä on verrattu eri kielten, ja niiden toteutuksien nopeutta. Tämä on hyvin tyypillinen ohjelmointikielen tehokkuusmittaus. Mittaustulokset osoittavat selkeän tehokkuuseron kääntämistyyppisten ja tulkkaustyyppisten toteutusten välillä: käännetyt kielet ovat joissakin tapauksissa jopa 45-kertaisesti tehokkaampia kuin tulkattavat ohjelmointikielet. Jotta omasta kielestä saataisiin todellisesti nopea, niin on kääntämisen hyödyntäminen melkein välttämätöntä.

Kun toteutustapoja tutkittiin enemmän, niin havaittiin useita etuja sille, että kieli käännetään ensin virtuaalikoneelle, ja tämän virtuaalikoneen ohjelma käännetään sitten eteenpäin prosessorille. Virtuaalikone tarkoittaa eräänlaista kuvitteellista prosessoria, jolla on kuvitteellinen käskyjoukko. Tästä syystä virtuaalikoneelle käytetään tätä nimitystä. Virtuaalikoneella on etu, että se voidaan rakentaa riippumattomaksi alustastaan niin, että sille käännetyt kielet tukevat alustoja, joilla virtuaalikoneen ohjelmien suoritus on tuettu. Tämän lisäksi virtuaalikone voidaan myös rakentaa riippumattomaksi kielestään. Tämä on erityisen suuri etu tämän projektin kannalta, koska tämä sallii vapaamman kielten prototyyppien rakentamisen virtuaalikoneelle niin, että vain pienen kieltä lähellä oleviin osiin kääntäjässä tarvitsee tehdä muutoksia. Tästä syystä kieli päätettiin kääntää virtuaalikoneelle ennen prosessorille kääntämistä.

Tämä dokumentti käsittelee ensin ohjelmointikielien toteutuksen teoriaa tarkemmin. Tämän jälkeen motivaatio luodulle kääntäjälle esitetään teoriaan perustuen. Lopuksi dokumentti käsittelee luodun kääntäjän rakennetta, joka on suurin puhuttu aihe dokumentissa. Kääntäjän koodin rakennetta käsitellään melko ympäröivästi komponenttitasolla menemättä täsmälliseen lähdekoodin rakenteeseen. Komponentteja käsitellään mahdollisimman eristyksissä toisistaan.

Dokumentti puhuu matalan tason ohjelmoinnista vain, kun se on aiheuttanut suurempien toimien toteutuksen tarvetta. Ennen dokumentin lukemista on suositeltavaa lukea x86-prosessorien toiminnoista, ja niille käyvien ohjelmien luomisesta koontikielellä.

2 Ohjelmointikielien toteutuksen teoriaa

Projektin alkuvaiheilla projektissa luotavalle kielelle oli päätettävä toteutustapa. Ohjelmointikielen toteutus on laaja projekti, joten projekti on tärkeää suunnitella hyvin. Tästä syystä projektissa aloitettiin tutkimaan erilaisia kielen toteutuksen tapoja.

Ohjelmointikielen toteutustavan valinta on tärkeää tehdä jo hyvin aikaisessa vaiheessa sen suunnittelua. Jos nopeus on ohjelmointikielen tavoitteena, niin kieli pitää toteuttaa jollakin tavalla prosessorille käännettynä. Ohjelmointikielen suunnittelu ja ominaisuudet vaikuttavat siihen, että kuinka tehokkaasti kieli pystytään kääntämään prosessorille. Vaikka hyvin älykäs kääntäjä voi optimoida tehokkuuden kannalta ongelmalliset ominaisuudet, niin tämän kääntäjän luominen voi olla hyvin vaikeaa, tai käytännössä mahdotonta. Jos kieli halutaan lopulta kääntää, niin se olisi hyvä ottaa huomioon jo ennen kielen ominaisuuksien suunnittelemista. [2]

Ohjelmointikielen toteutus on järjestelmä, joka kertoo, kuinka kielellä luotu ohjelma lopulta suoritetaan. Ohjelmointikielen toteutuksen on lopulta suoritettava joukko operaatioita, jotka vastaavat ohjelmointikielellä kuvattua toimintoa. Tämänlaisia operaatioita voivat olla esimerkiksi kahden numeron yhteen- ja vähennyslaskut.

2.1 Operaatioiden suoritus

Operaatioita voidaan suorittaa vain kahdella tapaa. Operaatioita voidaan suorittaa tietokoneen oikealla prosessorilla, tai ei. Kun operaatioita suoritetaan prosessorilla, niin tässä tapauksessa kyseessä on x86-prosessorille sopiva tavujono. Kun operaatioita ei suoriteta prosessorilla, niin silloin niitä tulkitaan jostakin syötteestä tulkiohjelmalla. Tulkiohjelma on itse oikealla prosessorilla ajettava ohjelma. Tulkiohjelma suorittaa operaatiot ohjelmallisesti, kun taas prosessori on fyysinen laite. On myös mahdollista suorittaa osa ohjelmasta tulkaamalla ja toinen prosessorilla. [3]

2.1.1 Tavukoodi ja virtuaalikoneet

Kielen tulkkaminen ei yleensä tapahdu lukemalla suoraan lähdekoodia, vaan tehokkuuden kannalta kieli yleensä käännetään ensin nopeammin tulkattavaan muotoon. Eräs yleinen lähestymistapa on kääntää kieli tavukoodiksi. Tavukoodi voi muistuttaa hieman oikean prosessorin koodia, joten tästä syystä tavukoodin tulkkia kutsutaan nimellä ”virtuaalikone”. [3] [4] [5] [6]

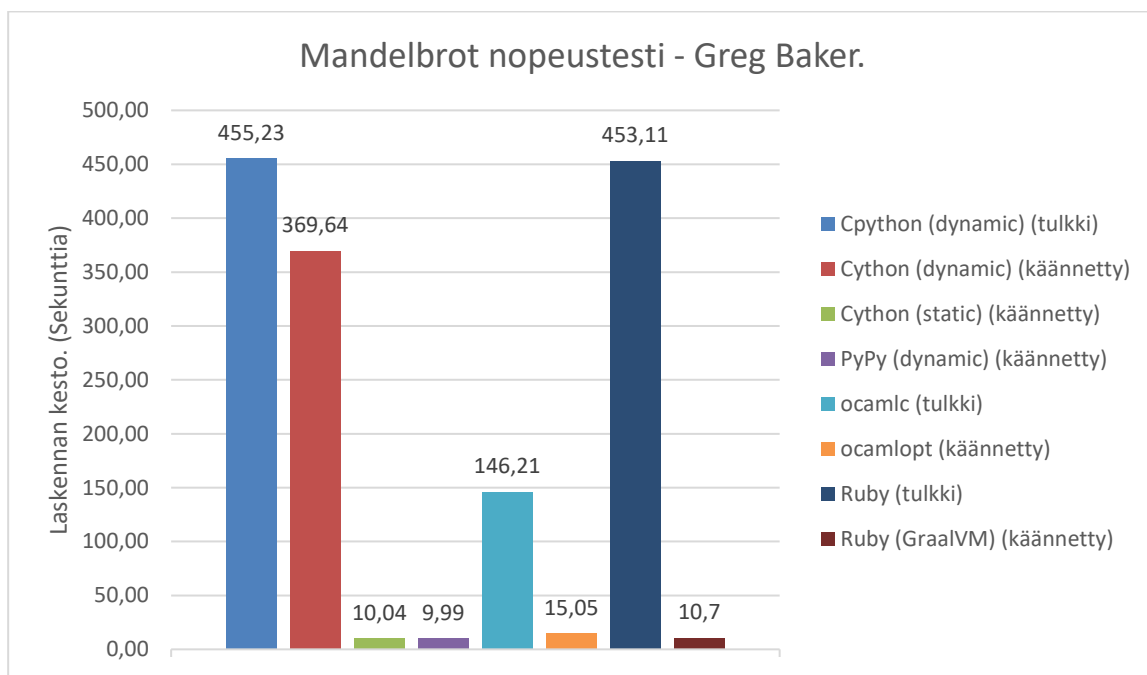
Virtuaalikone on täysin kuvitteellinen prosessori. Sillä on etu, että sen käskynä voidaan toteuttaa käytännössä mikä vaan toiminto [5]. Esimerkiksi tekstin tulostus konsolille voi olla vain yksi käsky virtuaalikoneelle. Virtuaalikoneen suunnittelussa on kuitenkin hyvä ottaa huomioon, että sen tavukoodia voidaan tulkata mahdollisimman nopeasti. Nopeus oli virtuaalikoneen alkuperäinen tarkoitus.

Virtuaalikoneen käyttötarkoitus ei kuitenkaan ole rajoittunut tulkin tehokkuuden nostamiseen. Virtuaalikone pystytään rakentamaan geneeriseksi niin, että sille voidaan kääntää useita eri ohjelmointikieliä. Esimerkiksi Java-virtuaalikoneelle (JVM) on toteutettu useita kieliä [7] [8] [9] [10] [11] [12]. JVM ei kuitenkaan välttämättä ole kovin geneerinen, vaikuttaisi siltä, että Java-virtuaalikone on suunniteltu nimenomaan Java-ohjelmointikieltä varten. Jotkin tutkimukset näyttävät, että tällä on kustannuksia muiden kielten tehokkuuteen [11] [12]. Täysin geneerisen virtuaalikoneen suunnitleminen ei ole helppoa.

Jos virtuaalikone luodaan geneeriseksi ja alustariippumattomaksi, niin virtuaalikoneelle voidaan toteuttaa useita eri ohjelmointikieliä. Virtuaalikoneen alustariippumattomuudesta johtuen nämä kielet tukevat alustoja, joilla virtuaalikoneen koodin suoritus on toteutettu.

2.1.2 Tulkkauksen tehokkuus

Mittauksessa [1] on kirjoitettu yksinkertainen laskentatehoa mittaava ohjelma. Ohjelma laskee Mandelbrot-joukon arvoja. Tämän on hyvin perinteinen lakennallisesti vaativa tehtävä, jota käytetään tehon mittauksissa. Tämä ohjelma on kirjoitettu useilla kielillä, ja näiden kielten eri toteutuksilla. Koodi on kirjoitettu hyvin jokapäiväiseen tyyliin, menemättä syvällisemmin optimoimiseen. Kuvassa 1 on esitetty graafisesti tämän mittauksen tulokset.



Kuva 1. Mittauksen [1] tuloksia graafisesti.

Kun testissä testattuja toteutuksia tutkitaan tarkemmin, niin havaitaan, että toteutukset, jotka suorittavat operaatiot prosessorilla, ovat huomattavasti tulkkeja nopeampia. Näiden kielten toteutustavat ollaan listattu tässä.

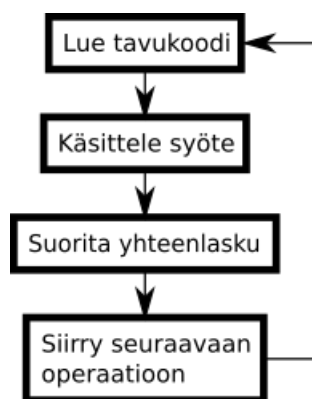
- CPython kääntää kielen tavukoodiksi virtuaalikoneelle, ja tulkitsee tätä [13].
- Cython kääntää kielen C-kielillä luoduksi lähdekoodiksi [14], ja tämä lähdekoodi voidaan sitten kääntää prosessorille [15].
- PyPy sisältää ajonaikaisen kääntäjän [16].
- ocamlc kääntää kielen tavukoodiksi, ja ocamlrun tulkaa tavukoodia [17].
- ocamlpt kääntää kielen prosessorille [18].
- Ruby:n normaali toteutus tulkaa kieltä. [19]
- Vaihtoehtoinen GRaalVM toteutus Ruby:lle kääntää koodin prosessorille. [20]

Puhtaasti prosessorilla ajamisesta saatava tehokkuuslisä ei ole pieni. Tehokkuuslisä, oli OCaml-kielille 10-kertainen, ja Python-kielille 45-kertainen. Tässä kuvassa käännetty Cython-toteutus on hyvin hidas johtuen siitä, että Python-kielen dynaaminen tyyppitys on aiheuttanut vaikeuksia

Cython-toteutukselle [21]. Tämä on esimerkki siitä, että jotkin tulkattavaksi suunnitellun kielen ominaisuudet voivat aiheuttaa vaikeuksia, kun kieltä käännetään.

2.1.3 Miten tulkki toimii

Tulkkaus on mittauksen [1] mukaan huomattavasti hitaampaa verrattuna koodin suorittamiseen prosessorilla. Tälle löytyy syy. Tulkki on itse ohjelma, joka suoritetaan oikealla prosessorilla. Tulkki-ohjelma suorittaa operaatiot ohjelmallisesti sen sijaan, että operaatiot suoritettaisiin puhtaasti prosessorilla. Tällä on luonnollinen tehokkuusongelma, mille ei ole tunnettua ratkaisua. Mietitään tilannetta, jossa tulkkia käytetään tulkkaamaan tavukoodia, joka kuvaa kahta peräkkäistä yhteenlaskua. Periaatteeltaan tulkkiohjelma toimii kuvassa 2 esitetyn silmukan mukaan. [4]



Kuva 2. Tulkin periaatteellinen toiminta.

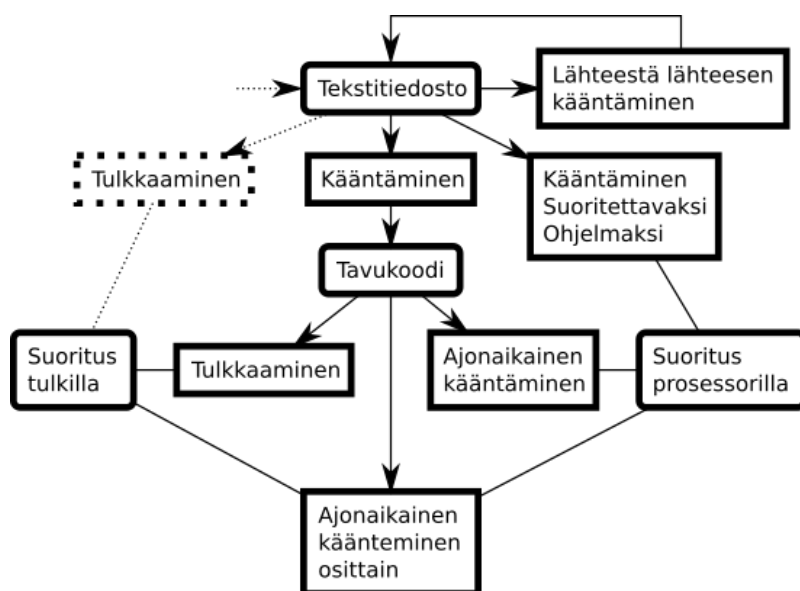
Näistä tulkin suorittamista operaatioista ainoastaan yhteenlasku on hyödyllinen ohjelman toiminnan kannalta. Tavukoodin lukeminen ja käsittely eivät mitenkään liity itse yhteenlaskun suorittamiseen, ja näin ollen nämä ovat hyödyttömiä operaatioita. Koska tulkkiohjelma on (yleensä) oikealla prosessorilla ajettava ohjelma, niin kaikki nämä tulkin suorittamat operaatiot tarkoittavat operaatioita prosessorilla. Prosessori myös sisältää kaiken tämän logiikan [22], joten tulkin logiikka tämän päällä on hyödytöntä. Tulkki on siis suorituksen kannalta turha kerros prosessorin päällä. Hyödyllisen ja hyödyttömän logiikan suhde voi jäädä todella huonoksi. [4]

Tulkin hyödyttömät operaatiot voidaan suorittaa yllättävän tehokkaasti käyttämällä kääntäjän tarjoamia lisäosia, mutta tulkin tehokkuus ei silti vieläkaan pääse lähellekään sitä, kuin prosessorille annettaisiin vain kaksi yhteenlaskuohjetta peräkkäin. [3] [4] [6] [23]

2.2 Ohjelmointikielen toteutuksen tavat

Toisin kuin operaatioiden suoritus, ohjelmointikielen koko toteutus voi tapahtua hyvin monella tapaa. Ohjelmointikieltä voidaan joko tulkata, tai se voidaan jollakin tapaa kääntää. Kääntäjä on ohjelma, joka kääntää kielen lähdekielestä kohdekieleksi [3] [24]. Kääntäjä voi esimerkiksi kääntää kielen suoritettavaksi ohjelmaksi, tai tavukoodiksi virtuaalikoneelle. Kääntäjä voi myös kääntää kielen toisen kielen lähdekoodiksi. Tätä kutsutaan lähteestä lähteeseen kääntämiseksi [3]. Jos kieli käännetään esimerkiksi C-kielellä tehdyksi koodiksi, niin kieli saadaan helposti suoritettua prosessorilla.

Yleisiä tapoja toteuttaa ohjelmointikieli on esitetty kuvassa 3. Tämä kuva ei kerro kaikkia mahdollisia tapoja toteuttaa kieli, vaan ainoastaan yleisimmät käytetyt tavat. Kuvassa on myös esitetty, että suorittaako tämä toteutus operaatiot prosessorilla, vai tulkkamalla.



Kuva 3. Yleiset tavat toteuttaa kieli.

Kuvasta voidaan havaita monet yleiset kielten toteutukset.

- Kielen kääntäminen tavukoodiksi, sen tulkkaus, ja ajonaikainen kääntäminen. Esim. HotSpot-JVM. [6] [25]
- Kielen kääntäminen tavukoodiksi, ja sen tulkkaus. Esim. CPython-toteutus Python-ohjelmointikielelle. [13]
- Kielen kääntäminen tekstitiedostosta suoritettavaksi ohjelmaksi. Esim. GCC (GNU Compiler Collection). [15]

Projektissa kieli halutaan toteuttaa niin, että sitä suoritetaan prosessorilla. Tämän toteutukseen on ehdolla erityyppisiä kääntäjiä.

Kieli voidaan kääntää lähteestä lähteeseen -kääntäjällä. Lähteestä lähteeseen -kääntäjä on kääntäjä, jonka lähdekielenä ja kohdekielenä on jollakin ohjelmointikielellä luotu lähdekoodi [3] [15] [26] [27]. Kohdekielenä jokin kieli, joka kääntyy konekieleksi, on mielenkiintoinen valinta. Jos käytetään jotakin kieltä, millä on tehokas pitkälle kehitetty toteutus, niin silloin tätä tehokkuutta siirtyy myös lähdekielelle. C-kieli on hyvä esimerkki kohdekielestä. C on yksinkertainen kieli, jolla ei ole mitään monimutkaisia taustalla olevia järjestelmiä, kuten automaattista muistin vapautusta tai roskien keruuta [28]. Omaa kieltä tehtäessä taustajärjestelmät voidaan toteuttaa itse niin kuin tarve vaatii. Lähteestä lähteeseen - kääntäjä voi myös toimia koodigeneraattorina niin, että osa jotain kohdekielellä tehtyä ohjelmaa on generoitu kääntäjällä. Tämä on helppo, monikäyttöinen ja tehokas toteutus kielelle. Tämä on aliarvostettu tapa toteuttaa kieli.

Kieli voidaan kääntää ennen sen suoritusta. Kääntäjää, joka kääntää ohjelman ennen sen suoritusta, kutsutaan ennaltakääntäjäksi, tai AOT (Ahead of Time) -kääntäjäksi [3] [29]. AOT-kääntäjä tunnetaan parhaiten siitä, että se yleensä kääntää ohjelmointikielen suoritettavaksi ohjelmätiedostoksi. Esimerkiksi Windows-ympäristössä tämä tarkoittaa .exe-päätteistä tiedostoa. Esimerkkinä AOT-kääntäjästä voisi olla hyvin suositut GCC:n (GNU Compiler Collection) C- ja C++-kääntäjät [15]. AOT-kääntäjä voi myös kääntää koodia ennalta käännettyksi jaetuksi kirjastoksi, jota voidaan sitten käyttää ohjelmissa, ilman että kirjaston koodia täytyy kääntää aina uudestaan [30] [31].

Kieli voidaan myös kääntää ajonaikaisella kääntäjällä, tai JIT (Just In Time) -kääntäjällä. JIT-kääntäjä tunnetaan parhaiten siitä, että se voidaan lisätä tulkkiin niin, että se kääntää jonkin usein ajetun palan koodia konekoodiksi, jonka tulkki sitten voi suorittaa normaalin tulkkauksen sijaan.

Tämänlaisella kääntämisellä tulkin vaatimat turhat operaatiot poistuvat ohjelmasta. Ajonaikainen kääntäjä voi nostaa ohjelman tehokkuutta huomattavasti. JIT-kääntäjä voi myös kääntää koko ohjelman ajettavaksi koodiksi ilman tarvetta tulkille. [3] [24]

Esimerkkinä JIT-kääntäjästä on HotSpot -Java virtuaalikone (JVM). HotSpot on periaatteeltaan tulkki, joka tulkaa sille annettua tavukoodia. HotSpot ei kuitenkaan ainoastaan tulkaa tavukoodia, vaan voi myös kääntää tätä koodia ohjelman suorituksen aikaisesti eteenpäin oikealle prosessorille. [6] [25]

Toisin kuin AOT-kääntäjän koodi, JIT-kääntäjän koodi ajetaan jo olemassa olevan prosessin virtuaalisessa muistissa. Esimerkiksi jos JIT-kääntäjän toteuttava tulkki ohjelmaa ajonaikaisesti, niin tulkki varastoi käännetyn koodipalan dynaamisesti varattuun muistiin, ja siirtää suorituksen tähän koodiin. Tämän huono puoli on se, että JIT-kääntäjän koodin on pystyttävä toimimaan yhdessä sen kääntämiseen käytetyn kääntäjän koodin kanssa. Tällä on kuitenkin merkittävä hyvä puoli, JIT-kääntäjän luoma koodi pystyy helposti käyttämään suurta määrää sen luontokielellä tehtyjä kirjastoja. JIT-kääntäjän luomalla koodilla on mahdollisuus kutsua JIT-kääntäjän koodiin mukaan rakennettuja funktioita, ja näin ollen käyttämään kääntäjän koodiin sisällettyjä kirjastoja. Tämä toiminnallisuus on toteutettu tässä projektissa. [24] [32] [33] [34]

JIT-kääntäjällä on myös toinen hyvä puoli. Koska JIT-kääntäjä ajetaan tietokoneella, jolla ohjelma lopuksi ajetaan, niin JIT-kääntäjä voi huoletta käyttää uusia prosessorin ominaisuuksia, jotka ovat saatavilla vain pienellä määrällä laitteita. AOT-kääntäjää käytettäessä ei välttämättä tiedetä lopullista kohdekonetta, joten silloin näitä ominaisuuksia on käytettävä varauksella. JIT-kääntäjällä tosin on ongelma. Koska JIT-kääntäjä suoritetaan, kun ohjelma halutaan suorittaa, niin JIT-kääntäjän pitäisi selviytyä käännoستهتävistä nopeasti. AOT-kääntäjälle tämä ei päde. [3]

3 Motivaatio luodulle kääntäjälle

Kun ohjelmointikielen mahdollisia toteutustapoja tutkittiin, niin päätös tehtiin tarkasta kääntäjän toteutustavasta tähän tutkimukseen perustuen. Tutkiessa havaittiin, että operaatioiden suoritus prosessorilla on huomattavasti tulkkausta nopeampaa. Koska kielen nopeus oli tärkeää, niin projektissa päätettiin, että luotu kieli tulee suorittaa prosessorilla tulkkauksen sijaan.

Prossessorilla suoritus voi tapahtua monella tapaa. Tutkimuksesta kävi ilmi kolme hyvää tapaa toteuttaa prosessorilla suoritus. Nämä tavat ovat lähteestä lähteeseen -kääntäminen, AOT-kääntäminen ja JIT-kääntäminen. Lähteestä lähteeseen kääntäminen tarjoaa helpon ja voimakkaan tavan toteuttaa kieli. Kuitenkin kiinnostusta ja halua löytyi kääntäjän kirjoittamiseen alusta loppuun, joten vaihtoehtoina ovat AOT-kääntäminen suoritettavaksi ohjelmaksi, ja JIT-kääntäminen.

JIT-kääntäminen on näistä pääasiassa parempi vaihtoehto. Tämä toteutustapa toimii erityisen hyvin sen kanssa, jos ohjelmointikieli käännetään ensin virtuaalikoneelle. Virtuaalikoneeseen voidaan lisätä ominaisuus, jolla pystytään kutsumaan JIT-kääntäjän ohjelmaan mukaan rakennettuja C++-kielellä luotuja funktioita. Tämä ominaisuus voidaan helposti toteuttaa JIT-kääntäjään, mutta käyttämällä AOT-kääntäjää, tämän toiminnallisuuden toteuttaminen on huomattavasti vaikeampaa. Koska kääntäjä luodaan C++-kielellä, niin virtuaalikoneeseen voidaan toteuttaa suuri määrä ominaisuuksia, käyttämällä laajaa valikoimaa kirjastoja, jotka ovat saatavilla C++-kielelle. Rakentamalla rajapinta jollekin kirjastolle virtuaalikoneeseen, saadaan tämä toiminnallisuus virtuaalikoneelle käännettävälle ohjelmointikielelle käytännössä ilmaiseksi. JIT-kääntäjällä on myös teoreettinen etu, että pitkälle kehitettynä se pystyy optimoimaan käännetyn koodin sen ajaneelle laitteelle. JIT-kääntäjä voi huoletta käyttää uusia prosessorin ominaisuuksia, jotka ovat saatavilla vain hyvin harvoilla laitteilla. Näiden ominaisuuksien käyttö on jätettävä pois AOT-kääntäjiltä ainakin oletuksena. Projektissa päätettiin, että kääntäminen toteutetaan JIT-kääntäjällä.

Hyvin suunnitellulle virtuaalikoneelle on helppo kääntää ohjelmia, ja virtuaalikoneen ohjelmia on nopea tulkata. Virtuaalikoneelle kääntämisen helppouden ansiosta virtuaalikone on hyvä valinta rajapinnaksi käyttäjälle, joka haluaa toteuttaa ohjelmointikielen. Näin käyttäjä voi luoda kääntämistyyppisen toteutuksen ohjelmointikielelle, ilman tietämystä oikeiden prosessorien toiminnasta. Kun kääntäjälle halutaan toteuttaa optimointeja, niin useita optimointeja voidaan toteuttaa virtuaalikoneen tasolla. Nämä optimoinnit vaikuttavat virtuaalikoneelle käännettyyn kieleen, ilman tarvetta käyttäjän huomiolle. Virtuaalikonetta, ja siihen liittyvää JIT-kääntäjää voidaan siis käyttää uudestaan muillekin kielille.

4 Tavoitteet luodulle kääntäjälle

Osassa 3 esitettiin projektin motivaatio perustuen tutkinnan tuloksiin. Tästä motivaatiosta kehitetty tavoitteet projektille. Osassa 3 päätettiin, että kieli käännetään ensin virtuaalikoneelle, ja virtuaalikoneen ohjelmat eteenpäin prosessorille ajonaikaisesti. Työssä päätettiin luoda oma virtuaalikone.

Virtuaalikoneelle päätavoitteeksi asetettiin kääntäjän väliformaattina toimiminen, ja se, että virtuaalikone olisi riippumaton kielestään. Toissijaiseksi tavoitteeksi asetettiin alustariippumattomuus. Kieliriippumattomuus on tärkeää projektin kannalta, koska silloin projektissa luotavalle kielelle voidaan helpommin rakentaa prototyyppejä. Virtuaalikoneen tarkoituksena on siis toimia alustana eri ohjelmointikielien toteutuksille. Virtuaalikoneen tarjoamaa alustaa suunniteltaessa sen helppokäyttöisyys, ja kyky abstraktoida oma alustansa (prosessori), ovat merkkejä virtuaalikoneen hyvästä suunnittelusta. Hyvin lähellä näitä asioita on virtuaalikoneen alustariippumattomuus. Tämä asetettiin toissijaiseksi tavoitteeksi, koska myös se on merkki hyvästä suunnittelusta. Projektissa ei ole suurempia suunnitelmia tukea muuta, kuin x86-prosessoria, mutta alustariippumattomuus on virtuaalikoneelle luonnollista, ja se saadaan helposti toteutettua muita tärkeitä tavoitteita kehittäessä. Virtuaalikoneen tavukoodin tulkkaukseen ei koettu tarpeelliseksi, joten tulkkia ei tehty tässä projektissa. Myöskään virtuaalikoneen ohjelmien tallentamista levyille ei koettu vielä tässä vaiheessa tarpeelliseksi. Tästä syystä virtuaalikoneen ”tavukoodille” ei luotu mitään tallennusmuotoa, vaan virtuaalikoneen ohjelmia kuvataan ainoastaan joukolla luokkia.

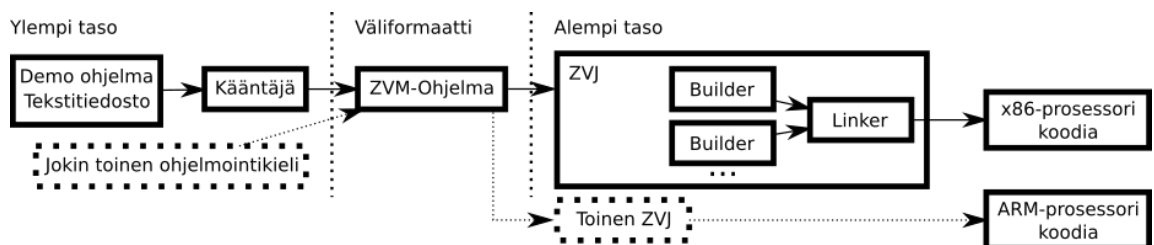
Projektin alkuperäisenä ideana oli kehittää pätevä ohjelmointikieli, mutta projekti on jo tällä laajuudella liian suuri opinnäytetyöksi. Tästä syystä ohjelmointikielelle asetettiin yksinkertainen tavoite, että sillä pitäisi pystyä luomaan ohjelma.

Kaikki projektissa kehitettävät osa-alueet on rajattu niin, että työssä ei toteuteta mitään muuta toiminnallisuutta kuin se, mikä on pakollista työssä luotavan yksinkertaisen ohjelmointikielen kääntämiseen.

5 Yleiskatsaus luodusta kääntäjästä

Projektissa luotu ohjelmointikieli käännetään ensin virtuaalikoneen ohjelmaksi, ja tämä virtuaalikoneen ohjelma käännetään eteenpäin x86-prosessorille. Virtuaalikonetta kutsutaan nimellä ZVM (Z Virtual Machine). Kääntäjää, joka kääntää virtuaalikoneen ohjelmat prosessorille, kutsutaan nimellä ZVJ (ZVM JIT). Ohjelmointikieli luotiin esittelemään, että virtuaalikoneelle voidaan toteuttaa ohjelmointikieliä. Tämä on kielen ainut tavoite. Tavoitteensa mukaisesti, kieltä kutsutaan nimellä ”Demokieli”.

Kääntäjän rakenne, ja käännösprosessi on esitetty kuvassa 4. Kuvassa kiinteällä viivalla näkyvät toteutetut järjestelmät, ja pisteillä kuvitteelliset.



Kuva 4. Käännösprosessi.

Kuvassa on eroteltu kääntäjäkokonaisuuden ylempi ja alempi taso. Ylempi taso koostuu demokie- len kääntäjästä, ja alempi taso virtuaalikoneen ajonaikaisesta kääntäjästä. Näiden tasojen väliin on vedetty katkoviiva. Tämä tarkoittaa abstraktiokerrosta. Demokie- len kääntäjä ei ole millään as- teella tietoinen ZVJ:stä, eikä ZVJ ole mitenkään tietoinen, mikä kieli virtuaalikoneelle alun perin käännettiin. Näitä komponentteja käsitellään tässä dokumentissa täysin erillään toisistaan. Kom- ponenttien käsittelyosat puhuvat vain kyseisen komponentin suorittamasta tehtävästä.

Virtuaalikonetta käytetään kääntäjän väliformaattina. Virtuaalikoneen ohjelmaa ei tulkata mil- lään tapaa. Ainut tapa suorittaa virtuaalikoneen ohjelmia on kääntämällä ne prosessorille. Virtu- aalikoneella on tavoite, että sitä pystyttäisiin käyttämään alustana, jolle voi toteuttaa useita eri ohjelmointikieliä. Virtuaalikoneen käyttämisenä alustana suuria etuja.

- Tarpeeksi geneeriselle virtuaalikoneelle toteutetut kielet tukevat alustoja, joilla virtuaalikoneen koodin suoritus on tuettu.
- Virtuaalikoneelle voi olla helppo toteuttaa kieliä, joten virtuaalikoneesta voidaan saada helppo ja tehokas toteutus.
- Kääntäjä voi suorittaa optimointeja ohjelmalle virtuaalikoneen tasolla. Nämä vaikuttavat ohjelmointikielen tehoon ilman käyttäjän toimenpiteitä.

Virtuaalikone, ja sen toiminnot on käsitelty tarkemmin osassa 6.

Virtuaalikoneen ohjelmat voidaan teoriassa kääntää useille eri alustoille, tai prosessoreille. Tässä työssä on luotu ainoastaan yksi kääntäjä kääntämään ohjelmat prosessorille. Kääntäjää kutsutaan nimellä "ZVJ" (ZVM JIT, tai matalan tason kääntäjä). ZVJ kääntää ohjelmat x86-prosessorille Linux-ympäristössä. ZVJ:n on mahdollista toteuttaa monia alustastaan riippuvaisia optimointeja. Nämä optimoinnit voivat vaikuttaa virtuaalikoneelle toteutettuun kieleen ilman, että virtuaalikoneen käyttäjän tarvitsee ryhtyä toimenpiteisiin. Kuitenkaan mitään optimointeja ei olla vielä toteutettu. ZVJ:n toimintaa on käsitelty osassa 7.

6 Virtuaalikone

Virtuaalikonetta käytetään kääntäjän väliformaattina. Virtuaalikonetta kutsutaan nimellä ZVM (Z Virtual Machine). Demokieli käännetään ensin ZVM-ohjelmaksi, ja ZVM-ohjelma sitten eteenpäin prosessorille.

ZVM toimii puhtaasti kääntäjän väliformaattina. ZVM ohjelmia ei tulkata ollenkaan, eikä mitään suoritukseen liittyviä komponentteja olla tehty tässä projektissa. Ainut tapa suorittaa ZVM ohjelma, on kääntää se eteenpäin prosessorille. Vain ZVM ohjelmista on kuvaus, joka on toteutettu joukolla luokkia. ZVM ohjelmia ei tallenneta levyille millään tapaa.

ZVM muistuttaa hieman oikeaa prosessoria. ZVM:llä on kuvitteellinen käskyjoukko. Käskyihin kuuluvat esimerkiksi kahden muuttujan yhteenlasku, muuttujien vertailu, ja johonkin kohtaan hyppääminen ohjelmassa. Tällä hetkellä ZVM-ohjelmat koostuvat vain joukosta käskyjä, eikä mu- kana ole tietoa missään muussa muodossa.

Kuvassa 5 on esitetty testikoodi, joka luo ZVM-ohjelman, ja tämä ohjelma on käännetty eteenpäin prosessorille käyttämällä matalan tason kääntäjää.

```

zvm_program p;
zvm_glblm glm;
zvm_proginstraccessor b(glm, p);

zvm_glblm::label l = glm.new_();

b.pi(instr::scopepush());
b.pi(instr::alloct(1, zvm_type::t_u64()));
b.pi(instr::alloct(1, zvm_type::t_u64()));
b.pi(instr::alloct(1, zvm_type::t_u64()));
b.pi(instr::set(1, data(zvvar(0)), data(imm::_u64(0))));
b.pi(instr::set(1, data(zvvar(1)), data(imm::_u64(0))));
b.pi(instr::set(1, data(zvvar(2)), data(imm::_u64(0))));

b.label_bind(l);

b.pi(instr::add(1, data(zvvar(0)), data(zvvar(0)), data(zvvar(2))));
b.pi(instr::add(1, data(zvvar(2)), data(zvvar(2)), data(imm::_u64(3))));
b.pi(instr::add(1, data(zvvar(1)), data(zvvar(1)), data(imm::_u64(1))));

b.pi(instr::compare(data(zvvar(1)), data(imm::_u64(10))));
b.pi(instr::branch(l, cndtype::NEQ));

b.pi(instr::progexit(data(zvvar(0)))); //This pops the last scope.

zvj2 jit(glm);
jit.load(p);
zvj_program prog(jit.compile());

u64 (*fn)() = prog.get_fn();
u64 res = fn();
BOOST_VERIFY(res == 135);

```

Kuva 5. Testikoodi, joka luo ja kääntää ZVM-ohjelman.

ZVM:llä on tavoite, että se olisi helppo ja tehokas alustariippumaton alusta, jolle voitaisiin toteuttaa useita ohjelmointikieliä. Alustariippumattomuus tarkoittaa sitä, että ZVM:lle toteutettujen kielten pitäisi tukea alustoja, joilla ZVM ohjelmien suoritus on toteutettu. Tarkoitustaan palvellen ZVM ja sen käskyt on suunniteltu niin, että sille mahdollisimman helppo kääntää ohjelmia. ZVM toimii abstraktiokerroksena alustalleen, eikä paljasta alustan ominaisuuksia käyttäjälleen. Kuitenkin tällä hetkellä ZVM-ohjelmat voivat käsitellä muistia suoraan, mikä on alustariippuvaista.

Monet käännettävän ohjelman optimoinnit voidaan toteuttaa helposti tällä väliformaattitasolla, mutta yhtäkään optimointia ei ole vielä toteutettu. Optimoinnit, jotka toteutetaan ZVM-tasolla, eivät vaadi huomiota ZVM:n käyttäjältä.

Tässä osassa käsitellään pääasiassa itse ZVM-ohjelmia, formaattia ja ominaisuuksia, joita ZVM tukee, eikä näiden toteutusta. Matalan tason kääntäjä kääntää ZVM-ohjelmat prosessorille, ja kaikkien ominaisuuksien toteutuksesta puhutaan tarkemmin tämän käsittelyssä.

6.1 Muuttujat

Muuttuja on yksittäinen numeerinen arvo, jota ZVM-ohjelman käskyt voi muuttaa ja käyttää. Muuttujia käytetään varastoimaan operaatioiden tuloksia. ZVM-ohjelman käskyt viittaavat muuttujiin numeerisella tunnisteella.

Muuttujia voi olla useita eri numeerisia tyyppejä. Näitä tyyppejä ovat erikokoiset kokonais- ja luituluvut. Tosin tällä hetkellä vain 64 bitin kokonaisluku on tuettu. Jos käännettävässä kielessä on luokkia tai muita asioita, joilla on useita jäsenmuuttujia, niin tämän kielen kääntäjä luo toteutuksen näille luokille.

Muuttujat luodaan käskyillä. Ensin annetaan SCOPEPUSH-käsky, joka luo näkvyysalueen. Tämän käskyn perään laitetaan yksi tai useampi ALLOCT-käsky, joka luo yhden muuttujan. ALLOCT-käskyllä luodut muuttujat kuuluvat luotuun näkvyysalueeseen. Muuttujien numeeriset tunnisteet vastaavat järjestystä, missä ALLOCT-käskyt annettiin. Uudessa SCOPEPUSH-käskyllä luodussa näkvyysalueessa muuttujien tunnisteet alkavat nollasta. SCOPEPUSH-käskyä seuraava koodi viittaa tunnisteilla näkvyysalueen muuttujiin. On olemassa myös SCOPEPOP-käsky, mikä on vastakohta SCOPEPUSH-käskylle. Tämän käskyn jälkeen seuraava koodi viittaa taas edellisen näkvyysalueen muuttujiin. Kuvassa 5 on luotu kolme kokonaislukumuuttujaa, ja näitä on käytetty silmukassa.

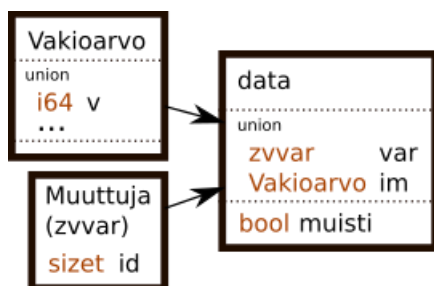
Muuttujien luominen tällä tavalla ei ole kovin järkevää, varsinkin kun mietitään funktioiden toteutusta. Tämänlainen muuttujien käsittely sallii hyvin erikoisia ja tarpeettomia ominaisuuksia, joiden toteutukseen tarvitaan taustajärjestelmiä lopulliseen ohjelmaan. Tälle muuttujien käsittelylle on esitetty parempi ratkaisu sisäisten funktioiden toteutuksen käsittelyssä.

6.2 Käsyt

ZVM-ohjelmat koostuvat käskyistä. Käskyt ovat yksinkertaisia operaatioita, kuten esimerkiksi kahden numeron lisääminen tai vähentäminen. Käskyt voivat ottaa tietoa sisään ja antaa sitä ulos. Käskyt voivat myös vaikuttaa ohjelman suoritukseen. Suoritukseen vaikuttavia käskyjä käytetään toteuttamaan ehtolauseet ja silmukat käännettävään kieleen.

ZVM-käskyt voivat hyväksyä eri määriä erilaista tietoa parametreiksi samaan tapaan kuin x86-prosessorien ohjeet ottavat operandeja. Tiedon välitys käskyille tapahtuu pääasiassa parametreilla, mutta erikoistapauksissa käskyt voivat myös lukea ennalta asetettuja lippuja. Tulevaisuudessa tarkoitus olisi, että kaikki käskyt käyttäisivät ainoastaan parametreja tiedon välitykseen. Yksittäiset parametrit voivat olla sisääntuloja, ulostuloja, tai kumpiakin samaan aikaan.

Käskyt voivat periaatteessa ottaa parametreiksi ihan kaiken tyyppistä tietoa, mutta yleisimmät käytetyt parametrin tyypit ovat "data" ja "zvvar". "zvvar"-tyypin parametri tarkoittaa muuttujan tunnustetta. "data"-tyypin parametri voi olla joko muuttujan tunniste tai vakioarvo. Parametreina käytetyt vakioarvot voivat olla tyypiltään joitakin ZVM:n tukemia tyyppisiä, eli kokonaislukuja tai tulevaisuudessa liukulukuja. "data"-parametrilla on myös lippu, joka kertoo, että viittaako tämä muuttuja tai vakioarvo johonkin muistiosoitteeseen. Kun lippu on asetettu, niin tässä muistissa olevaa arvoa pitäisi käyttää itse parametrin arvon sijaan. Tämä rakenne on esitetty kuvassa 6.



Kuva 6. Yleiset parametrien tyypit.

"data"-parametri noudattaa ideaa, että se viittaa mihin tahansa asiaan, josta voidaan lukea tietoa. "zvvar"-parametri ei vastaa mitään ideaa yhtä hyvin. Tämä on lievä ongelma. Tästä puhutaan myöhemmin, kun tulevaisuuden ideoita käsitellään.

ZVM-ohjelmien kuvaus tai käskyjoukko on melko vanha idea, ja siinä on useita ongelmia. Ongelmia käsitellään sisäisten funktioiden toteutusta pohtivassa osassa. ZVM:n kehittäminen on yksi seuraavista projektin tehtävistä.

Kaikki projektissa käytetyt käskyt, ja niiden matalan tason kääntäjän käyttämä toteutustapa on mainittu liitteessä 1. Kaikkia käskyjä tai niiden muotoja ei ole vielä toteutettu projektin aikarajan takia. Vain projektissa tarvittavat käskyt ovat toteutettu.

6.3 Viittaukset

ZVM sisältää hyppykäskyn, joka voi hypätä johonkin kohtaan ohjelmassa. Tämä kohta pitää jotenkin kertoa käskylle. Tämä ratkaisu on kopioitu suoraan ZVJ:n builder-komponentissa käytettävästä viitejärjestelmästä. ZVM ei tällä hetkellä tarjoa edistyneitä ominaisuuksia, jotka tarvitsevat builderin monimutkaista viitejärjestelmää, mutta tämä ratkaisu oli hyvin nopea kopioida.

Viitejärjestelmä sisältää viitemerkin, viitemerkki noodin ZVM-ohjelmassa ja viittaustietokannan. Käyttäjä, joka luo ZVM-koodia, pyytää viitemerkki-instansseja viittaustietokannalta, ja myöhemmin rekisteröi viitemerkit johonkin kohtaan ZVM-ohjelmassa. Viitemerkit viittaavat kohtaan, johon viitemerkki on rekisteröity tai tullaan myöhemmin rekisteröimään. Ohjelman kohdan antaminen parametriksi ohjeelle toimii antamalla ohjeelle viitemerkki. Tämän viitejärjestelmän toiminnasta puhutaan tarkemmin builderin käsittelyssä.

6.4 Ulkoiset funktiot

Ulkoisilla funktioilla tarkoitetaan ZVM-ohjelman ulkopuolella määritettyjä funktioita. Tämä tarkoittaa C++-kielellä ohjelmoituja kääntäjän koodista löytyviä funktioita. ZVM tukee tämänlaisten funktioiden kutsua.

Ulkoiset funktiot voivat ottaa argumenteiksi tai palauttaa vain tyyppejä, joille on määritetty yhteys ZVM:n tyyppin ja C++:n tyyppin välillä. Esimerkiksi yhteys voidaan helposti luoda ZVM:n tuke-
man 64 bitin etumerkillisen kokonaisluvun ja C++:n `int64_t` välille.

Jokainen C++-kielellä luotu ulkoinen funktio ottaa aina parametriksi kaksi kokonaislukuargumenttia. Nämä ovat tulkittavissa osoittimiksi funktion argumentteihin ja paluuarvoihin. C++-koodi manuaalisesti purkaa halutut arvot näistä osoittimista. Parametrit ja paluuarvot on toteutettu osoittimilla, koska eri C++-kääntäjät ovat käyttäytyneet eri lailla funktiokutsujen ympärillä, ja muuta yleispätevää järkevää ratkaisua ei ole. Tästä puhutaan enemmän luvussa 6.1.4.

Ulkoisten funktioiden kutsua varten on CALLE-ohje. CALLE ottaa parametriksi tiedot funktiosta, muuttujan johon paluuarvo tallennetaan ja listan muuttujista, joista funktion argumentit ladataan. CALLE-ohjeessa on ongelma, funktion paluuarvona ei voi olla kuin yksi muuttuja. Tämä voidaan helposti korjata vaihtamalla paluuarvotkin listaksi. Useaa paluuarvoa ei ole tarvittu, joten tätä ei aikarajan takia korjattu.

Useita ulkoisia funktioita on määritetty. Näistä tässä projektissa ovat käytettyinä seuraavat funktiot, "vmalloc", "vmfree", "print" ja "printnumber".

- "vmalloc"- ja "vmfree"-funktioita käytetään varaamaan ja vapauttamaan muistia. Nämä on käsitelty muistinhallinnan käsittelyssä.
- "print"-funktio ottaa argumentiksi muistiosoitteen C-kielen tyyliseen merkkijonoon, joka loppuu 0-koodiseen merkkiin. "print" tulostaa annetun merkkijonon antamalla tämän osoitteen suoraan C++-kielen "std::cout"-oliolle.
- "printnumber" on toinen versio "print"-funktioista. "printnumber" ottaa parametriksi kokonaisluvun ja tulostaa tämän luvun konsolille. "printnumber" voidaan tietysti toteuttaa "print"-funktioilla, mutta erillisen funktion tekeminen oli nopeampaa. Tämä todennäköisesti poistetaan tulevaisuudessa.

6.5 Muistinhallinta

ZVM tukee muistinhallintaa. Ohjelma voi varata, käyttää ja vapauttaa muistia. Varattuun muistiin viitataan normaalina kokonaislukumuuttujana, jonka arvo vastaa jotakin muistiosoitetta. Eli ZVM ei erottele osoittimien ja kokonaislukujen välillä. "data"-tyypin ohjeiden parametri sisältää lipun, joka kertoo, että itse parametrin arvo viittaa muistiosoitteeseen, jossa haluttu arvo on.

Muistia varataan kutsumalla ulkoista "vmalloc"-funktioita. "vmalloc" ottaa parametriksi varattavan muistialueen koon ja palauttaa kokonaislukumuotoisen muistiosoitteen varattuun muistiin.

Tälle ulkoiselle funktiolle on myös vastakappale `vmfree`. `vmfree` ottaa parametrina kokonaisluvun ja vapauttaa sen merkitsemän muistiosoitteen. Nämä kaksi funktiota ovat suoria rajapintoja C-kielen `malloc` - ja `free` -funktioihin.

ZVM sallii muistin käyttämisen suoraan. Muistin suora käyttäminen on joissakin tapauksissa alustariippuvaista, joten ZVM:llä voidaan luoda koodia, jonka käyttäytyminen riippuu alustasta. Esimerkiksi alustariippuaiseen koodiin helposti johtava, C++:n `reinterpret_cast` voidaan toteuttaa käännettävässä kielessä. Jos ZVM halutaan täydellisen riippumattomaksi alustastaan, niin muistinhallinta on abstraktoitava. [35]

6.6 Sisäisten funktioiden toteutus

ZVM-ohjelman koodissa määritettyjä funktioita kutsutaan sisäisiksi funktioiksi. ZVM:n ohjelmat eivät voi määrittää funktioita niiden koodin sisällä. Ainoastaan C++-kielellä luotuja ZVM-ohjelmien ulkopuolella määritettyjä funktioita voidaan kutsua ZVM-ohjelman koodista.

6.6.1 Nykyisen muuttujien käsittelyn ongelmat

Nykyisellä virtuaalikoneella on hyvin epätehokasta toteuttaa funktioita. Ongelmia seuraa siitä, kuinka muuttujia käsitellään ZVM-ohjelmissa. Käytössä oleva idea on, että jokainen funktio luo oman näkyvyysalueensa `SCOPEPUSH`-käskyllä, ja muuttujien määrittelyt `ALLOCT`-käskyllä. Funktio ei tietenkään tarvitse näitä käskyjä käyttää, ja silloin tämä funktio viittaisi muuttujien tunnistella kutsujansa muuttujiin. Tällä ominaisuudella ei ole mitään järkevää käyttötarkoitusta. Nämä käskyt sallivat täysin mielivaltaisen käytössä olevien muuttujien vaihtamisen. Muuttujien käsittely tällä tavalla vaatii taustajärjestelmien upottamista lopulliseen ohjelmaan pitämään kirjaa käytössä olevista muuttujista.

6.6.2 Funktioiden ja muuttujien suhde

Yleisellä tasolla puhuen muuttujat voidaan toteuttaa prosessorilla niin, että niille varataan paikka pinomuistista. Kun funktion suoritus aloitetaan, niin funktio voi luoda pinomuistiin kehyksen, jossa jokaiselle funktiolle on päätetty tietty muuttumaton paikka. Jotta tämä toteutus olisi mahdollinen, niin funktion käytössä olevat muuttujat eivät voi vaihtua mielivaltaisesti, eikä funktio saa missään nimessä viitata mielivaltaisesti kutsujansa muuttujiin. ZVM sallii tehdä nämä asiat.

6.6.3 Rajapinta funktioille

Alkuperäisenä ideana ollut lähestymistapa oli, että ZVM-ohjelman sisälle luotaisiin useita funktioita. Parempi ratkaisu olisi, että ZVM-ohjelmaa ei olisi olemassa tässä muodossa, vaan sen sijaan olisikin ZVM-funktio, joka tarkoittaisi yhtä funktiota.

Funktion paikalliset muuttujat voitaisiin määrittää niin, että ZVM-funktion tietorakenteeseen liittäisiin ohjeista erilliseksi tiedoksi lista ohjelman käyttämistä paikallisista muuttujista. Tällä tavalla paikalliset muuttujat eivät voi vaihtua funktion sisällä, ja nämä voidaan toteuttaa tehokkaasti prosessorilla. Tämän muutoksen tekemällä huonoja SCOPEPUSH- ja ALLOCT-käskyjä ei enää käytettäisi määrittämään muuttujia, ja nämä käskyt saadaan poistettua.

Funktion argumentti- ja paluuarvomuuttujat tarvitsevat vielä kuvauksen. Tähän kuvaukseen kuuluu niiden olemassaolon määrittäminen, niiden käyttö koodissa, ja niiden lähettäminen kutsujan ja kutsutun välillä.

Funktion argumentti- ja paluuarvomuuttujien määrittäminen voi tapahtua samalla tavalla kuin paikallisten muuttujien. Käytetyt argumentit ja paluuarvot voidaan merkitä erillisinä listoina ZVM-funktion tietorakenteessa paikallisten muuttujien vieressä. Näin myös itse ”pääohjelma” voi sisältää argumentteja ja paluuarvoja ilman ylimääräisiä toimenpiteitä.

Argumentti- ja paluuarvomuuttujia voitaisiin käsitellä kutsutun funktion koodissa niin, että ”zvvar”-tyypin ohjeiden parametria laajennettaisiin sisältämään tiedon siitä, että viitataan paikalliseen, paluuarvo, vai argumenttimuuttujaan. ”data”-parametri voi olla tyypiltään ”zvvar”, joten muutokset tulevat myös tälle parametrille. ”zvvar”-tyypin laajentaminen sallisi kaikkien ohjeiden käsitellä näitä muuttujia.

Kutsuja ei voi järkevästi viitata kutsutun funktion paluuarvoihin "zvvar"-parametrilla. Ongelmia tulee, kun kutsuja kutsuu monta eri funktiota, ja vielä mahdollisesti ehdollisen koodin alla. Silloin pitäisi pitää kirjaa siitä, mitä funktiota kutsuttiin viimeksi, jotta oikeaa tietoa saadaan ladattua. Ilman suurempia muutoksia paluuarvojen tallentaminen kutsujan paikallisiin muuttujiin voisi tapahtua lisäämällä latauskartta funktioiden kutsuohjeeseen. Tämä kartta kartoittaisi paluuarvot kutsujan paikallisiin muuttujiin. Kutsuohje sitten suorittaisi nämä kartan määrittämät lataukset. Toteutuksen kannalta tämä ei ole kovin hyvä ratkaisu. Tällä ratkaisulla funktio ei voi kirjoittaa paluuarvoja suoraan kutsujan pinomuistissa sijaitseviin muuttujiin. Tämän ongelman korjaus on monimutkaisempaa.

Näillä ideoilla kutsuohjeesta muotoutuisi ohje, joka ottaa parametriksi ZVM-funktion, jota kutsutaan, kartan, joka kartoittaa kutsuttavan ohjelman paluuarvot kutsujan paikallisiin muuttujiin, ja "data"-parametreja sisältävän kartan, joka kertoo, mistä sijainneista argumentit ladataan. Tämä ohje on esitettävissä taulukolla seuraavasti.

ZVM-käsky	CALLI		
Parametrit	zvm_program fn	zvvar[] retmap	data[] params
Toiminto	retmap = fn(params)		

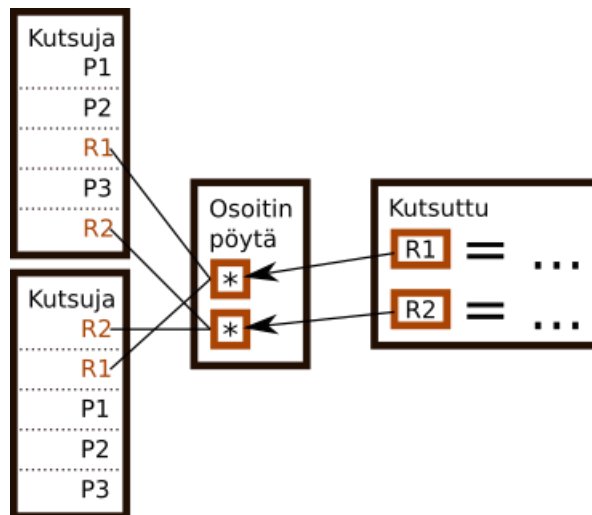
6.6.4 Toteutus rajapinnalle

Virtuaalikoneen ohjelmat käännetään prosessorille, niin sisäisten funktioiden toteuttaminen oikealla prosessorilla on huolenaihe.

Argumenttien toteutus toimii hyvin prosessorilla. Yksinkertaisin tapa on, että kaikki kutsijat luovat pinomuistiin aina samanmuotoisen kehyksen, jossa argumentit ovat kopioituna kutsujan paikallisesta muistista. Funktio voi sitten suoraan käyttää tätä kehystä argumenttimuuttujien tallennuspaikkana. Kutsutun funktion "zvvar"-parametri argumenttimuodossa voi viitata tähän kehykseen. Tässä ei ole mitään ongelmia.

Paluuarvoille yksi mahdollinen toteutus olisi luoda osoitinpöytä. Osoitinpöydässä on osoitin jokaiselle paluuarvomuuttujalle. Kutsuja asettaa osoittimen osoittamaan aikaan, mihin kutsuja haluaa paluuarvon tallennettavan. Kutsuttu funktio hakee pöydästä aina tietyssä sijainnissa olevan

osoittimen, johon funktio kirjoittaa paluuarvon. Näin kutsutun funktion kirjoitusoperaatiot saadaan kohdennettua suoraan kutsujan muistiin. Tämä on tehokkaampaa, kuin että funktiolle varattaisiin aina samanmuotoinen kehys, johon funktio kirjoittaa paluuarvon arvon ilman osoittimia. Tässä tapauksessa kutsujan pitäisi turhaan kopioida paluuarvot kehyksestä omiin paikallisiin muuttujiin. Kutsujen pinomuisti on eri järjestyksessä. Osoitinpöytäratkaisu on esitetty kuvassa 7. Kuvassa vasemmalla on kahden kutsujan pinomuistissa olevat muuttujat. Keskellä on kutsujen luoma osoitinpöytä.



Kuva 7. Paluuarvojen ratkaisu osoitinpöydällä.

Osoitinpöytä ei kuitenkaan ole täydellinen ratkaisu. Monimutkaisen osoitinpöydän luominen ja käsittely vaatii työtä. Tehokkaampaa olisi, että funktiolle annettaisiin ainoastaan yksi osoitin, joka osoittaa kutsujan paikalliseen muistiin, missä paluuarvojen muuttujat ovat aina samassa järjestyksessä. Näin kartoitusta ei tarvittaisi. Tämä vastaa ajatusta, että funktio palauttaisi vain yhden arvon, joka on tietorakenne. Tässä tapauksessa tietorakenne tarkoittaa joukkoa muuttujia, joiden asettelu muistissa on aina sama. Kaikki tämän joukon muuttujat varattaisiin kerralla.

ZVM ei tällä hetkellä tunnista tietorakenteen, tai muistin asettelun olemassaoloa, joten tämä ei ole mahdollista. Tietorakenteet voivat olla olemassa käännettävässä kielessä, mutta kun kieli käännettiin ZVM-funktioksi, niin tieto tietorakenteista hävisi. Tietorakenteiden olemassaolo auttaisi paljon myös tulevaisuudessa, joten tietorakenteet olisi hyvä luoda. Tietorakenteet toimisivat niin, että käyttäjä voisi luoda ja käyttää tätä tietotyyppinä ZVM:n ohjeissa. Koska ZVM:n käyttäjä itse käsittelee näitä tietorakenteita, niin tästä varmasti seuraa laajoja muutoksia muuttujien varamiseen ja käsittelyyn. Kuinka tämä rajapinta täsmällisesti toimii, ei vielä ole loppuun mietitty.

6.7 Virtuaalikone tulevaisuudessa

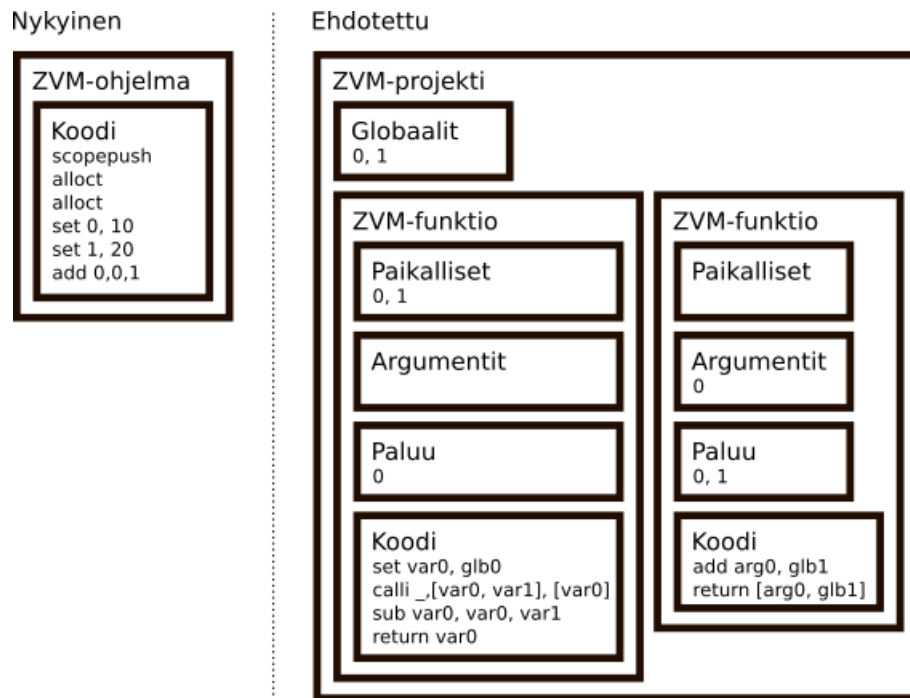
Välimuotoa on tarkoitus päivittää mahdollistamaan sisäiset funktiot. Tästä seuraa suurempia muutoksia. Tämän lisäksi pienempiä parannuksia ja siivoamista on paljon. Muutoksia tulee melkein joka paikkaan. Nämä muutokset yhdessä tarkoittavat täydellistä ZVM:n uudelleenkirjoitusta. On myös odotettavissa, että kun ZVM tulevaisuudessa mietitään uudestaan, niin silloin muutoksia tulee vielä lisää.

6.7.1 Ohjelman määrittäminen ja muuttujien luominen

Funktioiden määrittämisen pohdinnassa ehdotettiin, että ZVM-ohjelmaa ei kannattaisi olla olemassa tässä muodossa, vaan olisikin sen sijaan ZVM-funktio, joka vastaa yhtä funktiota ohjelmassa. ZVM-funktio sisältäisi käskyjen lisäksi tiedot paikallisista muuttujista, ohjelman argumenteista ja paluuarvoista. Muuttujien hallitsemiseen liittyvät käskyt poistetaan.

Kun useita ZVM-funktioita käytetään samassa ohjelmassa, niin globaalit muuttujat tulevat tarpeeseen, vaikka kieli ei tukisi niitä. Globaalit muuttujat tarkoittavat jotakin tiettyä joukkoa muuttujia, joita jokainen lopullisessa ohjelmassa käytetty ZVM-funktio pystyy käyttämään. Monien käytettyjen ZVM-funktioiden välillä on siis oltava globaalia jaettua dataa. Tämä voidaan toteuttaa luomalla käsite ZVM-projekti. Projekti sisältää kaikki lopullisessa ohjelmassa käytetyt funktiot, globaalit muuttujat, ja mahdollinen muu jaettu data.

Kuvassa 8 on esitetty nykyinen ZVM-ohjelman arkkitehtuuri, ja tätä verrataan uuteen ehdotettuun arkkitehtuuriin.



Kuva 8. Uusi ehdotettu ZVM-projekti verrattuna vanhaan ZVM-ohjelmaan.

6.7.2 Tietorakenteet

ZVM kuuluisi tunnistaa tietorakenteet. Tietorakenteet tässä tapauksessa tarkoittavat joukkoa muuttujia, joiden asettelu muistissa olisi taattu aina samaksi. Nykyään muuttujat varataan yksitellen sattumanvaraiseen järjestykseen. Tietorakenteet toimisivat eräänlaisena tietotyyppinä, jota voidaan käyttää ZVM:n ohjeissa. Tämä aiheuttaa laajoja muutoksia muuttujien käsittelyyn.

Tarve tietorakenteille tuli esille, kun mietittiin prosessorilla toteutusta ZVM-ohjelmassa määritettävälle funktioille. Kun näiden paluuarvojen käsittelyä mietittiin, niin havaittiin, että funktion paluuarvojen käsittely on epätehokasta nykyisellä virtuaalikoneella. Epätehokkaaksi ratkaisuksi paluuarvoille ehdotettiin, että jokainen funktion paluuarvo lähetettäisiin kutsutulle funktiolle osoittimen välityksellä, jotta kutsutun funktion operaatiot saataisiin kohdistettua suoraan kutsujan muistiin. Tämä on tehokkaampaa kuin paluuarvojen kopioiminen, mutta monimutkaisen osoitinpöydän käsittely voi aiheuttaa epätehokkuuksia. Parempi ratkaisu olisi lähettää kutsutulle funktiolle vain yksi osoitin kutsujan varaamaan tietorakenteeseen.

Tietorakenteita tarvitaan tulevaisuudessa myös kielellä määritettävien tietorakenteiden tehokkaaseen toteuttamiseen. Nykyinen mahdollinen toteutus kielellä määritettävälle tietorakenteille on purkaa tämä rakenne yksittäisiksi muuttujiksi. Jotta tähän tietorakenteeseen voitaisiin viitata

käännettävällä kielellä, niin sitä varten on luotava osoitinpöytä samaan tapaan kuin funktioiden kanssa. Jos ZVM tunnistaisi tietorakenteet, niin voitaisiin funktiolle lähettää vain yksi osoitin tähän tietorakenteeseen.

6.7.3 Ohjeiden parametrit

Ohjeiden parametrien tyypit eivät kuvasta käyttötarkoituksia kovin hyvin. Jotkin käskyt ottavat "data"-tyypin parametrin kirjoitustarkoitusta varten. Kirjoitustarkoituksessa "data"-tyypin parametri ei voi olla vakioarvo, jos parametri ei samalla myöskään viitata muistiin. Kirjoitustarkoituksessa "zvvar"-tyypin parametri taas ei voi viitata muistiin. Ehkä paremmat yleiset tyypit parametreille voisivat olla sisään- ja ulostulotyytit.

Tämän lisäksi funktioiden toteutusta miettiessä, "zvvar"-tyypin ohjeiden argumenttia laajennetaan kyvyllä viitata argumentti ja paluuarvomuuttujaan. Samalla havaittiin tarve tietorakenteille. Tietorakenteille ehdotetut ratkaisut todennäköisesti aiheuttavat muutoksia parametreissa. "zvvar"-tyypin kuuluisi myös pystyä viittaamaan globaaleihin muuttujiin. "zvvar"-tyyppi siis pystyisi viittaamaan argumenttimuuttujaan, paluuarvomuuttujaan, globaaliin muuttujaan ja paikalliseen muuttujaan.

6.7.4 Käskyjen puhtaus

ZVM-käskyjä on tarkoitus muuttaa niin, että ne olisivat puhtaita. Puhtaus tarkoittaa, että käskyjen toiminnot eivät riipu esimerkiksi mistään aikaisempien käskyjen asettamista lipuista, vaan ainoastaan tiedosta, jota käskyille itse syötettiin. Puhtaat käskyt eivät saa myöskään muuttaa ohjelman tilaa niille annettujen ulostulotyyppisten parametrien ulkopuolella.

Erityinen ongelmapaikka tällä hetkellä on COMPARE- ja BRANCH-ohjeet. COMPARE asettaa liput riippuen vertauksen tuloksesta, ja BRANCH lukee näitä. x86-prosessorilla hyppääminen toimii samaan tapaan ensin asettamalla liput CMP-ohjeella, ja tämän jälkeen lisätään jokin ehdollinen hyppyohje. ZVM:n on tarkoitus olla alustariippuvainen, joten se määrittää liput, jotka eivät ole suoraan yhteydessä prosessorin lippuihin. Ongelma on, että jos jollakin ZVM-käskyllä on toteutus, mikä asettaa prosessorin liput, niin silloin saatetaan kirjoittaa yli COMPARE-käskyn liput. Tämä

voi olla vastoin ohjelman määriteltä käyttäytymistä. Tämä on helposti ratkaistavissa niin, että COMPARE-käsky tallentaisi liput muuttujaan, ja BRANCH lukisi tätä muuttujaa.

Tällä hetkellä mikään muu ZVM-käsky kuin COMPARE, ei aseta prosessorin lippuja. Tästä syystä toteutus tällä hetkellä käyttää suoraan prosessorin lippuja. Tämä tulee muuttumaan tulevaisuudessa.

7 Matalan tason kääntäjä

ZVJ (ZVM JIT) on kääntäjä, joka on vastuussa ZVM-ohjelmien kääntämisestä x86-prosessorille käyväksi koodiksi, joka sitten suoritetaan. ZVJ siis ottaa sisään ZVM-ohjelman, ja antaa ulos x86-prosessorille käyvän tavujonon.

Jotta ZVM-ohjelma voidaan kääntää, niin on luotava yhteys kuvitteellisten ZVM:n käskyjen, ja oikean prosessorin koodin välillä. ZVM-käskyt voidaan enemmän tai vähemmän suoraan muuntaa x86-prosessorin ohjeiksi. Yhteys voidaan luoda helposti esimerkiksi ZVM:n ADD-käskyn ja x86-prosessorin samannimisen ohjeen välille. Toisen pään esimerkkinä ovat muuttujien luonti käskyt tai ohjelmasta poistumiseen tarkoitettu käsky PROGEXIT. PROGEXIT-käsky päättää ohjelman, ja palauttaa jonkin arvon. Näille ohjeille ei ole mitään suoraa vastinetta prosessorin puolella. Nämä ohjeet enemmänkin vaativat tietynlaisen ohjelman rakenteen, jotta nämä käskyt voidaan toteuttaa. Monet ZVM-käskyt voidaan myös toteuttaa joukolla prosessorin ohjeita, ilman muita toimenpiteitä. Monia ZVM-käskyjä ei ole toteutettu kunnolla aikarajan ja tulevaisuuden suunnitelmien takia. Vain ne toiminnallisuudet ovat toteutettu, mitä tässä projektissa tarvitaan demokieltä varten.

Muuttujien käsittelytavan siirros on suurempi ongelma, kuin yhteys ZVM:n ja prosessorin ohjeiden välillä. ZVM käsittelee tietoa muuttujina, kun taas x86-prosessori lataa tietoa muistista rajalliseen määrään rekisterejä, joissa tietoa käsitellään. ZVJ:n sisällä on rekisterivaraaja (VE), joka suorittaa tämän käsittelytavan vaihdoksen. VE toimii niin, että ensimmäiseksi ZVJ luo mallikoodin, ja sen jälkeen VE käsittelee tämän mallin. VE sijoittaa malliin muuttujien rekisterit, ja lataus- ja tallennusohjeita muuttujille. Kun malliin on sijoitettu muuttujien rekisterit ja ohjeet, niin koodi voidaan muuttujien puolesta kääntää. Tämän mallin olemassaolo vaatii tapaa käsitellä koodia abstraktissa muodossa, tämän toiminnallisuuden toteuttaa builder-komponentti. VE on tällä hetkellä ainut suuri järjestelmä, jota tarvitaan ZVM-ohjelmien kääntämiseen.

ZVJ käyttää builder-komponenttia kokoamaan ohjelmia x86-prosessorille. Builder pitää käskyt abstraktissa muodossa, ennen kuin ohjelmalle luodaan tavujono. Kun ohjeet ovat abstraktissa muodossa, niin ohjelmaa on tehokkaampaa muokata, kun että tavujonoja koodattaisiin uudestaan ja uudestaan. Abstraktin muodon lisäksi, esimerkiksi hyppykäskyjä varten tarvitaan tapa viitata johonkin kohtaan ohjelman koodissa. Builder tarjoaa myös tämän viitejärjestelmän.

7.1 Kääntäjän käyttäytyminen yleisesti

ZVJ kääntää ZVM-ohjelman funktioksi, jota voidaan kutsua C++-kielellä. Jotta käännetty koodi voi toimia kutsuttuna funktiona, niin ZVJ:n on oltava yhteisymmärryksessä C++-kääntäjän kanssa suuresta määrästä binääritason asioita. Tästä syystä ZVJ noudattaa Linuksilla käytössä olevaa ”SystemV AMD64 ABI” -binääritason standardia (sysvabi) [36] [37]. ZVJ luo koodia niin kuin muutkin kääntäjät yleensä luovat, ilman mitään erikoisia temppuja.

7.1.1 Muuttujat ja pinomuisti

RSP- ja RBP-rekisterejä käytetään viittaamaan pinomuistin nykyiseen huippuun, ja funktion kehukseen pinomuistissa vastaavasti. RSP-rekisteriä kuuluu aina käyttää viittaamaan pinomuistiin [38, s. 75], mutta RBP-rekisteriä ei ole pakko käyttää. Pinomuistin varaus toimii laskemalla RSP-rekisterin arvoa, koska pinomuisti kasvaa alaspäin [38, s. 151]. Pinomuistia käytetään varaamaan funktion kehys, joka sisältää funktion paikalliset muuttujat, ja mahdollisesti muuta tarvittavaa tietoa. Pinomuistin rakenne on esitetty taulukossa 1.

Taulukko 1. Pinomuistin rakenne funktion kehon alussa.

Tieto	Koko (tavua)	Osoitin	Kohdennus (tavua)
Paluuosoite	8 [38, s. 716]		
Vanha RBP	8		
Pinomuistiin upotettu tieto (käyttämätön)	0	RBP	16/32/64 (taattu). [37, s. 18]
Paikalliset muuttujat		RBP	16/32/64
Tallennetut rekisterit (sysvabi)	40		16
...	...	RSP	16

7.1.2 Segmentointi

ZVJ unohtaa segmentointi ominaisuudet kokonaan. Nykyään kun prosessori on 64-bittisessä tilassa, niin segmentointi ei ole käytössä ollenkaan [38, s. 73]. Segmentoinnin sijaan Linux käyttää virtuaalista muistia niin, että prosessille näkyy litteä osoiteavaruus [38, s. 2884] [39].

7.1.3 Binääritason standardien dokumentointi

ZVJ:n on mietittävä useita binääritason standardeihin liittyviä asioita. Näiden standardien dokumentointi on ollut puutteellista. Wikipedia mainitsee, että x86_64 ympäristössä Linux:illa binääritason standardi on ”System V AMD64 ABI” [40]. Myös Wikipedian lähde mainitsee, että tätä tietoa ei ole dokumentoitu, ja tieto perustuu kirjoittajan omiin kokeisiin [36]. Suoritin vielä omia kokeita, ja niihin perustuen standardi vaikuttaisi olevan oikein. Google-hakuihin ja hajanaisiin forumiposteihin perustuen sysvabi:sta on useita dokumentteja ja versioita. Foorumit mainitsevat, että yksikään näistä versioista ei täysin vastaa todellisuutta, kun kukaan ei ole pitänyt näitä dokumentteja ajan tasalla. Tämän lisäksi vielä ”epävirallisesti” joitakin eroavaisuuksia tapahtuu dokumenttien ja kääntäjien toiminnan välillä. Itse dokumentointi on vielä tämän lisäksi epämääräistä joissakin paikoissa. Omiin kokeisiin perustuen näissä paikoissa eri kääntäjien käyttäytymisellä on aikaisemmin ollut eroja. Näistä kääntäjän eroista on seurannut ongelmia.

7.1.4 Funktion kutsutapa

Funktion kutsutapa on binääritason standardin määrittämä asia. Standardin dokumentoinnin puutteellisuus on hieman aiheuttanut ongelmia.

Suoritin kauan sitten omia kokeita ja huomasin, että GCC:n C++- ja Clang++-kääntäjien välillä oli eroja siinä, että kuinka kääntäjät järjestelivät muistin kohdennukseen käytetyt täytetävät pinomuistissa. Tämä ongelma tulee vastaan, kun funktiolla on ensinnäkin riittävä määrä argumentteja, että niitä ei voida lähettää rekistereissä. Tässä tapauksessa argumentteja on lähetettävä pinomuistissa. Toisena ehtona eri kokoisia argumentteja on oltava muistin kannalta epäjärjestyksessä niin, että pinomuistiin joudutaan luomaan useita erikokoisia täytetäviä blokkeja.

En ole täysin varma mistä nämä erot käyttäytymisessä johtuivat. Binääritason standardi kertoo hyvin, kuinka funktion argumentit järjestellään rekistereihin, ja mitkä argumentit lähetetään pinomuistissa [37]. Standardi ei kuitenkaan antanut selkeää kuvaa siitä, kuinka täytetävät järjestellään pinomuistissa.

Suoritin nämä samat kokeet uudestaan ja havaitsin, että järjestyseroja ei enää ollut, vaan kummatkin kääntäjät käyttäytyivät samalla tavalla. Kääntäjät varasivat pinomuistista kahdeksan tavun tilan kaikille pinossa lähetettäville argumenteille riippumatta niiden koosta, ja täytetävät olivat järjestelty samaan tapaan. Tämä tuntuu noudattavan sysvabi-dokumenttia, joka käskää pyöristää argumenttien koot kahdeksantavuisiksi [37, s. 20-24].

Aikaisemmista ongelmista johtuen koodissa on ajateltu ratkaisu järjestysongelmalle, jotta kääntäjä ei riipu sen kääntäneestä kääntäjästä. Sen sijaan, että argumentit lähetettäisiin funktiolle normaalisti, lähetetäänkin funktiolle kaksi osoitinta sen argumentteihin ja paluuarvoihin. C++-koodi manuaalisesti purkaa argumentit lähetetyistä osoittimista. Standardin mukaisesti nämä osoittimet lähetetään funktiolle rekistereissä RDI ja RSI [37]. Kun argumentteina on vain nämä kaksi osoitinta, niin standardi kertoo selvästi mitä argumenteille tehdään. Kummatkin kääntäjät käyttäytyvät tässä tapauksessa samalla tavalla.

7.1.5 Upotettu tieto

x86-prosessorin koodiin voidaan upottaa tietoa kahdella tavalla. Ohjeisiin voidaan koodata vakioarvoja, tai itse ajettavan koodin lähelle voi sijoittaa mitä tahansa tavuja. Vakioarvoiksi ohjeisiin pääasiassa käyvät kokonaisluvut 32 bittiin asti. [38]

Koodin viereen upotettua tietoa voidaan käyttää säilömään monimutkaisia vakioarvoja, kuten luokkia tai listoja. Tämän lisäksi liukulukuvakioarvot säilötään aina tällä tavalla, koska prosessorin ohjeisiin ei voida koodata liukulukuja. Tällä hetkellä tiedon upotusta koodin viereen tehdään ainoastaan testikoodissa, koska ZVM ei vielä tue liukulukuja.

Upotettuun tietoon viitataan RIP-suhteellisella osoituksella. Kääntäjän luoma koodi on tästä syystä positiosta riippumaton. Jos asiaa ajatellaan pidemmälle, ja ohjelmaan halutaan upottaa jokin valtava tiedosto, jonka koko on enemmän kuin RIP-suhteellisen osoituksen suurin mahdollinen siirtymä 2 Gt [38, s. 88], niin silloin positio riippumattomuus särkyi. Silloin on luotava osoi-

tepöytä missä joillekin tiedoille on kahdeksantavuinen osoitin. Kääntäjä ei voi siirtää positioriip-
puvaista koodia muistissa. Esimerkiksi luodun ohjelman tallennus kovalevylle tarvitsee lisätoi-
menpiteitä. Ohjelmalle voidaan tässä tapauksessa rakentaa lataaja. Kun ohjelmaa käynnistetään,
niin lataaja voisi päivittää pöydän osoitteet paikkoihin, joihin tieto on sijoitettu. Tämä ajattelu
menee kuitenkin aika pitkälle, koska moni ohjelma ei ole 2 Gt kokoinen. Muutenkin, jos kohde-
käyttöjärjestelmänä on Windows, niin riippuen keneltä kysyy, suoritettavan ohjelman suurin
mahdollinen koko on joko 2 Gt tai 4 Gt [41] [42] [43] [44]. Tässäkin keskustelussa ongelmana oli
hyvin mahdollisesti RIP-suhteellinen osoitus. Koodiin upotus ei ole oikea paikka suurille tietokan-
noille. Suurien tietokantojen kuuluisi olla eri tiedostoissa.

7.1.6 Muistin kohdennus

Prossessorin SSE- ja AVX-ohjeet sisältävät monia ohjeita, jotka voivat ainoastaan käyttää muisti-
osoitteita, jotka ovat jonkin luvun kerrannainen. Muutenkin oikein kohdennettu muisti parantaa
suorituskykyä [22 s. 109, 119-121]. Kohdennuksen vaatimus pätee pääasiassa upotettuun tie-
toon, koska liukulukuvakioarvot tallennetaan tällä tavalla ohjelman koodin viereen, ja liukulukuja
käsitellään SSE- ja AVX-ohjeilla. Nämä asiat on otettu huomioon kohdentamalla koodiin upotettu
tieto ja pinomuisti oikein.

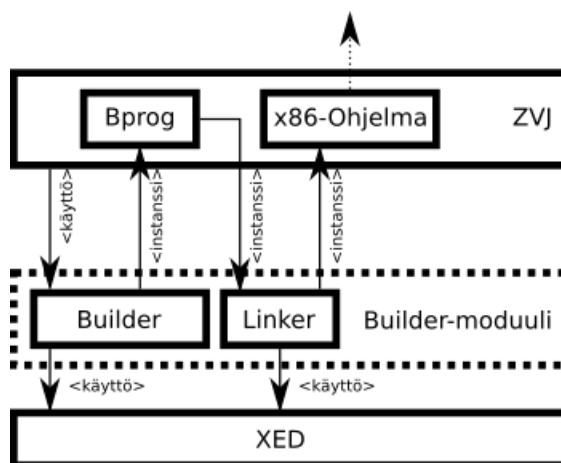
7.2 Builder

Builderia käytetään x86-prosessorille käyvien ohjelmien kokoamiseen.

Builder on koko kääntäjän alimman tason komponentti, lukuun ottamatta XED-kirjastoa (X86 Encoder Decoder). Builder käyttää XED-kirjastoa koodaamaan prosessorin ohjeita tavujonoiksi. Builder ei kokonaan piilota XED-kirjastoa, vaan paljastaa käyttäjälle tämän kirjaston määrittämät, prosessorin ohjeissa käytettyjen operandien tyypit. Käyttäjä antaa builderille näitä operandeja ohjeisiin, ja builder lähettää nämä eteenpäin XED-kirjastolle tavujonoja luodessa.

Käyttäjän näkökulmasta builder-komponentin kokonaisuus sisältää builder-projektin, builder-luokan, ja linker-luokan. Builder-luokalla luodaan koodipaloja, ja linker-luokalla yhdistetään kaikki luodut koodipalat yhdeksi lopulliseksi ohjelmaksi, jonka voi suorittaa prosessorilla. Builder-luokan instanssit pyydetään builder-projektilta. Koodipalojen luominen builder-luokalla tapahtuu niin, että builderi pitää sisällään muokattavan koodipalan, ja builderille kerrotaan mitä prosessorin ohjeita halutaan lisätä, ja mihin kohtaan ohjelmassa.

Builder-moduulin rakenne, käyttö ja tiedonsiirto on esitetty kuvassa 9. Kuvassa "Bprog"-luokka on yhden builderin luoma tavujono.



Kuva 9. Builderi-moduulin paikka arkkitehtuurissa.

XED-kirjasto on periaatteeltaan hyvin yksinkertainen kirjasto. XED-kirjastolle kerrotaan mikä ohje halutaan, sen operandit ja formaattitiedot. Tällöin XED antaa tavujonon halutulle ohjeelle. Tämän

lisäksi XED osaa myös purkaa tavujonot takaisin abstraktimpaan muotoon, mutta tätä ominaisuutta ei tässä projektissa käytetä, varsinkin kun builderi jo pitää ohjeet tämänlaisessa helposti analysoitavassa muodossa. XED ei itse kokoa ohjelmaa, eli pistä ohjeiden tavujonoja toisen perään. XED ei myöskään tarjoa mitään muita projektin kannalta tärkeitä ominaisuuksia, kuin ohjeiden tavujonojen hakemisen. Yksistään XED:in käyttäminen ohjelmien kokoamiseen on melkein mahdotonta. Tietenkin tavujonot pitää pistää edellisen perään, mutta on myös paljon tätä suurempi ongelma, kohtiin viittaaminen ohjelmassa. XED:iä käytettäessä kohtiin ohjelmassa ei voida viitata abstraktisti. Viittaamisen sijaan ohjeille annetaan suoraan suhteellinen siirtymä tavuina. Builderin tasolla on toteutettu viittausjärjestelmä, joka sallii kohtiin viittaamisen abstraktisti. Builderi laskee ohjeille tarvittavat siirtymät annetuista viittauksista.

Saatavilla on useita kirjastoja, joita voi käyttää x86-ohjelmien kokoamiseen, ja jotka toteuttavat monet builderin ominaisuudet, kuten viittaukset. Kuitenkin projektin aikana kokeiltiin monia tämänlaisia kirjastoja, ja yksikään näistä ei täyttänyt kaikkia vaatimuksia. Builderi on rakennettu kirjastojen tapaan niin, että se pystytään ottamaan erilleen tästä projektista, ja käyttämään sitä muiden kirjastojen tyyliseen tapaan. Builderin muistuttaa monia näitä kirjastoja, mutta se on hie- man monimutkaisempi, ja paljon vahvempi.

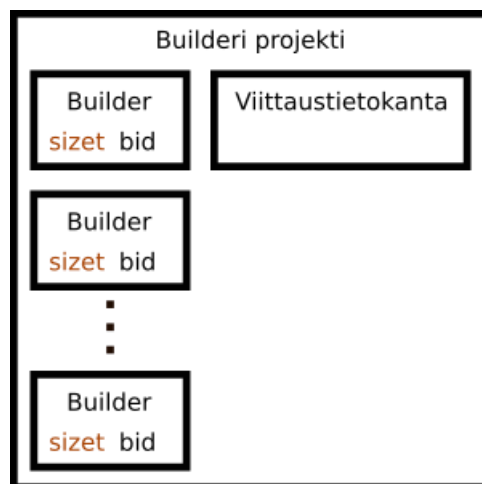
7.2.1 Monisäikeinen koodin generointi

Eräs huomattava ominaisuus mitä builderilta halutaan, on koodin luominen monella säikeellä. Tämän pitää tapahtua antamalla jokaiselle säikeelle instanssi builderista. Näitä buildereita tulee pystyä käyttämään itsenäisesti. Monen eri säikeen builderin sisältämä koodi liitetään lopussa yhdeksi ohjelmaksi linkittäjällä. Tämä aiheuttaa vaatimuksia builderille. Builder-projekti kokonaisuus on luotu, jotta monisäikeinen koodin generointi saadaan toteutettua. Builderilla on viitejärjestelmä, jota käytetään viittaamaan johonkin kohtaan ohjelman koodissa. Tähän viitejärjestelmään on pitänyt toteuttaa toiminnallisuutta monisäikeisyyttä varten. Viitejärjestelmä käsitellään tarkemmin myöhemmin.

Suurin vaatimus, jonka monisäikeisyys aiheuttaa builderille on, että jokaisen builderin on kyettävä viittaamaan tietoihin, joka on olemassa jollakin toisella builderilla. Esimerkkinä tästä tietoihin viittauksesta on funktiokutsu käännettävässä kielessä. Käännettävässä kielessä jonkin funktion määrittäminen voi olla toisessa tiedostossa. Tämän funktion koodi voidaan generoida toisella säikeellä samaan aikaan, kun funktiota käyttävää koodia luodaan.

Builderilla on mahdollista luoda funktiota käyttävä koodi samaan aikaan, kuin funktion koodia luodaan. Tämän mahdollistaa builderille rakennettu viitejärjestelmä. Viitejärjestelmää käyttämällä käyttäjä voi viitata funktion osoitteeseen sitä kutsuvassa koodissa, vaikka tätä osoitetta ei vielä tiedetä. Kun funktio, ja sitä käyttävä koodi ovat valmiita, pitää käyttävään koodiin sijoittaa funktion osoite paikoilleen. Tämä tapahtuu antamalla kummankin builderin luomat koodipalat linker-luokalle, joka sijoittaa osoitteet paikoilleen.

Viittausjärjestelmä ei toimi täysin ilman builderien välistä kommunikointia. Tätä viittausjärjestelmää varten on oltava kaikkien kääntämisessä käytettyjen builderien välinen yhteinen tietokanta. Tiedon synkronointia varten luodaan kokonaisuus, jota kutsutaan builderi projektiksi. Projekti sisältää listan kaikista ohjelman kokoamisessa käytetyistä buildereista, ja niiden yhteisen viittaus-tietokannan. Projektissa jokaiselle builderille annetaan uniikki tunniste viittausjärjestelmää varten. Tämä rakenne on esitetty kuvassa 10. Kuvassa ”bid” tarkoittaa builderin uniikkia tunnistetta.



Kuva 10. Builder projekti.

Tällä hetkellä builderia ei käytetä luomaan koodia monella säikeellä, vaikka käytössä oleva järjestelmä pystyy siihen. Kaukana tulevaisuudessa, kun builderin päälle rakennetut järjestelmät alkavat vakiintua, niin koodia on tarkoitus luoda monella säikeellä.

7.2.2 Ohjeiden korjaus

Eräs asia minkä builderi joutuu hoitamaan, on ohjeiden korjaus käyttämällä muita ohjeita. Builder ei korjaa käyttäjän virheitä, vaan vastuu prosessorien ohjeiden validiteetista on käyttäjällä. Käyttäjä siis saa syöttää builderille vain prosessorille käypiä ohjeita, jotka se voi suorittaa. Kuitenkin asiat eivät mene näin helposti, vaan on tapauksia, joissa käyttäjä ei pysty tietämään milloin jokin ohje on validi. Näissä tapauksissa builderi voi hiljaisesti korjata ohjeita käyttämällä yhtä tai useampaa muuta ohjetta.

Tästä erinomaiseksi esimerkiksi saadaan hyppyohteet, jotka hyväksyvät vain hyvin pienen suhteellisen siirtymän operandiksi, nimenomaan "loop"-ohje. "loop"-ohje voi ottaa operandiksi vain yhden tavun suhteellisen siirtymän, mikä tarkoittaa -128 – 127 tavun siirtymää [38, s. 1167]. Tämä raja voi tulla usein vastaan suurissa funktioissa. Ohjelma on luontihetkellä vielä abstraktissa muodossa, eikä ohjeiden formaattia olla vielä päätetty. Tämän lisäksi ohjeita voidaan lisätä mihin tahansa kohtaan ohjelmassa, vaikka hypyn ja sen kohteen väliin, joten siirtymä muuttuu paljon. Käyttäjä ei voi mitenkään tietää lopullisen ohjelman kokoa, joten käyttäjä ei voi tietää milloin tätä ohjetta voidaan käyttää. Builder on rakennettu niin, että käyttäjä voi aina syöttää tämän ohjeen builderille. Jos tätä ohjetta ei lopulta voida käyttää, niin builderi emuloi tätä ohjetta muilla.

7.2.3 Builderin abstrakti formaatti

Ohjelman luontihetkellä builder pitää ohjeet koodaamattomassa muodossa tietorakenteina. Builder ei siis luo tavujonoja aina, kun builderilla lisätään ohje ohjelmaan. Vasta sitten kun ohjelma on valmis, niin builder luo oikealle prosessorille käyvän tavujonon. Kun ohjelma on vielä abstraktissa muodossa, niin sitä on tehokkaampi käsitellä verrattuna siihen, että tavujonoja ruvettaisiin siirtelemään ja muuttamaan aina, kun ohjeita muutetaan. Abstrakti muoto tarjoaa tehokkaan ja helpon tavan muokata ohjelmaa. Abstrakti muoto myös sallii mielivaltaisen tiedon käytön ohjelmassa, ennen kuin tavujono luodaan. Tämä on tärkeä ominaisuus matalan tason kääntäjälle.

Builderin abstrakti muoto koostuu useasta peräkkäin olevasta noodista (engl. node). Noodi voi olla tyypiltään joko ohje, viitekohta, DB, muistin kohdistuksen merkki tai käyttäjän oma data.

- Ohje-noodi sisältää tietoa mikä prosessorin ohje ja operandit ovat kyseessä. Lopulliseen ohjelmaan luodaan tavujono tälle ohjeelle tämän noodin paikalle.
- Viitekohta-noodi ei luo mitään tavuja lopulliseen ohjelmaan, vaan tämän noodin kohdalle voidaan viitata ohjelmassa käyttämällä noodin sisältämää viitemerkkiä.
- DB-noodi tarkoittaa upotettavaa tavujonoa. Tämä noodin sisältää tietoina jonkin pituisen tavujonon. Tämä tavujono upotetaan suoraan noodin kohdalle ohjelmassa.
- Muistin kohdistus -noodi sisältää luvun, jonka kerrannainen ohjelman koon pitää olla tämän noodin jälkeen. Tätä käytetään DB noodin kanssa, jotta DB:n tiedot saadaan kohdennettua, ja näin ladattua ohjeilla, jotka vaativat kohdistetun osoitteen.
- Käyttäjä data -noodeja ei oteta huomioon. Tämä noodin voi sisältää jotakin mielivaltaista tietoa. Builderia käyttävät komponentit saavat tehdä tällä tiedolla mitä tarvitsevat.

7.2.4 Viitejärjestelmä

x86-prosessorilla monet ohjeet voivat ottaa operandiksi suhteellisen etäisyyden suorituksen kanalta nykyisestä kohdasta ohjelmassa tavuina. Suhteellisen siirtymän lisäksi ohjeet yleensä voivat lukea absoluuttisen osoitteen rekisteristä, tai muistista. Tällöin osoite pitää tietenkin ladata rekisteriin ensin, mikä tarkoittaa ylimääräisiä ohjeita. Muutenkin kaikki ohjeet, kuten esimerkiksi LOOP, eivät tarjoa mahdollisuutta käyttää absoluuttista osoitetta [38, s. 1167]. Mahdollisuudet luoda suhteellinen viittaus on siis pakko tehdä. XED-kirjasto ei tarjoa mahdollisuutta viitata kohtiin ohjelmissa niin, että XED laskisi etäisyyden itse. Etäisyys lasketaan itse viitauksista, ja annetaan XED:ille tavuina.

Viittausjärjestelmää käytetään tarjoamaan suhteellinen siirtymä ohjeille. Käytännössä tämä tarkoittaa hypyn kohteen ilmoittamista hyppyoheille, tai koodin viereen upotetun tiedon käyttämistä käyttämällä muistioperandeja.

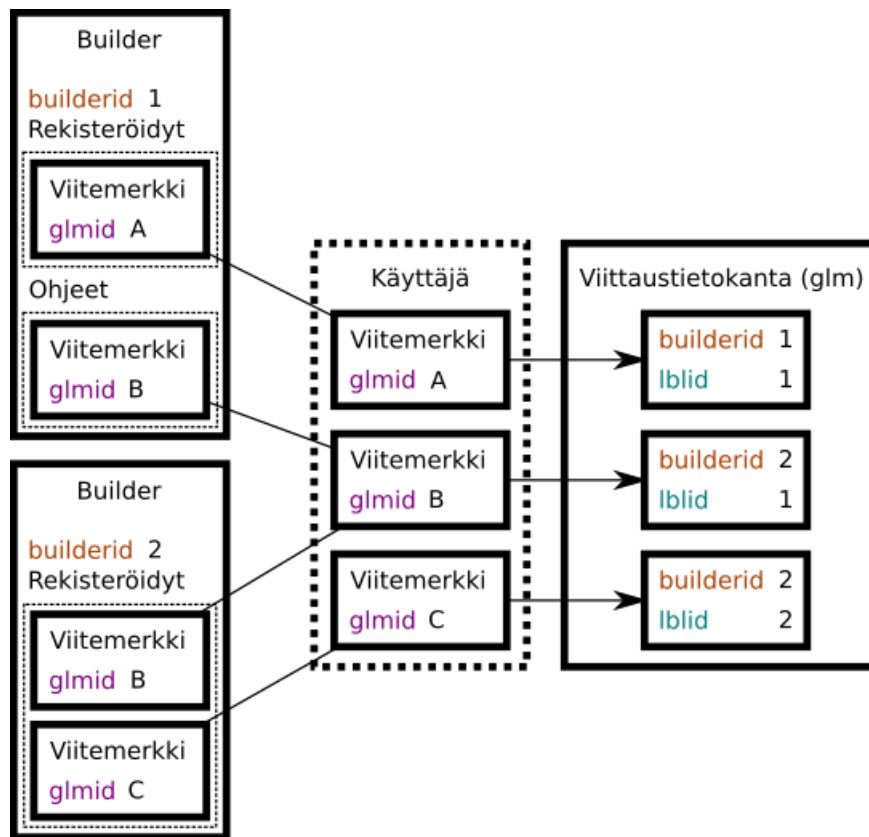
Viitejärjestelmä koostuu viittaustietokannasta, viitemerkistä, viitemerkki-noodista builderilla, sekä erikoisoperandista ohjeissa. Kohtiin viittaaminen builderilla toimii rekisteröimällä viittaustietokannan antama viitemerkki jollekin builderille johonkin kohtaan ohjelmassa. Tähän kohtaan voidaan sitten viitata antamalla sama viitemerkki erikoisoperandina ohjeisiin. Kohtaan voidaan viitata viitemerkillä muistakin buildereista, kuin siitä, mihin viitemerkki rekisteröitiin. Viitemerkki voidaan myös antaa parametreiksi ohjeille, ennen kuin se on rekisteröity, ja rekisteröimällä tämä viitemerkki myöhemmin. Tällöin myös jo parametreiksi annettu viitemerkki rekisteröityy kyseiseen kohtaan ohjelmassa. Tarkemmat kuvakset näille komponenteille ovat tässä.

- Viitemerkki on luokka kääntäjän koodissa. Builderin käyttäjä pyytää viitemerkit viittaustietokannalta, ja käsittelee näiden instansseja. Jokaisella viitemerkillä on viittaustietokannan tunniste. Tunnistetta käytetään hakemaan viitemerkin tiedot viittaustietokannasta. Tämä tunniste on piilotettu builderilta, ja builderin käyttäjältä. Kun viitemerkit annetaan parametreiksi ohjeille, ja rekisteröidään myöhemmin, niin rekisteröitäessä myös parametreiksi annettu kopio rekisteröityy. Tämä johtuu siitä, että kun kääntäjän koodissa viitemerkin instanssista otetaan kopioita, niin tällä koko kopiopuulla on aina sama tunniste. Huomattava asia on, että viitemerkki voidaan myös asettaa osoittamaan samaa paikkaa kuin toinen viitemerkki. Tämä tarkoittaa ”=”-operaattoria. Kun näin tehdään, niin kummankin viitemerkin kopiopuu yhdistyy samaksi niin, että kaikilla viitemerkeillä on sama viittaustietokannan tunniste. Tämä on toteutettu disjoint-set-tietorakenteella. Nämä ominaisuudet ovat olemassa siksi, että viitemerkkejä voidaan käsitellä normaalisti, ennen kuin ne ovat rekisteröity johonkin paikkaan ohjelmassa.
- Viittaustietokannasta on yksi yhteinen globaali instanssi kaikille buildereille. Käyttäjä pyytää viitemerkki-luokan instanssit tältä oliolta. Viittaustietokannalla on muistissa tietorakenne jokaiselle viitemerkille. Tietorakenne sisältää builderin tunnisteiden, ja kyseisen builderin antaman uniikin tunnisteiden tälle viitemerkille. Nämä tunnisteet määritetään, kun käyttäjä rekisteröi viitemerkin jollekin builderille. Viitemerkin tiedot pyydetään antamalla viitemerkin instanssi viittaustietokannalle. Builder pyytää näitä tietoja viitemerkille, mutta builderin käyttäjän ei tarvitse koskea tähän tietoon. Riittää että käyttäjällä on instanssi viitemerkistä.
- Viitemerkki-noodi sisältää viitemerkin. Tämä noodi asetetaan johonkin kohtaan ohjelmassa, kun viitemerkki rekisteröidään kyseiseen paikkaan. Tämä noodi merkitsee sen sisältävän viitemerkin rekisteröityä paikkaa ohjelmassa.

- Erikoisoperandi voidaan antaa ohjeille samaan tapaan, kuin kaikki muutkin operandit. Tätä operandia ei tietenkään voi antaa XED:ille tai prosessorille, vaan tämä operandi muunnetaan, joko muistioperandin siirtymäksi tai suhteelliseksi siirtymäksi, riippuen ohjeesta mille operandi annettiin. Siirtymät saadaan hakemalla operandin sisältävän ohjeen paikka ohjelmassa, ja laskemalla etäisyys operandin sisältämän viitemerkin noodiin.

Builderilla halutaan luoda koodia monella säikeellä yhtä aikaa niin, että jokaisella säikeellä on oma builderi. Jokainen builderi voi viitata kaikkien muiden säikeiden builderien sisältämään viitemerkkiin. Tätä varten säikeellä on oltava kopio jonkin toisen säikeen builderin sisältämästä viitemerkistä.

Kuvassa 11 on esitetty, kuinka kaikki tunnisteet suhtautuvat toisiinsa tapauksessa, jossa kaksi builderia sisältävät rekisteröityjä viitemerkkejä, ja builderi viittaa toisen builderin viitemerkkiin ohjeella. Nimi "builderid" tarkoittaa builderin uniikkia tunnistetta projektissa, nimi "glmid" tarkoittaa viittaustietokannan tunnistetta ja "lbid" tarkoittaa builderin antamaa tunnistetta rekisteröidylle viitemerkille. Viitemerkki A on rekisteröity builderille 1, ja viitemerkit B ja C ovat rekisteröity builderille 2. Builderin 1 ohjelma sisältää viitemerkkinoodin, siihen rekisteröidylle viitemerkille A. Tämän lisäksi builderin 1 ohjelma sisältää ohjeen, mikä viittaa builderin 2 ohjelmaan viitemerkillä B. Builderilla 2 on kaksi rekisteröityä viitemerkkiä, B ja C. Builderi 2 on viitemerkkejä rekisteröitäessä antanut näille eriävän builderikohtaisen tunnisteen. Tämä builderikohtainen tunniste menee päälle toisen builderin antamien tunnisteen kanssa niin kuin kuuluukin, koska tunniste on builderikohtainen.



Kuva 11. Viitemerkit ja tunnisteet monella builderilla.

7.2.5 Viittausjärjestelmän toteutus

Toteutuksen kannalta on huomioitava pari asiaa, jotka aiheuttavat vaatimuksia.

Viitemerkin lopullinen osoite ohjelmassa saadaan vasta, kun kaikkien builderien koodi on generoitu tavujonoksi, ja liitetty yhteen lopulliseksi ohjelmaksi. Tästä syystä yhden builderin tavujonoa generoidessa, ei ole mahdollista tietää toiseen builderiin rekisteröidyn viitemerkin kohtaa lopullisessa ohjelmassa, koska lopullinen ohjelma kasataan kaikista näistä generoidusta tavujonoista. Eli kyseessä on ympyräriippuvaisuus, tavujonot ovat generoitava ensin, jotta lopullinen ohjelma saadaan näistä kasaan, ja lopullinen ohjelma on generoitava ennen tavujonoja, jotta kun tavujonoja generoidaan, niin voidaan hakea viittausten osoitteet suhteellisen etäisyyden laskemiseksi. Tämä käy ilmiselväksi, kun kaksi funktiota, jotka ovat luotu eri buildereilla, kutsuvat toisiaan. Tämä osoitteen hakemisen ongelma tapahtuu myös yhden builderin sisällä. Ohjeen viittaus voi viitata kohtaan, joka tulee myöhemmin ohjelmassa. Koska ohjelman tavujonoa luodaan järjestyksessä, niin tämä kohta tulee nykyisen generoinnin kohdan jälkeen. Viittauksen sisältävää ohjetta

ei voida generoida, koska se vaatii siirtymän myöhempään kohtaan ohjelmassa. Koodia ei ole generoitu tähän asti, joten siirtymää ei voida laskea. Kyseessä on taas ympyräriippuvaisuus.

Builderin edespäin suuntautuvan viittauksen generoinnin ongelma voidaan ratkaista luomalla tämän ohjeen paikalle tyhjä ohje, mihin ei ole vielä määritetty tarkkaa siirtymää. Tämä tarkka arvo kirjoitetaan paikalleen myöhemmin. Siirtymän koko riippuu etäisyydestä sen mukaan, kuinka moni tavuinen numero tarvitaan etäisyyden esittämiseen. Tämä ratkaistaan laskemalla pahimman mahdollisen tapauksen etäisyys eteenpäin olevaan viittaukseen, ja käyttäen näin monta tavua esittämään etäisyyttä viittauksen sisältävässä ohjeessa. Näin ohje voidaan generoida niin, että sen koko ei muutu, ja näin voidaan jatkaa koodin generointia eteenpäin. Tarkat etäisyydet sijoitetaan paikoilleen koodin generoinnin lopussa.

Kahden builderin välisen viittauksen ongelma noudattaa samaa periaatetta. Koska tässä tapauksessa pahimman tapauksen etäisyyttä ei voida laskea, niin tässä yksinkertaisesti käytetään suurimman kokoista lukua. Tätä viittausta builderi ei voi ratkaista vielä tavujonoa generoitaessa, joten builderi antaa käyttäjälleen metatietoja tästä ratkaisemattomasta viittauksesta. Käyttäjä antaa kaikki ohjelmat, ja niiden metatiedot linkittäjälle, joka kasaa näistä ohjelman yhteen, ja ratkaisee loput viittaukset.

7.2.6 Tavujonon luominen ohjelmalle

Builder luo prosessorin ohjelmalle tavujonon prosessilla, joka on esitetty kuvassa 12.



Kuva 12. Tavujonon luominen builderilla.

Tarkempi kuvaus näille vaiheille on esitetty tässä.

1. Tyhjä tietokanta luodaan, missä on jokaisen käytetyn viitemerkin tiedot. Tiedot kertovat, että onko viitemerkille jo saatu paikka koodiin, ja missä tämä paikka on. Tämä paikka on siirtymä ohjelman alusta tavuina. Tätä tietokantaa kutsutaan viitelinkkikannaksi.
2. Ohjelma jaetaan osiin niiden positioriippuvaisuuteen perustuen. Ohjelma jaetaan osiin näin, jotta joidenkin osien tavujonot voidaan luoda ensin. Tämä sallii tehokkaan ja paremman etäisyydenmittauksen myöhemmin. Osat koostuvat peräkkäisistä noodeista, joiden tavujonoa generoitaessa joko pitää, tai ei pidä tietää noodin täsmällistä paikkaa ohjelmassa. Ohjeet, jotka käyttävät viitemerkin sisältävää erikoisoperandia, tarvitsevat tiedon paikastaan lopullisessa ohjelmassa etäisyyden laskua varten. Tämän lisäksi muistin kohdistus noodit tarvitsevat myös paikkansa. Jaon säännöstä johtuen ohjelma jakaantuu

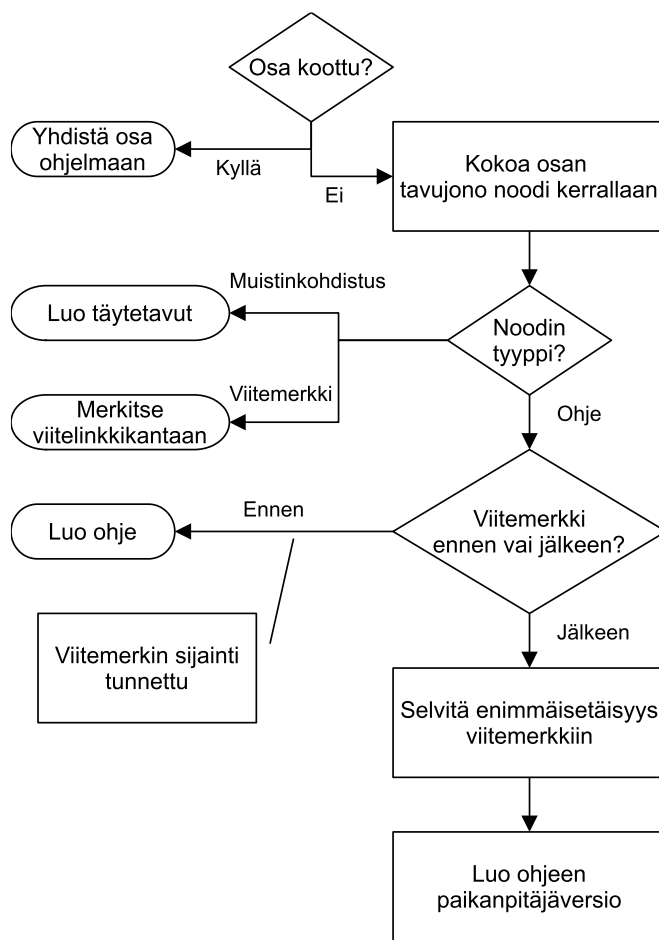
osiin niin, että joka toisen osan on tiedettävä paikkansa, ja toisen ei. Kuvassa 13 on kuvattu ohjelman jakautumista eri osiin. Kuvassa viitemerkki- operandi luodaan "ba_l"-funktioilla.

```
builder& b = prj.builder_get(prj.builder_new());
builder_glblm::label l = prj.glm.new_();

b.loop(ba_l(l));
.....
b.mov(xed_reg(XED_REG_EAX), xed_reg(XED_REG_ECX));
b.mov(xed_reg(XED_REG_RAX), xed_reg(XED_REG_RCX));
.....
b.mov(xed_reg(XED_REG_RAX), ba_l(l));
.....
b.add(xed_reg(XED_REG_EAX), xed_reg(XED_REG_ECX));
b.add(xed_reg(XED_REG_ECX), xed_reg(XED_REG_EAX));
b.mov(xed_reg(XED_REG_RDX), xed_reg(XED_REG_RBX));
.....
b.loop(ba_l(l));
b.loop(ba_l(l));
.....
b.mov(xed_reg(XED_REG_EBX), xed_reg(XED_REG_EAX));
```

Kuva 13. Testikoodi ohjelma ja sen jakautuminen eri osiin.

3. Kaikki osat, joiden ei tarvitse tietää paikkaansa koodissa, kootaan tavujonoiksi. Kun tavujonoa kootaan, niin jokaisen osassa olevan viitemerkkinoodin viitemerkille saadaan paikka tämän osan tavujonossa. Tämä paikka on siirtymä suhteessa osan alkuun, eikä ohjelman alkuun. Siirtymä korjataan myöhemmin, kun osien tavujonoille saadaan paikka lopullisessa ohjelmassa.
4. Ohjelma on nyt osissa, joista joka toinen osa on koottu, ja toiset eivät ole koottu. Lopullisen ohjelman tavujonoa aloitetaan luomaan järjestyksessä. Vastaan tulevia osia tutkitaan kuvan 14 mukaisesti.



Kuva 14. Ohjelman osan käsittely.

Kun vastaan tulee koottu osa, niin tämän osan tavujono yhdistetään lopullisen ohjelman tavujonon perään. Tämän osan viitemerkkinooodeilla on paikka suhteessa osan alkuun. Tämä siirtymä korjataan niin, että se on nyt suhteessa ohjelman alkuun, ja tälle viitemerkille merkitään paikka viitelinkkikantaan. Kyseisellä viitemerkillä on nyt tunnettu lopullinen sijainti ohjelman tavujonossa, ja ohjeiden etäisyydet tähän viitemerkkiin voidaan laskea.

Kun vastaan tulee osa, jota ei ole koottu, niin sen tavujonoa lähdetään kokoamaan noodin kerrallaan kuvan 14 mukaisesti. Nämä osat sisältävät ainoastaan noodeja, jotka joko ovat viitemerkkioperandia käyttäviä ohjeita, tai noodit ovat tyypiltään joko muistinkohdistus, tai viitemerkki noodeja. Näitä noodeja generoitaessa tiedossa on nykyinen sijainti lopullisessa ohjelmassa, eli generoitavan noodin sijainti on tiedossa.

Muistinkohdistus-noodi saadaan luotua helposti. Myös viitemerkkinoodille saadaan suoraan lopullinen paikka viitelinkkikantaan.

Kun vastaan tulee ohje, niin tämän ohjeen käyttämää viitemerkkiä analysoidaan.

Jos viitemerkki tuli ohjelmassa ennen tätä ohjetta, niin silloin tällä viitemerkillä on tunnettu paikka viitelinkkikannassa, ja näin ollen etäisyys saadaan laskettua, ja ohje koottua.

Jos viitemerkki tuli nykyisen ohjeen jälkeen, niin tällä viitemerkillä ei ole tunnettua paikkaa. Tämä ratkaistaan niin, että ohjeesta luodaan ensin paikanpitäjä versio, johon ei vielä ole sijoitettu etäisyyttä. Ohjeen luontia varten tarvitaan siirtymän tavumäärä. Siirtymän pahimman tapauksen enimmäisetäisyys saadaan laskettua tutkimalla ohjelmaa eteenpäin. Jos vastaan tulee osa, mikä on käännetty, niin tämän osan koko lisätään laskuun. Jos vastaan tulee kääntämätön ohje, niin silloin oletetaan, että tämä ohje joutuu käyttämään pisintä muotoa itsestään. Näin saadaan laskettua suurin mahdollinen etäisyys edessäpäin olevaan viitemerkkiin, ja näin ollen tarvittavien tavujen määrä.

5. Kun kaikki ohjeet ollaan käsitelty, niin paikanpitäjä ohjeisiin sijoitetaan paikoilleen tarkka etäisyys.

Monta ongelmaa voisi välttää, jos olettaisi aina, että ohje vie suurimman mahdollisimman tilan. Mutta näin tehdessä joitakin ohjeita, jotka eivät kykene käyttämään suuria siirtymiä, ei voida käyttää, ja näin ollen builderi joutuu korjaamaan paljon enemmän ohjeita.

7.2.7 Builder tulevaisuudessa

Builder on pääasiassa valmis. Kaikki tarvittavat ominaisuudet ovat toiminnassa. Kuitenkin builderi saattaisi olla viisaampaa toteuttaa hieman tyhmempänä komponenttina.

Ohjeiden korjaus ja prosessorin tila yhdessä on ongelma. Jos esimerkiksi LOOP-ohjetta joudutaan korjaamaan, niin sitä voidaan emuloida käyttämällä CMP-, JMP- ja DEC-ohjeita. Ongelmana on, että LOOP-ohje ei koske prosessorin lippuihin, mutta CMP- ja DEC-ohjeet muuttavat näitä lippuja. Tästä voi seurata tilanne, missä käyttäjä odotti lippujen olevan muuttumattomia LOOP-ohjeen yli, mutta näin ei enää ole, koska LOOP korvattiin muilla ohjeilla. Tämä on ratkaistavissa tallentamalla liput LOOP-ohjeen paikalle sijoitetun koodin yli. Tästä seuraa ylimääräisiä ohjeita. Käyttäjän olisi

lopulta ollut tehokkaampaa käyttää normaalia CMP ja JMP -yhdistelmää, kuin LOOP-ohjetta. Lipuja ei tarvitse aina tallentaa, mutta builder ei tiedä, milloin näin on. Ainoastaan käyttäjä tietää tämän.

Prosesorin liput ja muu tila ovat vielä harmaalla alueella, näiden käsittelemiseen ei ole hyvää arkkitehtuuria. Builder ei voi korjata ohjeita kovin tehokkaasti, mutta ZVJ voi tehdä päätöksiä käytetyistä ohjeista. Builder voisi olla tyhmempi komponentti, joka ei korjaisi ohjeita, vaan paljastaisi etäisyyksien laskemisen käyttäjälle. Tämä tarkoittaa, että builderin päälle voitaisiin rakentaa jokin komponentti, mille haluttu toiminta kerrotaan abstraktissa muodossa. Tämä komponentti voisi päättää tarkat käytetyt ohjeet ohjelman luontihetkellä. ZVM on komponentti, joka täyttää korkeamman tason builderin kuvauksen. Tästä syystä nämä ominaisuudet voitaisiin jättää builderista pois, ja antaa ZVJ:n hoitaa tämän. Tällöin ZVJ:n sisälle luodaan uusia järjestelmiä.

7.3 Rekisterivaraaja

ZVM käsittelee tietoa muuttujina, kun taas x86-prosessori lataa tietoa muistista rajalliseen määrään rekisterejä, joissa tietoa käsitellään. Prosessorilla on hyvin monia yleiskäyttöisiä rekisterejä, mutta muuttujia voi olla käytännössä rajaton määrä, joten muuttujia ei voida pitää ainoastaan rekistereissä. Tästä johtuen jokaiselle muuttujalle luodaan tallennuspaikka pinomuistiin. Toteutus yrittää parhaansa mukaan pitää muuttujat ladattuna rekistereissä niin pitkälle kuin mahdollista. Kuten mainittu, yleiskäyttöisiä rekisterejä on hyvin monia, niin teoriassa on mahdollista toteuttaa kokonaisia funktiota käyttämällä ainoastaan rekisterejä. Rekisterivaraaja on vastuussa muuttujien lataamisesta rekistereihin, ja näiden rekisterien päättämisestä. Rekisterivaraajan on tarkoitus päättää nämä asiat niin, että latauksista ja tallennuksista tulee mahdollisimman pienet tehokkuus kustannukset. Rekisterivaraajasta käytetään lyhennettä VE (Var Emitter).

ZVJ luo eräänlaisen malliohjelman, jonka VE-järjestelmä käsittelee myöhemmin. Malliohjelma on builderi ohjelma, jota ei pystytä kääntämään ilman lisäkäsittelyä. Malliohjelmassa joitakin ohjeiden operandeja ei ole määritetty. Malliohjelma sisältää useita käyttäjädatanoodeja, mitä VE käyttää lukemaan tietoa muuttujien käytöstä. VE lisää tähän malliin muuttujien lataus- ja tallennusohjeita, ja tämän lisäksi VE sijoittaa muuttujien päätetyt rekisterit paikoilleen ohjeisiin.

Muuttujilla voi olla useita eri tietotyypppejä, ja näitä tietotyypppejä käsitellään prosessorilla eri tavalla, joten VE joutuu miettimään tätä. ZVM tukee teoriassa kokonaislukuja ja liukulukuja, mutta vain 64 bitin kokonaisluku on tällä hetkellä tuettu. Koska VE on kiirehditty valmiiksi, niin VE ei

lopulta mieti tätä ongelmaa. Tulevaisuudessa liukulukujen käsittely voi toimia pitkälti samaan tapaan niin kuin kokonaislukujenkin, koska toteutukseen tullaan käyttämään prosessorin SSE- ja tai AVX-ohjeita. Erot näillä ohjeilla verrattuna kokonaislukujen käsittelyyn voivat olla hyvin pieniä, esimerkiksi vain tiedonsiirto ohjeiden nimet ja käytettävät rekisterit saattavat muuttua. Liukulukujen käsittelystä puhutaan enemmän myöhemmin.

7.3.1 Rajapinta

VE:lle on luotu rajapinta builderin käyttäjädatanoodilla. Tällä hetkellä tämä rajapinta koostuu neljän eri tyyppin käyttäjädatanoodista, VII, VRST, VURST ja VSAVE.

- VII-tyypin noodi lisätään aina ennen jokaista ohjetta. Tämä noodi kertoo missä rekistereissä minkäkin muuttujan on oltava tämän ohjeen kohdalla, ja mitä muutoksia ohje tekee muuttujiin tai rekistereihin. Käyttäjä voi jättää rekisterit päättämättä, ja antaa VE:n päättää nämä käyttämällä tätä noodia. Noodin tieto koostuu sisään- ja ulostulotiedosta.

Sisääntulotieto kertoo halutut rekisterit muuttujille. Käyttäjän ei tarvitse päättää täsmällistä rekisteriä, vaan käyttäjä voi antaa VE:n päättää tämän muuttujan rekisterin. Sisääntulotieto kertoo myös, että minkä operandin paikalle muuttujan rekisteri kuuluisi sijoittaa. Jos ohje ottaa implisiittisen operandin, niin operandin sijoitus tieto jätetään pois.

Ulostulotieto kertoo, että mitä muutoksia ohje teki rekistereihin. Ulostulotieto sisältää rekisterin, jota ohje muuttaa. Koska ohjeet hyvin usein muuttavat niille operandina annettua rekisteriä, ja käyttäjä saa antaa VE:n päättää tämän rekisterin muuttujille, niin tämä ulostulotiedossa käytetty rekisteri voi olla joko jokin tietty rekisteri, mitä tämä ohje aina muuttaa, tai rekisteri, joka sijoitettiin jonkin operandin paikalle ohjeessa. Tämän lisäksi ulostulotieto kertoo, että kirjoitettiin tähän rekisterin jonkin muuttujan uusi arvo, ja mikä muuttuja on kyseessä.

- VRST-tyypin noodi sisältää tietona rekisterin. VE ei saa käyttää annettua rekisteriä ennen kuin vastaava VURST-noodi tulee ohjelmassa tälle rekisterille. Rekisterin rajaus sallii käyttäjän käyttää annettua rekisteriä, miten huvittaa. VE ei luo muuttujien lataus tai tallennus ohjeita koskien tätä rekisteriä tällä alueella. Poikkeuksena ovat implisiittiset VII sisääntulot, joilla on toinen tarkoitus merkitä latausta rajoitetuille rekistereille.

- VSAVE-tyypin noodia käytetään, kun jostakin rajoitetusta rekisteristä halutaan päivittää muuttujan arvo. Tätä käytetään VRST noodin kanssa, koska VE ei koske näihin rekistereihin.

Tällä rajapinnalla on lieviä ongelmia. Nämä ongelmat eivät kuitenkaan aiheuta mitään toiminnallisuuden puutetta. Nämä ovat pääasiassa optimointiongelmia. Hyvin monet ongelmat tulevat esille, kun halutaan luoda ohje, joka hyvin yksinkertaisesti asettaa jonkin muuttujan arvon vakioarvoksi paikassa, missä muuttujan arvoa ei tarvitse heti lukea. Ei ole hyvää tapaa luoda ohjetta, joka vain asettaa jonkin muuttujan arvon ilman, että muuttujan nykyistä arvoa täytyy tietää. Tähän käytettävä ohje on todennäköisesti MOV. MOV voi siirtää jonkin vakioarvon muistiin ja rekisteriin [38, s. 1211]. Edellisistä ohjeista riippuen, muuttuja voi olla asetusohjeen kohdalla joko ladattuna rekisterissä, tai se on pinomuistissa. VE:lle voidaan ainoastaan kertoa, että jonkin muuttujan on löydyttävä jostakin rekisteristä tämän ohjeen kohdalla. Tästä johtuen VE:n on ladattava muuttuja rekisteriin ohjeen ajaksi. Tämä lataus on turha, koska MOV olisi voinut suoraan siirtää arvon pinomuistiin.

Toinen tapa toteuttaa ohje on, että ohje vain asettaisi jonkin rekisterin, ja antaisi ulostulon muuttujalle tässä rekisterissä, mutta tämä ei toimi. Jos muuttuja oli pinomuistissa niin silloin tämä rekisterin asetus oli turha, koska MOV-ohje voi kirjoittaa vakioarvon muistiin. Mutta tämä ei ole se paha ongelma vaan se on, että ei ole tapaa valita jotakin rekisteriä, mitä asetetaan, koska VE:ltä ei voida kysyä mitä rekisterejä sillä on käytössä, koska VE päättää nämä rekisterit tulevaisuudessa vasta sitten, kun malliohjelmaa käsitellään. VRST-noodilla voidaan omia jokin rekisteri, mutta tälle noodille on annettava täsmällinen rekisteri, ja käyttäjä ei voi tietää meneekö hänen antama rekisteri jonkin muuttujan päälle pilaten lataukset. Käytännössä VE voi päättää omat lataukset VRST-noodien mukaan, niin VE:n päättämät lataukset eivät mene minkään päälle. Kuitenkaan VRST:n käyttö rekisterin valintaan ja ulostulon antaminen ei vielä onnistu, koska VE ei saa käyttää tätä rajoitettua rekisteriä, eli VE ei voi tallentaa tämän ulostulon muutoksia.

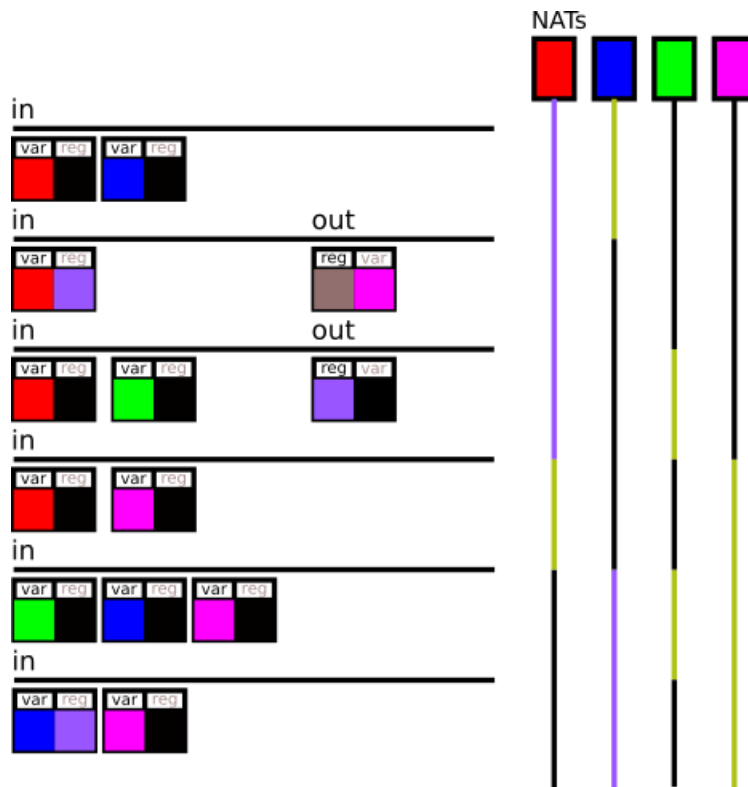
Teoriassa VE pystyy analysoimaan koodia ja optimoimaan nämä asiat, mutta rajapinta on silti ontuva. Rajapinta on myös hyvin monimutkainen, ja tämän tiedon käsittely vaatii hyvin monimutkaiset algoritmit.

7.3.2 Toteutus

Yksinkertaisin tapa toteuttaa muuttujat oikealla prosessorilla on tutkia jokaista builder-ohjelman ohjetta yksitellen. Jokaiselle ohjeelle luodaan lataus- ja tallennusohjeet jokaiselle ohjeen käyttämälle muuttujalle. Tämä on ilman muuta todella epätehokasta. Mietitään yleistä ohjetta, joka lisää muuttujaan luvun yksi, ja pistetään näitä ohjeita peräkkäin. Jokainen näistä ohjeista suorittaa latauksen ja tallennuksen. Tämä muuttuja on koko ajan samassa rekisterissä, joten lataus tarvitsee suorittaa vain ketjun alussa, ja tallennus lopussa. Kun muuttuja ladataan rekisteriin, niin se on saatavilla tässä rekisterissä, kunnes jokin ohje ei-halutusti muuttaa tätä rekisteriä. VE on suunniteltu ensinnäkin pitämään muuttujat rekistereissä, mutta myös optimoimaan latauksia. Kuitenkin aikarajan takia VE:n optimointeja ei olla tehty loppuun. Tällä hetkellä VE ei luo paljoa parempaa koodia, kuin mitä saadaan aikaan edellä esitetyllä minimaalisella tavalla.

VE:n algoritmi on eräänlainen kaavion väritysalgoritmi. Nykyinen algoritmi etsii ohjelmasta monia peräkkäisiä ohjeita, jotka käyttävät samaa muuttujaa. Tätä aluetta kutsutaan aktiiviseksi alueeksi. Muuttuja voidaan pitää samassa rekisterissä näiden ohjeiden yli. Ideana on, että muuttujan latauskäsky luodaan aktiivisen alueen alkuun, ja tallennuskäsky loppuun. Algoritmin oli tarkoitus pitää muuttujat rekisterissä silloinkin, kun niillä ei ole aktiivista aluetta, mutta algoritmia ei kehitetty pidemmälle aikarajan takia.

Kuvassa 15 on nykyisen algoritmin ohjelman koodille luoma latauskartta. Kuvassa vasemmalla on ohjeita peräkkäin, joiden yksinkertaistettu sisään- ja ulostulotieto ovat näytetty. Muuttujat ja rekisterit ovat värikoodattu. Sisääntulotiedon ”var”-solu on jonkin muuttujan tunniste, ja oikea solu on rekisteri, mihin tämä muuttuja on ladattava. Sisääntulossa muuttujan rekisteri voi olla musta, mikä tarkoittaa, että VE päättää rekisterin. Ulostulotiedossa vasemmalla on rekisteri, jonka arvoa ohje muuttaa. Ulostulossa oikealla voi olla muuttuja, minkä uusi arvo on tässä rekisterissä. Ulostulon muuttuja voi olla musta, mikä tarkoittaa, että tämä rekisteri ja sen sisältämä muuttuja ylikirjoitetaan ei-toivotusti. Kuvassa oikealla on ohjelmalle luotu minimaalinen latauskartta. Kartassa ylhäällä on muuttuja, ja alaspäin menevät viivat kertovat missä rekisterissä muuttuja on minkäkin ohjeen kohdalla. Keltavihreä väri tarkoittaa, että VE voi valita minkä tahansa rekisterin.



Kuva 15. Ohjelman latausalueet

Kuvassa 15 esitetty algoritmin luoma kartta sisältää tarvittavan tiedon lataus- ja tallennusohjeiden luomiseen. Malliohjelman käsittely aloitetaan sijoittamalla ohjeisiin algoritmin muuttujille päättämät rekisterit. Jäljellä ovat vielä lataus- ja tallennusohjeet.

Latausohje lisätään jokaisen aktiivisen alueen alkuun lataamaan alueen muuttuja kaikkiin sille päätettyihin rekistereihin (rekisterejä voi olla monta, jos sama ohje vaatii muuttujan useassa eri rekisterissä). Toinen paikka mihin latausohje lisätään, on alueen lopussa oleva hyppyohje. Hypyn kohteesta noudetaan kaikki siinä olevat aktiiviset muuttujat, ja näille luodaan latausohjeet ennen hyppyohjetta.

Tallennusohjeet vaativat hieman lisää työtä. Muuttuja tallennetaan vain, jos jokin ulostulo on sitä muuttanut. Tämä tapahtuu tutkimalla aktiivisen alueen ohjelmaa, jos jonkin ohjeen ulostulo päivittää alueen muuttujan arvon, niin silloin merkitään, että on tallentamaton muutos. Jos ohjelman lopussa on tallentamattomia muutoksia, niin silloin loppuun lisätään tallennusohje.

Nyt kun rekisterit on sijoitettu, ja lataus- ja tallennusohjeet on lisätty ohjelmaan, niin ohjelma on muuttujien puolesta toiminnallinen. Kuitenkin kaikille muuttujille on vielä varattava tila pinomuksista, mutta tämä hoidetaan muualla. Algoritmi jo loi latausohjeet, vaikka pinomuistia ei ole varattu. Tämä onnistuu siksi, että tämän pinomuistin kehyksen muoto on jo tiedossa, vaikka sen varaavaa koodia ei ole lisätty ohjelmaan. On myös tiedossa, että tähän kehykseen viitataan RBP-rekisterillä, niin tästä johtuen VE pystyy luomaan lataus- ja tallennusohjeet.

7.3.3 Rekisterivaraaja tulevaisuudessa

Nykyään VE pitää muuttujan rekisterissä vain sen aktiivisella alueella. VE:n oli tarkoitus pitää muuttujat rekisterissä aina, kun se on mahdollista. VE:n kuului myös luoda optimointeja, kuten nostaa lataus- ja tallennusohjeet ulos silmukasta. Joitakin osia tämän vaatimasta rakenteesta on, mutta aikarajan takia näitä optimointeja ei tehty loppuun. Optimoinnit ovat tietenkin optimointeja, eivätkä tarpeellisia asioita toiminnan kannalta. VE:stä oli vanha versio, joka kykeni luomaan täydellisen optimoitua koodia, mutta tämä algoritmi kaatui hyppyihin. Uudella VE:llä on joitakin rajapintaongelmia, joita vanhalla ei ollut. Nämä rajapintaongelmat aiheuttavat optimointiongelmia, mutta teoriassa nämä voidaan ohjelmaa käsittelemällä optimoida pois. Rajapinta myös vaatii monimutkaiset algoritmit toteutukselta. Vaikka nämä asiat eivät ole kovin vakavia, niin todennäköisesti tulevaisuudessa VE silti toteutetaan jollakin toisella tavalla.

Yksi täysin erilainen tapa toteuttaa VE, on ympyrävaraaja. Ympyrävaraaja on hyvin yksinkertainen rekisterivaraaja, joka antaa muuttujalle aina pisin aika sitten annetun rekisterin. Tästä tulee nimi ympyrävaraaja. Ympyrävaraajaa voidaan käyttää perustana monimutkaisemmalle varaajalle, joka kykenisi optimointeihin. VE mahdollisesti toteutetaan tulevaisuudessa jonkinlaisena ympyrävaraajana.

7.4 Virtuaalikoneen käskyjen toteutus

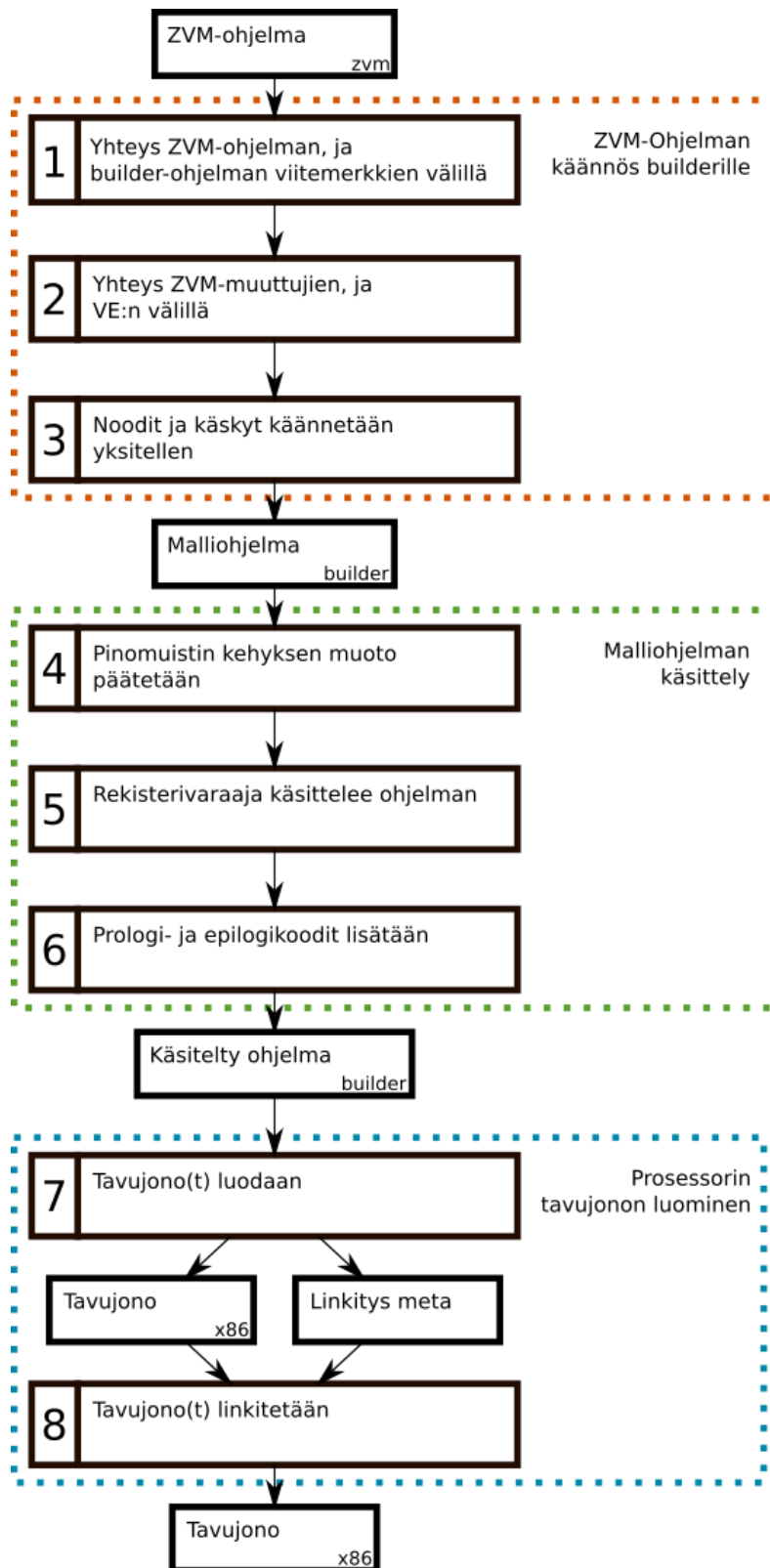
Monet ZVM-käskyt voidaan melko suoraan kartoittaa x86-prosessorin ohjeiksi, mutta jotkin ohjeet vaativat monimutkaisemman toteutuksen. Joskus ohjeen toteuttamiseen vaaditaan tietynlainen ohjelman rakenne. ZVM-käskyt voivat myös ottaa parametriksi hyvin paljon erilaista tietoa, kun taas x86-prosessorin ohjeet ovat paljon rajoittuneimpia. Tästä syystä monet ZVM-käskyjen muodot voivat luoda useita x86-ohjeita.

Käskyjä toteuttaessa on otettava huomioon muutamia asioita, jotka tulevat vastaan melko usein. Prosessori ei pysty liikuttamaan tietoa muistista muistiin, kun taas ZVM-ohjeet pystyvät. Tällöin on luotava ylimääräinen ohje, joka lataa tietoa muistista rekisteriin. Usein ohjeisiin ei voida myöskään koodata kahdeksantavuista vakioarvoa. Tällöin tarvitaan ylimääräinen ohje lataamaan vakioarvon rekisteriin. Tämän lisäksi monet ZVM-käskyt sallivat määrittää ulostulo muuttujan, kun taas x86-prosessorin ohjeet yleensä käyttävät toista lähdeoperandia ulostulona. Tästäkin voi seurata monta x86-prosessorin ohjetta.

Vain demokieltä varten tarvittavat ohjeet, ja niiden muodot ovat toteutettu. Tämä johtuu aikarajasta, ja siitä, että ZVM-käskyjä tai ylipäättään koneen ohjelmien kuvausta on tarkoitus parantaa tulevaisuudessa. Jotkin ohjeet ovat toteutettu tarkoittamaan ihan muuta kuin mitä niiden oli alun perin tarkoitus tehdä. Kaikki projektissa käytetyt virtuaalikoneen käskyt tai toiminnot, niiden täsmällinen toteutustapa, ja niille käytetyt prosessorin ohjeet ovat selitetty liitteessä 1.

7.5 Käännösprosessi

ZVM-ohjelman kääntäminen prosessorille tapahtuu vaihe vaiheelta. Nämä vaiheet ovat esitetty kuvassa 16.



Kuva 16. ZVJ:n käänносprosessi.

Kuvan 16 vaiheet ovat selitetty tarkemmin tässä.

1. Ensimmäiseksi ZVJ lukee ZVM-ohjelman, ja luo builderin viitemerkin jokaiselle ZVM-ohjelmasta löytyvälle viitemerkkinoodille. Näille viitemerkeille luodaan yhteys niin, että builderin viitemerkit tunnistaa ZVM-ohjelman viitemerkeistä.
2. Muuttujien luontiohjeet, SCOPEPUSH ja ALLOCT noudetaan ohjelmasta ja käsitellään. Näihin perustuen luodaan tietokanta kaikista käytetyistä muuttujista. Tietokanta tarjoaa ZVM-muuttujille tunnisteet, joita VE käyttää.
3. ZVM-ohjelman noodit käsitellään järjestyksessä alusta alkaen.
 - a. Kun vastaan tulee viitemerkkinoodi, niin vastaava builderin viitemerkki rekisteröidään.
 - b. Kun vastaan tulee ohje, niin tälle ohjeelle luodaan sovittu toteutus. Ohjeiden toteutukset on listattu liitteessä 1. Muuttujien luontiohjeita ei enää käsitellä toista kertaa.

Näiden toimenpiteiden jälkeen tuloksena on builder malliohjelma, joka sisältää x86-ohjeet, ja ohjeiden muuttujien käyttöä kuvaavat VE:n käyttäjätietonoodit. Luodun builder-malliohjelman ohjeiden operandeissa on hyvin monia rekisterejä, joita ei olla vielä päätetty. Tämä malliohjelma käsitellään seuraavasti.

4. Pinomuistin kehyksen muoto päätetään muuttujiin ja muuhun upotettuun tietoon perustuen. Tällä hetkellä ohjelmaan ei upoteta mitään tietoa, mutta tämä ominaisuus on olemassa.
5. Nyt kun pinomuistin kehyksen muoto on päätetty, niin ohjelma lähetetään käsiteltäväksi rekisterivaraajalle. Rekisterivaraaja sijoittaa ohjelmaan muuttujien rekisterit, latausohjeet ja tallennusohjeet.
6. Ohjelman alkuun ja loppuun sijoitetaan prologi- ja epilogikoodi vastaavasti. Nämä koodit suorittavat seuraavat toimenpiteet.
 - a. Koodit tallentavat ja lataavat kaikki rekisterit, joiden sisältöön kutsuttu funktio ei saa binääritason standardin mukaan koskea.

- b. Koodit lisäävät ohjelman koodin, joka varaa pinomuistista tilan funktion kehykselle.
- c. RBP-rekisteri asetetaan osoittamaan pinomuistissa olevaan kehykseen rekisterivaraajan lisäämiä ohjeita varten.

Builder malliohjelma on nyt käsitelty. Lopullinen ohjelma on vielä abstraktissa muodossa, joten sille on luotava tavujono.

- 7. Käsitellylle ohjelmalle luodaan tavujono builderilla. Builder luo myös metatiedot mahdollisille ratkaisemattomille viitemerkkien linkeille.
- 8. Luodut tavujonot ja linkitystiedot annetaan linkittäjälle. Tämän tuloksena on tavujono, joka voidaan suorittaa.

Projektissa ei tällä hetkellä käytetä enempää kuin yhtä builderia, mutta linkittäjä on silti ajettava. Linkittäjä ei ainoastaan yhdistä builderin ohjelmia, vaan myös samalla kopioi koodin muistiin, millä on erityinen suoritusoikeus. Tätä oikeutta tarvitaan ajamaan ohjelma Linux-ympäristössä. Muisti, jolla on tämä oikeus, varataan ”mmap”-funktioilla. Tämän funktion suorittaminen ei tarvita erityis oikeuksia. [34]

Ohjelmaan voidaan nyt ottaa funktio-osoitin C++-koodissa, ja tätä funktiota voidaan kutsua normaalisti.

7.6 Liukulukujen toteutus

ZVJ ei tällä hetkellä tue liukulukuja. Liukuluvut eivät kuitenkaan ole mikään ongelma. Liukulukuja on jo käsitelty paljon testikoodissa.

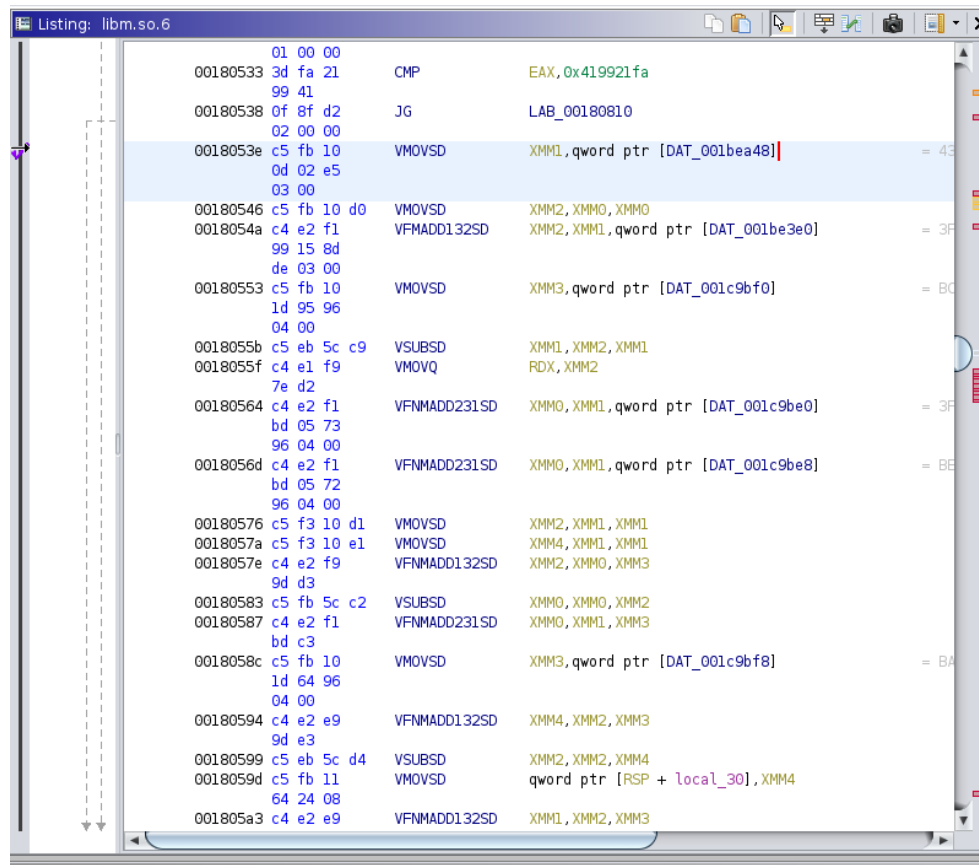
Prosesorilla liukulukuja voidaan käsitellä monilla eri ohjejoukoilla. Näistä vanhin joukko on FPU-ohjeet, ja uudemmat joukot ovat eri versiot SSE- ja AVX-ohjeista. Nämä kaikki ohjejoukot menevät toiminnallisuuden kanssa päällekkäin. FPU:n toiminnan ja modernimpien SSE:n ja AVX:n välillä on suuri ero, mutta SSE- ja AVX-ohjeet ovat melko lähellä toisiaan. Monet AVX-ohjeet ovat vain parannuksia SSE-ohjeista. SSE- ja AVX-ohjeet ovat ehkä parhaiten tunnettu SIMD-toiminnallisuudesta, mutta nämä ohjeet soveltuvat yhtä hyvin skalaareille.

FPU:lla on muusta prosessorin arkkitehtuurista hyvin erillään oleva ympäristö [38, s. 201], mistä voi todennäköisesti syyttää sitä, että historiallisesti FPU on ollut keskusyksiköstä erillään oleva prosessori [46]. Nykyään CPU sisältää integroidun FPU:n [46], mutta FPU:n ohjelmointirajapinta ei muutu. FPU toimii ensin lataamalla tietoa FPU:n pinomuistin tyyliin toimimaan rekisterijoukkoon, ja sitten käsittelemällä tietoa näissä rekistereissä. Ohjelmointirajapinnassa on ongelmia, esimerkiksi monet operaatiot eivät voi käyttää muuta, kuin pinomuistin päällä olevaa lukua, yleensäkin FPU:n rekisterejä ei voida suoraan käyttää ohjeissa.

SSE-ohjeet toteuttavat kaikki peruslaskuoperaatiot liukuluvuille, niin kuin FPU:kin, joten SSE:tä voidaan käyttää korvikkeena vanhalle FPU:lle. SSE on huomattavasti paremmin integroitavissa muuhun koodiin. SSE on myös yksikertaisesti nopeampi [22, s. 140]. FPU sisältää ohjeet trigonometrisille funktioille, mitä SSE- ja AVX-ohjeet eivät tarjoa. Tämä vaikuttaisi olevan ainut syy käyttää FPU-ohjeita paikoittain laskemaan nämä operaatiot. Kuitenkin optimisaatio ohje sanoo, että monissa tapauksissa kirjastoratkaisu näiden operaatioiden toteuttamiseen voi olla nopeampi kuin FPU-ohjeiden käyttäminen [22, s. 140].

Vertasin FPU:n tarjoamien trigonometristen funktioiden laskemiseen tarkoitettujen ohjeiden tehoa näille funktioille saatavilla olevaan kirjastoratkaisuun. Suoritin seuraavan kokeen. C++-kielellä tehtiin lyhyt koodipala, joka kutsuu "sin"-kirjastofunktiota silmukassa 100000000 kertaa. "sin"-funktion esittely löytyy "math.h"-tiedostosta. Testikoneessa on koneen Intel i7-8850H- prosessori ja Linux-pohjainen käyttöjärjestelmä. Testikoodi käännettiin GCC:n C++-kääntäjällä käyttäen optimointilippua "-O2". Koska tässä tapauksessa kääntäjä optimoi kutsusilmukan pois, niin silmukan ehto asetettiin volatileksi, jotta tämä saadaan estettyä. Koodin suoritukseen meni hyvin lähelle

yksi sekunti. Kun luotu koodi purettiin, ja sitä tutkittiin, niin koodi suoritti dynaamisen linkin ratkaisufunktion "sin"-funktioille. Dynaamisen linkin ratkaisu lopulta kutsui "sin"-funktia. "sin"-funktion toteutus löytyi tiedostosta "/usr/lib/libm.so.6". Toteutus on jokin algoritmi, mikä koostuu suuresta määrästä ehtoja ja AVX-ohjeita. Toteutus ei käyttänyt FSIN-ohjetta, tai ylipäättään yhtään FPU-ohjeita. Kuva 17 on kuvakaappaus puretusta "sin"-kirjastofunktiosta keskeltä sen koodia. Funktio on paljon pidempi kuin mitä kuvaan mahtui.



Kuva 17. "sin"-funktion toteutuksen osa.

Tämän lisäksi koontikielellä luotiin käsin tehty funktio, joka käytti FSIN-ohjetta laskemaan sinin. Tällä korvattiin silmukan "sin"-funktio. Kääntäjä kopioi oman "sin"-funktion koodin silmukan sisälle, ja lopulta "sin"-funktion laskuun käytettiin vain tämä yksi ohje. Silti tässä meni kolme sekuntia.

FSIN on siis poikkeuksellisen hidas ohje, mikä kannattaa korvata kirjastofunktiolla. Kirjastofunktion suorittamiseen käytettiin satoja ohjeita, ja useita funktiokutsuja, mutta kirjastofunktio oli silti kolme kertaa FSIN-ohjetta nopeampi. Tarkkaa syytä tämän ohjeen hitaudelle ei ole tiedossa.

Kokeita suoritettiin myös normaaleille laskutoimituksille, kuten jakamiselle ja kertomiselle. Tuloksena oli, että GCC:n C++-kääntäjä ei ikinä käyttänyt FPU-ohjeita, vaan ainoastaan SSE-ohjeita.

Liukulukujen toteutus tullaan siis tekemään SSE- tai AVX-ohjeilla. FPU:n käyttöön ei ole mitään syytä. SSE- ja AVX-ohjeet käyttävät omaa rekisterijoukkoa irrallaan muista yleistäytöisistä rekistereistä, mutta näiden rekisterien käyttämisessä ei ole paljoa eroa. Kun ZVJ tulevaisuudessa tukee liukulukuja, niin kaikki mitä tarvitaan, on eri rekisterijoukon ja ohjeen valitseminen riippuen muutujan tyypistä. VE:tä voidaan helposti laajentaa näillä ominaisuuksilla.

7.7 Matalan tason kääntäjä tulevaisuudessa

ZVM-formaatti uudistetaan ennen kuin tämän formaatin kääntäjää lähdetään miettimään. Näistä muutoksista ilman muuta seuraa laajoja muutoksia ZVJ:hin. Minkälaiselle ZVJ tulevaisuudessa näyttää, ei ole paljoa tietoa. Esimerkiksi jos ZVM ei tulevaisuudessa käytä muuttujia ollenkaan, niin valtaosa tästä dokumentista ei enää pidä paikkaansa.

Pieniä ongelmia on myös ohjeiden korjauksella builderilla. Builderi ei voi korjata ohjeita yhtä tehokkaasti, kuin ZVJ voi tehdä päätöksiä käytetyistä ohjeista. Ohjeiden korjaus saatetaan poistaa, ja vastuuta saatetaan antaa enemmän ZVJ:lle. Suurin muutos tässä olisi se, että builderi ei sisällä viitejärjestelmää, vaan viitejärjestelmä kuuluisi toteuttaa korkeammalla tasolla.

8 Demokieli

Demokieli on luotu toimimaan mahdollisimman yksinkertaisena esimerkkikielenä näyttämään, että virtuaalikoneelle on mahdollista toteuttaa ohjelmointikieliä. Demokieleltä ei siis vaadita paljoa. Projektin alkuvaiheilla ideana oli kehittää oikeasti hyödyllinen kieli, jota voisi käyttää jokapäiväisesti. Demokielellä ei ole mitään tekemistä tämän kanssa. Demokieli on hyvin yksinkertainen kieli, joka kykenee peruslaskutoimituksiin, muistinhallintaan ja asioiden tulostukseen konsolille. Kieli ei tue mitään ominaisuuksia, joilla voisi abstraktoida ohjelmaa, kuten tietorakenteita tai kielellä määritettäviä funktioita. Virtuaalikoneen ominaisuuksien ansiosta kieli tosin tukee kielen ulkopuolella määritettäviä C++-kielellä luotuja funktioita.

Kielellä määritettävien funktioiden toteutus nykyisellä virtuaalikoneella olisi hyvin epätehokasta, joten tähän ei kannata ryhtyä, ennen kuin virtuaalikoneetta päivitetään. Tietorakenteet voitaisiin toteuttaa (tosin myös hieman epätehokkaasti), mutta kielen kannalta nämä eivät lisää paljoa arvoa, koska kielestä puuttuvat kaikki mahdollisuudet abstraktoida koodia. Tietorakenteiden toteutus vaatisi kieleltä tyyppijärjestelmän, mikä tarkoittaa kohtalaista määrää työtä. Tyyppijärjestelmä kielen puolella ei ole teknisesti kovin mielenkiintoinen, joten sitä ei lähdetty tekemään.

Tämä kieli käännetään ZVM-ohjelmaksi, joka taas käännetään eteenpäin x86-prosessorille.

8.1 Syntaksi

Syntaksi on rekursiivista kuten monilta ohjelmointikieliltä odotetaan. Rekursiivisuus toteutettiin, koska se oli nopeaa toteuttaa siitä syystä, että se ei vaadi mitään monimutkaisia järjestelmiä Demokielen kääntäjältä.

Kuvassa 18 on Demokielellä tehty ohjelma, joka laskee kymmeneen.

```

i64 num 10;
i64 lineend malloc 16;
mw lineend 10;
mw add lineend 8 0;

while num {
    printnumber sub 10 num;
    print lineend;
    num sub num 1;
};

free lineend;
return 0;

```

Kuva 18. Demokielellä tehty laskuri

Kieli sisältää vain ilmaisuja (engl. expression) ja lauseita (engl. statement). Näiden ero on, että ilmaisu palauttaa aina arvon, ja lause ei ikinä. Kumpikin on ulkonäöltään hyvin samannäköinen. Yllä olevassa kuvassa lauseita ovat esimerkiksi `free`, `while` ja `"num sub num 1;"`. Ilmaisusta esimerkkinä on lauseessa parametrina oleva `"sub num 1"`.

`Free`:stä seuraa funktiokutsu, mutta sitä silti kutsutaan lauseeksi, koska kieli tunnistaa vain lauseen ja ilmaisun olemassaolon, mutta ei funktiokutsua. Funktiokutsua voidaan käyttää jonkin operaation toteutuksena, kuten tässä tapauksessa on tehty `free`-lauseen kanssa. Toisena esimerkkinä lukuja voidaan laskea yhteen `add`-ilmaisulla, ja tämän syntaksi vastaa sitä mille funktiot näyttävät monissa kielissä, mutta `add`-ilmaisusta ei tule funktiokutsua.

Ilmaisulla ja lauseella voi olla parametrejä vasemmalle kirjoitettuna. Parametrien tyyppejä ovat muuttujan nimi (`IDENT`), ilmaisu (`EXPRESSION`) ja koodipala (`PBLK`). Ilmaisu parametrilla on vielä monta eri tyyppiä. Nämä tyypit ovat vakioarvo, muuttujan nimi ja ilmaisu. Ilmaisu voi siis rekursiivisesti sisältää parametrinä lisää ilmaisuja.

Ilmaisun tai lauseen parametri voi ilmaisun lisäksi olla muuttujan nimi. Muuttujan nimi on merkijono, joka ei voi olla sama, kuin mikään kielen varaama sana kuten `"while"` tai `"free"`. Myös ilmaisu voi olla muuttujan nimi, mutta tämä parametri on sitä varten, että joihinkin paikkoihin ei voi antaa muuta parametria kuin muuttujan. Yksi tämänlainen tapaus on muuttujan asetus lause, joka voi asettaa vain muuttujan, ei laskutoimitusta.

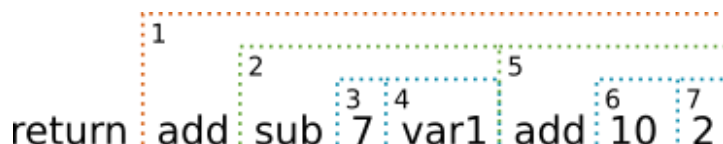
Ilmaisun tai lauseen koodipala-parametri on pala koodia, mikä on erotettu muusta ohjelmasta hakasulkeilla (`"{"`, `"}"`). Koodipala-parametri löytyy kuvasta 18 `while`-silmukassa. Yksityiskohtana

voidaan ajatella, että koko ohjelmalla on juurikoodi, mikä on koodipala, mutta tätä ei vain ole suljettu sulkeilla, kääntäjän toteutus käsittelee tiedoston tähän nojaten.

8.2 Parametrien jäsennys

Jäsennystä käsiteltäessä tulee esille käsite ”tokeni”. Tässä tapauksessa tämä tarkoittaa yksittäistä sanaa koodirivillä. Koodirivi koostuu listasta tokeneita.

Parametrit jäsennetään odottamalla, että jokaiselle lauseelle ja ilmaisulle annetaan tietty määrä parametrejä. Jäsennys muistuttaa funktioiden parametrien jäsennystä monissa kielissä, mutta parametrejä ei ole eroteltu pilkuilla tai suluilla. Ilmaisun tai lauseen parametrien jäsennys toimii rekursiivisesti. Parametrejä tutkitaan vasemmalta aloittaen, kääntäjä ajetaan rekursiivisesti operaa- tion ensimmäiselle parametrille, ja rekursiivisesti myös sen parametreille. Kun yksi parametri ja sen parametrit ovat jäsennetty rekursiivisesti, niin tiedossa on, että kuinka monta tokenia parametri, ja sen parametrit käyttivät riviltä. Tämän parametrin jälkeen alkaa välittömästi toinen parametri, joka jäsennetään samalla lailla. Tämä on helpompi hahmottaa tutkimalla koodiriviä, ”return add sub 7 var1 add 10 2;”. Kuvassa 19 on ratkaistu parametrien rajat jäsennyksen sääntöjen mukaan. Tämän lisäksi järjestys, missä parametrit jäsennettiin, on kuvattuna numerolla.



Kuva 19. Parametrien jäsennys.

Koodirivissä on puurakenne. Ohjelmassa on return- lause, joka ottaa parametriksi ilmaisun, tämä ilmaisu on ilmaisu add, ilmaisulle add annetaan kaksi parametriä, jotka ovat ilmaisut sub ja add, nämä ilmaisut puolestaan ottavat parametreiksi ilmaisuja, aivan kuten aikaisemminkin, mutta tällä kertaa ilmaisut ovat vakioarvoja ja muuttujien nimiä. Kielen jäsennyksen toteutus noudattaa samaa ajattelutapaa.

Tämä koodirivi suorittaa laskutoimituksen,

$$f(v_1) = (7 - v_1) + (10 + 2) .$$

8.3 Kielen tukemat operaatiot

Kaikki demokielen tukemat operaatiot on listattu liitteessä 2.

8.4 Muistinhallinta

Demokieli tukee muistinhallintaa. Muistia varataan malloc- ja free-operaatioilla. Nämä operaatiot kutsuvat suoraan C-kielen malloc- ja free-funktioita. Tämä lisäksi kielessä on mw- ja mr-operaatiot, joilla kirjoitetaan ja luetaan muistia vastaavasti. mw-lause ottaa kaksi parametriä, joista ensimmäistä tulkitaan muistiosoitteena, ja toista arvona, mikä kirjoitetaan tähän osoitteeseen. mr-ilmaisu ottaa yhden parametrin, mitä tulkitaan muistiosoitteena, mistä luetaan muuttuja, joka palautetaan. Demokieli tunnistaa vain kahdeksantavuisen kokonaisluvun olemassaolon, joten mw ja mr käsittelevät tämän kokoista varausta. malloc-funktiota kutsutaan suoraan annetulla parametrilla, joten demokielen käyttäjän pitää itse syöttää kahdeksan kerronnainen muistia varattaessa.

8.5 Teksti

Demokieli tunnistaa vain numeron olemassaolon, mutta ei merkin. Ainut ominaisuus mikä käsittelee merkkejä, on print-lause. print-lause tulkitsee annettua numeroa muistiosoitteena kahdeksantavuisiin numeroihin, mitkä vastaavat merkkejä. Ohjelmoija joutuu käyttämään muistinhallintaominaisuuksia varaamaan tilan tekstille, ja kirjoittamaan merkkien koodit tähän muistiin. Tekstin muistin osoitin voidaan antaa print-lauseelle. print-lause tulkitsee numeron 0 olevan tekstin loppu.

Jos ohjelmoija voisi käyttää lainausmerkkejä merkitsemään halutun tekstin, niin kääntäjä voisi itse ladata halutun tekstin muistiin. Tämä voi toimia kahdella tavalla. Ensimmäinen tapa on, että käyttäjä saa muistiosoitteen tähän kääntäjän varaamaan tekstiin. Muistin sisältö on vakioarvo. Muistin pitää olla vakioarvo, koska ohjelman ei kuuluisi muokata omaa koodiaan. Käyttäjä voi sitten ottaa tästä tekstistä kopioita itselleen, tai antaa vakioarvon sellaisenaan joillekin algoritmeille. Toinen tapa on, että käyttäjä käyttää jotakin malloc-ilmaisua vastaavaa ominaisuutta varaamaan muistin, ja tämä ominaisuus täyttäisi tämän muistin tekstillä.

Tapa että käyttäjä saa vakioarvon ei ole vielä mahdollista, koska demokieli ei tunnista vakioarvojen olemassaoloa, tai ylipäätään sisällä tyyppijärjestelmää. Toki käyttäjälle voidaan antaa tämä osoite joka tapauksessa, mikä sallisi käyttäjän muuttaa tätä tekstiä, ja aiheuttaa hullua käyttäytymistä. Ongelmia voi myös tulla, jos tämä teksti sijaitsee vain-luku-muistissa lopullisessa käännetyssä ohjelmassa.

Toinen tapa, että käyttäjälle on jokin teksti malloc-ilmaisu, voidaan toteuttaa ilman suuria vaivoja. Tämä vaatii uuden tekstiparametrin lauseille ja ilmaisuille, mutta sen toteutus on helppoa.

Tekstin käsittelyyn liittyy monia muitakin toimintoja, kuten tekstin lisääminen toisen perään. Tämä voidaan toteuttaa erilaisella add-ilmaisulla, joka lisää kaksi, nimenomaan tekstiä sisältävää muistia toisten perään, ja palauttaa uuden varatun muistin. Tekstin käsittelyä ei kuitenkaan lähdetty kehittämään, koska kunnollinen ratkaisu on, että ohjelmointikielessä on tekstityyppi joko sisäänrakennettuna, tai paremmin, luokkana. Demokieli kykenee käsittelemään tekstiä jo nyt, mutta nämä toimenpiteet ovat vaivalloisia.

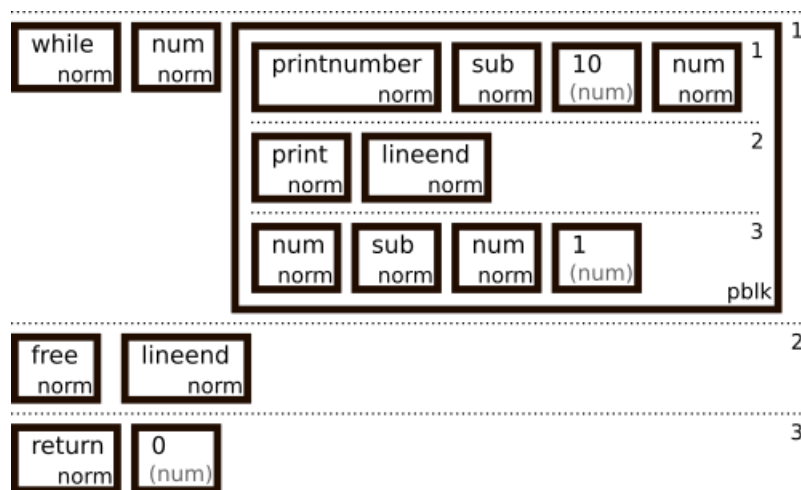
8.6 Kääntäminen virtuaalikoneen ohjelmaksi

Kieli on hyvin yksinkertainen, ja ZVM-formaattiin on helppo kääntää ohjelmia. ZVM-formaatti onnistuu hyvin abstraktoimaan alustan vaikeudet. Tämän lisäksi ZVM hoitaa itse monet asiat, kuten muuttujien kartoituksen muistiin. Toisin kuin prosessorin ohjeet, ZVM-käskyt sallivat sekoittaa hyvin erityyppistä tietoa parametreiksi. Näiden seikkojen ansiosta kääntäjässä on vain 750 riviä koodia.

8.6.1 Jäsennys

Ensimmäiseksi ohjelma tokenisoidaan. Tämä on eräänlaista leksikaalista analyysiä, joka ottaa raakatekstin sisään, ja jakaa tämän listaksi tokeneita [24, s. 6]. Tokeni on tietorakenne, joka sisältää yhden lauseen osan tekstin, ja tälle osalle annetun luokan. Tämä yksinkertaistaa ohjelman käsittelyä lopulle kääntäjälle. Lisäksi koodi ei ainoastaan tokenisoi ohjelmaa, vaan samalla jäsentää ohjelman eteenpäin listaksi rivejä. Rivi on lista tokeneista, mikä tekstissä päätettiin tokenilla ”;”.

Tokeni koostuu tekstimuotoisesta sanasta, ja tälle annetusta luokasta. Luokka voi olla ”normaali”, ”numero”, tai PBLK, eli koodipala. Numero tyyppiä ei kuitenkaan toteutettu oikeaan koodiin. Normaali tokeni on yksi välimerkein eroteltu merkkijono lauseessa, eli yksi sana. Numero tokeni on sama kuin normaali tokeni, mutta merkkijono sisältää vain numeroita. PBLK on tietorakenne, joka on lista lauseista. PBLK-tyypin tokeni löytyy if- ja while-lauseista. Johtuen PBLK-tyypin tokeneista lauseissa, ohjelmasta muotoutuu puu, jossa on yksittäisiä sanoja, tai jonkin sanan paikalla voi olla kokonainen uusi koodipala, joka taas sisältää lauseita ja tokeneita. Ohjelma tässä muodossa on pitkälle jäsennetty, ja vastaa syntaksin kuvausta hyvin. Osa kuvan 18 ohjelmasta on jäsennetty tähän rakenteeseen kuvassa 20. Kuvassa 20 on jäsennetty kolme riviä ohjelmaa. Yksi näistä on while-lause, jolla on PBLK parametri, joka sisältää kolme riviä lisää.



Kuva 20. Demokieli jäsennettynä.

8.6.2 Ohjelman kääntäminen

Jäsennyksen jälkeen ohjelmaa lähetään oikeasti kääntämään. Ensimmäiseksi haetaan kaikki muuttujien määrittelyt, ja yhteys luodaan muuttujan tekstimuotoisen nimen, ja numeerisen ZVM-muuttujan tunnisteen välille. Muuttujien nimet myös tarkistetaan siltä varalta, että nämä eivät käytä mitään kielen varaamia sanoja. Ohjelman alkuun lisätään SCOPEPEUSH- ja ALLOCT-käskyt kaikille muuttujille. Ohjelma voi tämän lisäksi vielä tarvita väliaikaisia muuttujia joitakin ilmaisia käännettäessä tallentamaan niiden parametreja. Tarvittavat väliaikaiset muuttujat eivät ole vielä tiedossa mutta ne lisätään myöhemmin.

Ohjelman kääntäminen onnistuu tämän jälkeen melko suoraviivaisesti. Ohjelma on jäsennetty listaksi rivejä. Ohjelman rivit voivat sisältää PBLK-tokeneita niin, että ohjelmasta muotoutuu puu. Ohjelman kehoa aletaan kasaamaan rekursiiviseen malliin ajamalla kääntäjä juurikoodipalalle. Toteutuksessa on oletus, että jokainen rivi ohjelmassa on lause, eikä rivi ala ilmaisulla. Nykyhetkellä jos rivi sisältäisi vain ilmaisuja, niin ohjelma ei tekisi mitään. Mikään ei kuitenkaan estä sitä, että rivit sisältäisivät vain ilmaisuja. Lauseita ovat muuttujan määrittäminen, muuttujan asetus, if-lause, while-silmukka ja return-lause. Monet lauseet puuttuvat tästä. Puuttuvat lauseet ovat toteutettu ulkoisena funktiona. Ulkoisilla funktioilla toteutetuilla lauseilla ja ilmaisuilla on erityinen toteutus. Näiden kohdalla suoritetaan nimen haku funktioiden tietokannasta. Yksittäiset lauseet käännetään seuraavasti.

- Muuttujan määrittäminen voi sisältää alkuarvon, joka asetetaan tämän lauseen kohdalla. Jos näin on, niin ilmaisuparametri käännetään asettamalla ulostulo kyseiseen muuttujaan. Ilmaisun kääntäjä luo kaikki ohjeet. Jos määrittäminen ei sisällä alkuarvoa, niin tälle lauseelle ei tehdä mitään, koska se on jo käännetty alussa.
- If-lause käännetään luomalla viitemerkki lauseen loppuun. Lauseen alkuun sijoitetaan COMPARE- ja BRANCH-ohje. Nämä ohjeet hyppäävät lauseen kehon yli, jos ehto ei ole totta. Kääntäjä ajetaan rekursiivisesti lauseen keholle, joka on PBLK-tyyppinen parametri.
- While-silmukka käännetään samaan tapaan kuin if-lause, mutta juuri ennen silmukan lopun viitemerkkiä lisätään ehdoton hyppy silmukan alussa olevaan tarkistukseen.
- return-lauseita voi olla monta, vaikka ZVM ei tue kuin yhtä PROGEXIT-ohjetta. Tämä on toteutettu niin, että return-lause hyppää ohjelman loppuun, mihinkä on lisätty yksi PROGEXIT-ohje. Paluuarvo toimii tallentamalla ohjeen ilmaisuparametri muuttujaan, joka koodin lopussa annetaan PROGEXIT-ohjeelle.
- mw-lauseen ilmaisu parametrit tallennetaan muuttujiin ja SET-ohjetta käytetään toteutukseen. SET-ohjeen ensimmäinen parametri on tyyppiltään data, joka viittaa annetun muuttujan muistiosoitteeseen. Toinen parametri on normaali muuttuja. Tämä on käännetty mr-ilmaisulle, joka päinvastoin lukee muistia.

Näiden lisäksi kääntäjä vielä kääntää ulkoisilla funktioilla toteutetut lauseet ja muuttujan asetukset. Toisin kuin edellisillä lauseilla, näille kummallekaan ei ole suoraan saatavilla staattista tunnistavaa tokenia. Muuttujan asetus lause alkaa muuttujan nimellä, mikä toimii tunnistavana tokenina. Ulkoisilla funktioilla toteutetut lauseet ovat toteutettu niin, että demokielen kääntäjällä on tietokanta, joka sisältää kaikkien kielessä käytettyjen funktioiden nimet, ja näille käytetyn ZVM-funktion. Kaikki nimet tässä tietokannassa ovat tunnistavia tokeneita. Muuttujat ja funktiot erotetaan nimen perusteella, koska muuttujien nimet eivät saa mennä päällekkäin funktioiden kanssa.

Muuttujien asetus -lauseet toteutetaan kääntämällä niiden ilmaisuparametri asettaen ulostulo haluttuun muuttujaan. Ilmaisun kääntäjä luo kaikki tarvittavat ohjeet.

Viimeisenä jäljellä ovat lauseet, jotka ovat toteutettu ulkoisilla funktioilla. Ulkoisten funktioiden kääntäminen tapahtuu ensin noutamalla "zvm_extfn"-tyypin tietorakenne funktioiden tietokannasta nimen perusteella. Tietorakenteesta käy ilmi funktion argumentit ja paluuarvo. Argumentit ovat vaihteleva määrä ilmaisuja. Argumenttien ilmaisujen tulokset tallennetaan väliaikaisesti muuttujiin ilmaisun kääntäjällä. Argumenttimuuttujien lista annetaan CALLE-ohjeelle funktion tietojen kanssa. Koska kyseessä on lause mikä ei palauta arvoa, niin tässä käytetään CALLE-ohjeen formaattia, joka ei sisällä ulostulo- parametria.

Kun koko ohjelma on käännetty, niin ohjelman loppuun rekisteröidään return-lauseen viitemerkki ja PROGEXIT-ohje. Tämän lisäksi ohjelman alkuun lisätään vielä luontikoodi kaikille tarvittaville väliaikaisille muuttujille. Tästä tulee valmis ZVM-ohjelma.

8.6.3 Ilmaisujen kääntäminen

Lauseiden kääntäjän lisäksi on ilmaisujen kääntäjä. Ilmaisut yleensä sisältävät ilmaisuparametreja, joten kääntäjä toimii rekursiivisesti. Ilmaisujen kääntäjälle kerrotaan muuttuja, mihin lopullinen ilmaisun tulos tallennetaan. Ilmaisujen kääntäjä voi käyttää useita väliaikaisia muuttujia tallentamaan tietoa tarvittaessa. Tarvittavat väliaikaiset muuttujat kerrotaan käyttäjälle. Käyttäjä pitää kirjaa siitä, että mitä muuttujia tarvitaan kaikkien ilmaisujen kääntämiseen. Kun ilmaisun toiminto on suoritettu, ja tulos on annettu ulostulomuuttujassa, niin käytettyjä väliaikaisia muuttujia voidaan käyttää uudestaan muille ilmaisuille.

8.6.3.1 Ilmaisujen toteutukset

Ilmaisuja ovat add/sub, mr, vertaukset, muuttujat, vakioarvot ja ilmaisut, jotka ovat toteutettu ulkoisilla funktioilla.

”add” ja ”sub” -ilmaisujen ilmaisu parametrit käännetään rekursiivisesti. Toinen parametri tallennetaan suoraan ilmaisulle asetettuun ulostulomuuttujaan, ja tähän muuttujaan lisätään, tai siitä vähennetään toinen muuttuja, käyttämällä ZVM:n ADD- tai SUB-ohjeita.

Muistin luku ilmaisu ”mr” lukee ilmaisuparametrin osoittaman muistiosoitteen. Tämän ilmaisuparametrin käännökseen ulostulo asetetaan asetettuun ulostulomuuttujaan. Tämän jälkeen ohjelmaan lisätään SET-ohje, jonka ulostulo parametriksi annetaan sama ilmaisun ulostulomuuttuja, ja sisääntuloksi sama ilmaisun ulostulomuuttuja, mutta tämä parametri viittaa muuttujan osoittamaan muistiosoitteeseen. Kaksi koodiriviä, jotka toteuttavat tämän ovat listattu tässä. Näillä koodiriveillä ”outvid” tarkoittaa ilmaisun kääntäjälle annettua ulostulomuuttujaa.

```
cexpret r = compile_expr(st, begintoken+1, outvid, ac, vars, glm,
dumystack);
ac.pi(zvm_instr::set(1, data(zvvar(outvid)), data(zvvar(outvid), true,
zvm_type::t_i64(), 0)));
```

Vertailu-ilmaisut palauttavat arvoksi joko yhden tai nollan riippuen täsmällisestä ilmaisusta. Ensimmäiseksi ulostulomuuttuja asetetaan arvoksi 1. Tämän jälkeen kummatkin ilmaisuparametrit käännetään, ja näitä verrataan COMPARE-ohjeella. Vertailun jälkeen lisätään ehdollinen hyppy-ohje, joka hyppää hypyn jälkeen olevan nollaksi asetuksen yli, riippuen täsmällisestä ilmaisusta.

Muuttuja- ja vakioarvoilmaisut koostuvat ainoastaan yhdestä arvosta. Tämä arvo asetetaan ulostulomuuttujaan.

Viimeiseksi jäljellä ovat ilmaisut, joiden toteutuksena käytetään ulkoisia funktioita. Nämä toimivat samalla tavalla kuin lauseetkin. Parametri- ilmaisut käännetään ja niiden lista annetaan CALLE-ohjeelle funktion tietojen kanssa. Tällä kertaa CALLE-ohjeelle annetaan paluuarvon tallennus muuttuja, joka on kääntäjälle ilmoitettu ulostulomuuttuja.

8.6.3.2 Ilmaisun kääntäjän optimoinnit

Koko demokielen kääntäjää ei ole optimoitu millään tavalla. Kuitenkin seuraava optimointi olisi helppo toteuttaa. Jos ilmaisuna on käytetty vakioarvoa, niin tämä vakioarvo on tallennettava muuttujaan. Mietitään lausetta `"var1 add 50 50;"`. Tässä `add`-ilmaisua käännettäessä kääntäjä rekursiivisesti kääntää vakioarvo ilmaisut, ja tallentaa ensimmäisen ilmaisuparametrin ulostuloon, ja toisen väliaikaiseen muuttujaan. Tämän jälkeen `add`-ilmaisun kääntäjä lisää ohjelmaan ohjeen, joka lisää nämä kaksi argumentin sisältävää muuttujaa yhteen.

Mahdollinen optimointi olisi sallia ilmaisun kääntäjän palauttaa vakioarvo sen lisäksi, että kääntäjä aina tallentaisi ilmaisun tuloksen muuttujaan. `add`-ilmaisun parametrit ovat vakioarvoja, joten näiden parametrien kääntäjä voisi palauttaa vakioarvon. Tämän jälkeen itse `add`-ilmaisun kääntäjä huomaa, että kummatkin parametrit ovat vakioarvoja ja kääntäjä voisi käännöksen aikana lisätä nämä yhteen luvuksi 100, eliminoiden yhteenlaskun. Tämä luku voidaan sitten tallentaa `var1`-muuttujaan suoraan. Tämä on alkeellinen muoto optimoinnista, mitä kutsutaan englanniksi nimellä *"constant folding"* [46].

9 Yhteenveto

Projektissa on onnistuneesti luotu sen päätavoitteena oleva kokonaisuus, missä omatekoinen ohjelmointikieli käännetään kuvitteellisen virtuaalikoneen ohjelmaksi, ja virtuaalikoneen ohjelmat käännetään eteenpäin x86-prosessorille. Kääntäjä toimii Linux-pohjaisessa PC-ympäristössä.

Virtuaalikoneen tavoitteena ovat olleet alusta- ja kieliriippumattomuus. Virtuaalikone on muuten alustariippumaton, mutta sille on rakennettu tuki suoralle muistin käyttämiselle osoittimien avulla. Suoraa muistinkäyttöä tarvitaan nykyistä Demokieltä varten, jotta sillä voidaan käsitellä merkkijonoja. Tämä on toteutuksessa otettu oikotie. Jotta täydellinen alustariippumattomuus saavutettaisiin, niin muistinhallinta on abstraktoitava. Toinen tärkeämpi virtuaalikoneen tavoite, sen riippumattomuus sille käännetystä kielestä, on saavutettu.

Virtuaalikoneessa on lukuisia rajoitteita, joiden takia esimerkiksi funktioiden toteutus käännettävään kieleen on epätehokasta. Tämä johtuu muuttujien käsittelyn ominaisuudesta, jonka avulla kutsuttu funktio voi käyttää kutsujan muuttujia. Ratkaisuksi ehdotettiin vaihtoehtoista muuttujien käsittelyä, missä funktion paikalliset muuttujat määritetään virtuaalikoneen ohjelman tietorakenteessa. Toinen pienempi ongelma on, että virtuaalikoneen käskyt käyttävät lippuja. Käskyn puhtaus helpottaisi ohjelman kääntämistä prosessorille. Virtuaalikoneen kehitys on seuraava askel projektissa. Virtuaalikoneen tavoite oli toteuttaa niin minimaalinen joukko ominaisuuksia, että sitä käyttämällä pystytään toteuttamaan mahdollisimman yksinkertainen kieli. Demokieli on toteutettu virtuaalikoneelle, joten tähän tavoitteeseen on päästy.

Demokielen tavoitteena oli olla yksinkertaisin mahdollinen kieli, jolla pystytään luomaan ohjelmia. Demokieli on luotu näyttämään, että virtuaalikoneelle voidaan toteuttaa ohjelmointikieli. Tavoitteensa mukaisesti Demokieli tukee vain hyvin pientä joukkoa ominaisuuksia. Demokieli kykenee ainoastaan laskutoimituksiin, asioiden tulostukseen konsolille, ja sillä voidaan luoda silmuja ja vertailuja. Ominaisuuksien pienen määrän takia ohjelmien luominen kielellä on vaivalloista, mutta kuitenkin mahdollista. Kieli on saavuttanut tavoitteensa. Virtuaalikoneen ominaisuuksien ansiosta Demokieli pystyttiin integroimaan C++-kielellä tehtyjen funktioiden kanssa. Tämä on merkittävä ominaisuus, koska pidemmälle kehitettynä C++-kielen laaja kirjastovalikoima voidaan integroida virtuaalikoneeseen ja saada näin kirjastojen toteuttama toiminnallisuus käännettävälle kielelle käytännössä ilmaiseksi.

Koska projektista tehtiin opinnäytetyö, niin kehittämiseen on vaikuttanut aikaraja, johon mennessä kokonaisuus piti saada toimivaksi. Tästä johtuen projektissa on kehitetty hieman kaikkea sen sijaan, että vain paria asiaa mietittäisiin pidemmälle. Tämä on pääasiallinen syy melkein kaikille projektissa esiintyville ongelmille. Toisaalta laaja kehittäminen on myös antanut arvokasta tietoa koko järjestelmän toiminnasta. Kun koko järjestelmän toiminta on tiedossa, siihen on helpompaa kehittää yksittäisiä komponentteja, koska näiden vastuu ja toiminta on paremmin tiedossa. Koska kiinnostus projektia kohtaan ei ole vielä laantunut, niin melkein kaikkea projektissa luotua tullaan vielä kehittämään tulevaisuudessa. Esimerkiksi virtuaalikoneen ongelmien ratkaisemiseksi on tehtävä laajoja muutoksia. Matalan tason kääntäjä kääntää virtuaalikoneen ohjelman eteenpäin prosessorille, joten tähän kääntäjään on tietysti myös tehtävä laajoja muutoksia. Lisäksi kun tulevaisuudessa aloitetaan kehittämään projektin alkuperäisenä ideana olevaa kieltä, niin Demokieli tullaan purkamaan kokonaan. Nämä muutokset koskevat suoraan kaikkia suuria osa-alueita projektissa.

Koska koko kokonaisuus piti saada toimivaksi ajoissa, niin kehitystyössä suosittiin yksinkertaisia mutta toimivia ratkaisuja. Projekti on melko monimutkainen. Vaikka yksittäiset asiat suunniteltiin hyvin, ja kehitettäisiin pitkälle, niin ei kaikki varmasti siltikään menisi kerralla oikein. Kun jokin asia ei menekään niin kuin piti, niin sitten pitkälle kehitettyjä ratkaisuja on purettava pois. Yksinkertaisella rajoittuneella kokonaisuudella aloittaminen tarjoaa hyvän aloituspisteen syvemmälle kehitykselle. Tästä syystä yksinkertaisten ratkaisujen suosiminen on todennäköisesti paras lähestymistapa.

Olen oppinut tämän projektin aikana hyvin paljon tietokoneiden toiminnasta. Syvempää tietämystä on myös tullut ohjelmointikielten toiminnasta. Tämä voi auttaa uusien ohjelmointikielten opettelua. Näiden lisäksi itse kääntäjän ohjelmoiminen on haastava projekti, joten tämä projekti on kehittänyt ongelmanratkaisutaitojani.

Tämän projektin jatkokehitykseen voidaan kuluttaa käytännössä loputtomasti aikaa. Ohjelmointikielen ja kääntäjän kehittäminen on aikaavievä prosessi. Näitä voi parannella käytännössä loputtomiin. Jos kiinnostusta riittää, niin ohjelmointikielelle voidaan myös tulevaisuudessa rakentaa kehitysympäristöt, ja muut työkalut.

Lähteet

1. Baker G. Mandelbrot Set Language Shootout. Saatavilla:
<https://coursys.sfu.ca/2021su-cmpt-383-d1/pages/ExampleMandelbrot>
2. Tierney L. Compiling R: A Preliminary Report. Proceedings of the 2nd International Workshop on Distributed Statistical Computing. 2001.
3. Baker G. Language Implementations. Saatavilla:
<https://ggbaker.ca/prog-langs/content/lang-implement.html>.
4. M. Anton Ertl, Gregg D. The Structure and Performance of Efficient Interpreters. Journal of Instruction-Level Parallelism 5. 2003.
5. Bytecode - Wikipedia. Saatavilla: <https://en.wikipedia.org/wiki/Bytecode>. Haettu 22.5.2021.
6. Agesen O, Detlefs D. Mixed-mode Bytecode Execution.
7. Kotlin Documentation – FAQ. Saatavilla: <https://kotlinlang.org/docs/faq.html>. Haettu 22.5.2021
8. The Java Language Specification. Saatavilla:
<https://docs.oracle.com/javase/specs/jls/se16/html/index.html>. Haettu 23.5.2021.
9. Java SE 16 Development Kit Documentation – Readme. Saatavilla:
<https://docs.oracle.com/en/java/javase/16/index.html>. Haettu 22.5.2021.
10. Java Virtual Machine – Wikipedia. Saatavilla:
https://en.wikipedia.org/wiki/Java_virtual_machine. Haettu: 22.5.2021.
11. Sarimbekov A, Podzimek A, Bulej L, Zheng Y, Ricci N, Binder W. Characteristics of Dynamic JVM Languages. Proceedings of the 7th ACM Workshop on Virtual Machines and Intermediate Languages. 2013.

12. Li W.H., White D.R., Singer J. JVM-hosted Languages: They Talk the Talk, but Do They Walk the Walk? 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools. 2013.
13. Python 3.9.5 Documentation - Glossary. Saatavilla: <https://docs.python.org/3/glossary.html>. Haettu 30.5.2021.
14. Cython kotisivu. Saatavilla: <https://cython.org/>. Haettu 30.5.2021.
15. GCC kotisivu. Saatavilla: <https://gcc.gnu.org/>. Haettu 23.5.2021.
16. PyPy kotisivu. Saatavilla: <https://www.pypy.org/>. Haettu 30.5.2021.
17. OCaml Manual – ocamlc. Saatavilla: <https://ocaml.org/manual/comp.html>. Haettu 30.5.2021.
18. OCaml Manual – ocamlrun. Saatavilla: <https://ocaml.org/manual/native.html>. Haettu 30.5.2021.
19. Smyth N. Ruby Essentials. Saatavilla: https://www.techotopia.com/index.php/Ruby_Essentials. Haettu 30.5.2021.
20. GraalVM – Reference Manual. Saatavilla: <https://www.graalvm.org/reference-manual/ruby/RuntimeConfigurations/>. Haettu 30.5.2021.
21. Cython – Documentation. Saatavilla: <https://cython.readthedocs.io/en/latest/src/quickstart/cythonize.html>. Haettu 5.6.2021.
22. Intel® 64 and IA-32 Architectures Optimization Reference Manual.
23. Labels as Values – GCC Documentation. Saatavilla: <https://gcc.gnu.org/onlinedocs/gcc/Labels-as-Values.html#Labels-as-Values>. Haettu: 22.5.2021.
24. Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman. Compilers: Principles, Techniques, and Tools.

25. The Java HotSpot Performance Engine Architecture. Saatavilla:
<https://www.oracle.com/java/technologies/whitepaper.html>. Haettu 23.5.2021
26. Ilyushin E, Namiot D. On source-to-source compilers. International Journal of Open Information Technologies, vol. 4, no. 5. 2016.
27. Source-to-source compiler – Wikipedia. Saatavilla:
https://en.wikipedia.org/wiki/Source-to-source_compiler. Haettu 23.5.2021.
28. ISO/IEC 9899:2018: Information technology — Programming languages — C.
29. Ahead-of-time compilation – Wikipedia. Saatavilla:
https://en.wikipedia.org/wiki/Ahead-of-time_compilation. Haettu: 23.5.2021.
30. Shared Libraries - The Linux Documentation Project. Saatavilla:
<https://tldp.org/HOWTO/Program-Library-HOWTO/shared-libraries.html>.
Haettu 23.5.2021
31. Windows app developer documentation - Dynamic-Link Libraries. Saatavilla:
<https://docs.microsoft.com/en-us/windows/win32/dlls/dynamic-link-libraries>. Haettu 30.5.2021.
32. How to write a JIT compiler. Saatavilla: <https://github.com/spencertipping/jit-tutorial>.
Haettu 30.5.2021.
33. Process – The Linux Documentation Project. Saatavilla:
<https://tldp.org/LDP/tlk/kernel/processes.html>. Haettu 5.6.2021.
34. mmap-funktion man-sivu. Saatavilla:
<https://man7.org/linux/man-pages/man2/mmap.2.html>. Haettu 28.3.2021.
35. reinterpret_cast – C++ reference. Saatavilla:
https://en.cppreference.com/w/cpp/language/reinterpret_cast. Haettu 5.6.2021.
36. Fog A. Calling conventions for different C++ compilers and operating systems. Saatavilla:
<https://www.agner.org/optimize/>. Haettu 26.3.2021.
37. System V Application Binary Interface AMD64 Architecture Processor Supplement. Saatavilla:

<https://raw.githubusercontent.com/hjl-tools/x86-psABI/x86-64-psABI-1.0.pdf>.

Haettu 13.4.2021

38. Intel® 64 and IA-32 architectures software developer's manual combined volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4.
39. Love R. Linux Kernel Development 3rd Edition. 2010. ISBN: 9780672329463.
40. x86 calling conventions – Wikipedia. Saatavilla:
https://en.wikipedia.org/wiki/X86_calling_conventions. Haettu 26.3.2021.
41. Visual Studio, Microsoft Portable Executable and Common Object File Format Specification. Saatavilla: https://download.microsoft.com/download/9/c/5/9c5b2167-8017-4bae-9fde-d599bac8184a/pecoff_v83.docx. Haettu 5.6.2021.
42. "what is the maximum size of a PE file on 64-bit Windows?" Saatavilla:
<https://stackoverflow.com/questions/6976693/what-is-the-maximum-size-of-a-pe-file-on-64-bit-windows>. Haettu 24.5.2021.
43. "Is it possible to run a larger than 4GB .exe?" Saatavilla:
<https://superuser.com/questions/667593/is-it-possible-to-run-a-larger-than-4gb-exe>.
Haettu 5.6.2021.
44. Gordon J. Writing 64-bit programs. Saatavilla:
<https://www.godevtool.com/GoasmHelp/64bits.htm>. Haettu 24.5.2021.
45. x87 – Wikipedia Saatavilla: <https://en.wikipedia.org/wiki/X87>. Haettu 28.3.2021.
46. Constant folding - The Free On-line Dictionary of Computing. Saatavilla:
<https://foldoc.org/Constant+Folding>. Haettu: 30.4.2021.

Liitteet

1 Virtuaalikoneen operaatiot ja niiden toteutukset

1.1 ADD ja SUB

ZVM-käsky	ADD		
Parametrit	data dest	data src1	data src2
Toiminto	dest = src1+src2		

ZVM-käsky	SUB		
Parametrit	data dest	data src1	data src2
Toiminto	dest = src1-src2		

ADD ja SUB ZVM-käskyt toteuttavat yhteen ja vähennyslaskun. Parametri ”dest” sisältää tuloksen. ”dest” voidaan asettaa samaksi muuttujaksi kuin lähdemuuttujat. ”dest”-parametri ei voi olla vakioarvo, jos ”dest” ei myöskään viittaa muistiin.

Nämä käskyt käyttävät samannimisiä x86-ohjeita ADD ja SUB. ADD- ja SUB-ohjeet toimivat samalla tavalla, mutta toinen laskee yhteen ja toinen vähentää. Ohjeiden ensimmäiseen operandiin lisätään tai siitä vähennetään toinen operandi. [38, s. 625, s. 1842]

Toteutus tapahtuu lisäämällä ohjelmaan x86:n ADD- tai SUB-ohje, ja tälle ohjeelle annetaan rekisterivaraajan VII-noodi. VII-noodin sisääntuloina ovat kummatkin muuttujat, ja ulostulona ensimmäisen operandin sijoitettu rekisteri, mikä menee ”dest”-parametriin. VE hoitaa muuttujien latauksen ja tallennuksen. ZVM-käsky voi lisätä kaksi muistia yhteen, ja ZVM sallii valita ulostulo muuttujan. x86-prosessorin ADD- ja SUB-ohjeet eivät voi ottaa kahta muistioperandia, tai ottaa vastaan ylimääräistä ulostulo operandia. Jos ZVM-käskyssä tallennus tapahtuu lähteeseen ”src1”, niin silloin operaatio käy suoraan yhteen x86:n ohjeen ”add | sub src1, src2” kanssa. Jos tallennus tapahtuu lähteeseen ”src2”, tai johonkin muuhun muuttujaan, niin silloin VE voi joutua lisäämään ylimääräisiä latauksia ja tallennuksia, koska src1 kirjoitetaan yli ei-halutusti. Yksi tapa välttää tämä

ongelma yhteenlaskun kanssa olisi vaihtaa operandien paikkaa, koska yhteenlasku on vaihdannainen. Tosin tätä ei oikeassa koodissa tehdä. Näistä ZVM-käskyistä syntyy suuri määrä parametri yhdistelmiä, joita ei voida suoraan antaa prosessorille. Suuri määrä näistä yhdistelmistä vaatii useamman x86-prosessorin ohjeen. Kaikkia parametri yhdistelmiä ei olla toteutettu, jos niitä ei olla käytetty demokielessä. Monet näiden käskyjen parametrien käsittelyn monimutkaisuuksista pätevät useille ZVM-käskyille.

1.2 SCOPEPUSH, SCOPEPOP ja ALLOCT

ZVM-käsky	SCOPEPUSH
Parametrit	
Toiminto	Luo uuden näkyvyysalueen muuttujille, ja aloittaa tunnisteet nollasta.

ZVM-käsky	SCOPEPOP
Parametrit	
Toiminto	Ottaa edellisen näkyvyysalueen taas käyttöön

ZVM-käsky	ALLOCT
Parametrit	zvm_type t
Toiminto	SCOPEPUSH-ohjeen alla määrittää uuden muuttujan.

Näitä käskyjä käytetään luomaan muuttujat. Nämä käskyt ovat melko huono idea. Lähitulevaisuudessa tarkoitus on päivittää ZVM-formaattia niin, että nämä käskyt otetaan kokonaan pois.

Toteutus ei mitenkään ota huomioon näiden alkuperäistä tarkoitusta, vaan sen sijaan käyttää AL-LOCT-käskyjä ainoastaan merkitsemään mitä muuttujia ohjelma sisältää. Kääntämisen alussa AL-LOCT-käskyt etsitään, ja näistä luodaan tunnisteet kaikille muuttujille. Nämä käskyt ovat osana määrittämässä ohjelman pinomuistin kehyksen muotoa. Nämä käskyt eivät luo ylimääräisiä prosessorin ohjeita. Kun ohjelmaa ruvetaan kääntämään näiden käskyjen käsittelyn jälkeen, niin näitä käskyjä ei enää oteta huomioon.

1.3 COMPARE

ZVM-käsky	COMPARE	
Parametrit	data src1	data src2
Toiminto	Vertaa parametrejä ja asettaa piilotetut liput.	

Tämä käsky toteutetaan suoraan x86-prosessorin CMP-ohjeella, joka asettaa prosessorin lippuja. [38, s. 755]. Näitä prosessorin lippuja käytetään suoraan ZVM:n hyppykäskyissä.

Tulevaisuudessa tämä käsky saatetaan päivittää varastoimaan tulos muuttujaan sen sijaan, että se asettaisi piilotettuja lippuja. Tuloksen tallentaminen muuttujaan sen sijaan, että se pidettäisiin lipuissa, ratkaisee hypoteettisen ongelman, jossa prosessorin liput ja ZVM-ohjelmasta ilmi käyvät liput eivät vastaa toisiaan. Tätä ongelmaa ei ole käytännössä tavattu. Jos tämä käsky tallentaa tuloksen muuttujaan, niin silloin prosessorin hyppyoheiden ympärille pitää lisätä tuloksen lataavaa koodia. Tosin jos käyttäjä käyttää tätä käskyä hypyn läheisyydessä, niin tämä koodi voidaan monissa tapauksissa eliminoida optimoineilla. Käännettävä kieli todennäköisesti voi myös tallentaa vertauksien tuloksia muuttujiin, niin silloin tämä muutos olisi mainio.

1.4 BRANCH

ZVM-käsky	BRANCH	
Parametrit	label dest	cndtype cond
Toiminto	Tutkii vertauskäskyn asettamia lippuja. Annetun ehdon tyypistä ("condt") riippuen hyppää kohtaan dest.	

x86-Proessorilla on hyvin monia eri hyppyohjeita, jotka hyppäävät tietyn siirtymän eri ehdon täyttyessä [38, s. 1107]. Näistä ohjeista valitaan sopiva perustuen annettuun ehdon tyyppiin "condt". Prosessorin hyppyohjeet lukevat EFLAGS-rekisterin lippuja [38, s. 1110]. Prosessorin CMP-ohjetta käytetään asettamaan näitä lippuja [38, s. 755]. CMP-ohje lisätään ohjelmaan ZVM:n COMPARE-käskyllä. Builderin viitejärjestelmä muuttaa viitemerkin sisältävän erikoisoperandin suhteelliseksi siirtymäksi. Jos COMPARE-käsky tulevaisuudessa tallentaa vertauksien tuloksia muuttujiin niin silloin BRANCH-käsky saa tämän muuttujan parametrina.

Taulukossa 1 on esitetty mikä hyppyohje valitaan riippuen ehdon tyypistä.

cndtype	Selitys	Käytetty x86-ohje
cndtype::NONE		JMP
cndtype::EQ	Yhtä suuri	JZ
cndtype::NEQ	Erisuuri	JNZ
cndtype::GT	Enemmän	JNLE
cndtype::LT	Vähemmän	JL
cndtype::GTE	Enemmän tai yhtä suuri	JNL
cndtype::LTE	Vähemmän tai yhtä suuri	JLE

Taulukko 1. Ehdon tyypin ja valitun ohjeen yhteys.

1.5 PROGEXIT

ZVM-käsky	PROGEXIT
Parametrit	data ret
Toiminto	Välittömästi sulkee ohjelman ja palauttaa "ret"-parametrin arvon takasin C++-koodille. Tämä käsky pitää olla ohjelman lopussa, ja näitä käskyjä ei voi olla kuin yksi

ZVM-ohjelman pitää sisältää tämä käsky. Tällä hetkellä useita PROGEXIT-käskyjä ei ole tuettu. PROGEXIT-käskyn täytyy olla viimeinen ohje. Tulevaisuudessa, kun funktioita tuetaan, niin tämän käskyn toiminta tulee varmasti vastaamaan enemmän return-lausetta ohjelman sulkemisen sijaan.

Toteutus lataa "ret"-parametrin arvon RAX-rekisteriin, ja aloittaa VE:n rajoitusalueen tälle rekisterille. Tämä rajoitusalue loppuu luodun malliohjelman lopussa. Koska tämän täytyy olla viimeinen käsky, niin ohjelman epilogi tulee heti tämän ohjeen jälkeen. Epilogi sisältää x86-prosessorin RET-ohjeen, joka palauttaa suorituksen takaisin C++-koodiin, joka käynnisti ohjelman [38, s. 1731].

Rekisteriä RAX käytetään varastoimaan paluuarvo, koska kääntäjän noudattama binääritason standardi määrittää funktion paluuarvon olevan tässä rekisterissä [37, s. 24].

Jos useita PROGEXIT-käskyjä ohjelmassa halutaan tukea, niin silloin tälle käskylle voidaan varata käyttäjälle näkymätön muuttuja, mihin PROGEXIT tallentaa "ret"-parametrin. Tämän jälkeen PROGEXIT voi lisätä ohjelmaan hyppyohteen epilogiin, missä tästä erikoismuuttujasta ladataan ohjelman paluuarvo. Tämä ei ole nykyistä ratkaisua mitenkään vaikeampi, tai vaadi mitään uutta toimintoa, mutta nykyinen ratkaisu oli nopeampi kirjoittaa.

1.6 CALLE

ZVM-käsky	CALLE		
Parametrit	zvm_extfn fn	data dest	data[] params
Toiminto	dest = fn(params)		

ZVM-käsky	CALLE	
Parametrit	zvm_extfn fn	data[] params
Toiminto	fn(params)	

”zvm_extfn” on tietorakenne, johonka on määritetty kaikki tarpeellinen tieto funktiosta sen kutsua varten. Tämä sisältää funktion osoitteen, argumentit ja paluuarvon.

CALLE-käsky on yksi mielenkiintoisimpia käskyjä. Sitä käytetään kutsumaan C++-kielellä määritettyä funktiota. Koska funktio on käännetty C++-kääntäjällä, niin silloin on mietittävä, kuinka binääritason yhteensopivuus C++-kääntäjän kanssa hoidetaan.

Toteutus luo VE:n rajoitusalueen kaikille rekistereille, joiden sisältöä kutsuttu funktio saa sysvabi:n mukaan muuttaa [37, s. 23]. Toteutus lähettää argumentit funktiolle pinomuistissa. Myös paluuarvo lähetetään takaisin pinomuistissa. Ensimmäiseksi toteutus varaa pinomuistista tilan funktion argumenteille ja paluuarvolle. Paluuarvo varataan vain, jos funktio palauttaa jotakin. Tämän jälkeen argumentit kopioidaan paikallisista muuttujista pinomuistiin. Tämän jälkeen osoittimet argumentteihin ja paluuarvoon, annetaan funktiolle rekistereissä RDI ja RSI, koska nämä ovat kaksi ensimmäistä käytettävää rekisteriä sysvabi:n mukaan [37, s. 22]. Tämän jälkeen funktiota kutsutaan x86:n CALL-ohjeella. C++-funktio kirjoittaa paluuarvon sille parametriksi annettuun osoitteeseen. Lopuksi paluuarvo ladataan pinomuistista kutsujan muuttujaan.

1 Demokielen tukemat operaatiot

Kielen kaikki ominaisuudet noudattavat hyvin yhtenäistä syntaksia, joka vastaa funktiota monissa kielissä. Kielen syntaksissa kummallakin, ilmaisulla ja lauseella on parametrejä. Parametrit kirjoitetaan ominaisuuden tunnistavan tokenin oikealle puolelle, kuin kyseessä olisi funktio. Ominaisuuden tunnistetokeni toimii funktion nimen tapaan, parametrit ovat parametrejä ja käsitys, että onko jokin lause vai ilmaisu vastaa sitä, että palauttaako funktio arvoa. Ilmaisua on palauttava funktio ja lause on funktio, joka ei palauta arvoa. Tähän perustuen kaikki ominaisuudet voidaan listata funktioina.

Listassa operaatioille on merkitty parametrit kuten funktioille. Parametrien tyypit ovat muuttujan nimi (IDENT), ilmaisu (EXPRESSION) ja koodipala (PBLK). Jos operaatio palauttaa arvon alla olevassa listassa, niin se on ilmaisu, ja se voidaan sijoittaa muuhun operaatioon ilmaisun paikalle. Listassa oleva kenttä (1. tokeni) tarkoittaa tämän operaation tunnistetokenia, joka tulee kirjoittaa ensimmäisenä. Muuttujan asetuksella tämä ensimmäinen tokeni on muuttujan nimi, tämä vastaa "="-operaattoria muissa kielissä.

Uusien muuttujien luonti.

Palautus	1. tokeni	Parametrit	Huom.
EI	i64	IDENT	
EI	i64	IDENT, EXPRESSION	Asettaa muuttujan arvon ohjeen kohdalla.

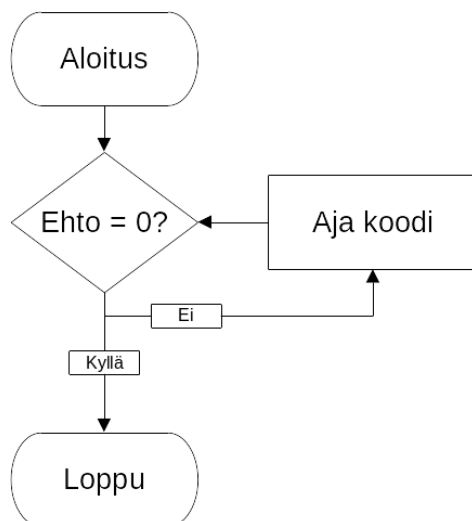
Muuttujan asetus.

Palautus	1. tokeni	Parametrit	Huom.
EI	IDENT	EXPRESSION	

While-silmukka.

Palautus	1. tokeni	Parametrit	Huom.
Ei	while	EXPRESSION, PBLK	

Silmukan ehtona käytetään ensimmäistä ilmaisuparametria, ja silmukan koodi annetaan toisena PBLK-parametrina. Ehtoparametrin tulos lasketaan ennen silmukan koodin suoritusta. Jos ehtoparametri antaa arvoksi nollasta poikkeavan arvon, niin silmukan koodi ajetaan. Jos ehtoparametri antaa tulokseksi nollan, niin silmukan koodin yli hypätään, eikä ehtoa enää tutkita. Jos ehto oli nollasta poikkeava arvo, niin kun silmukan koodi on ajettu, niin aloitetaan taas alusta tutkimalla ehdon arvoa. Tämä on kuvattu kuvassa 21.

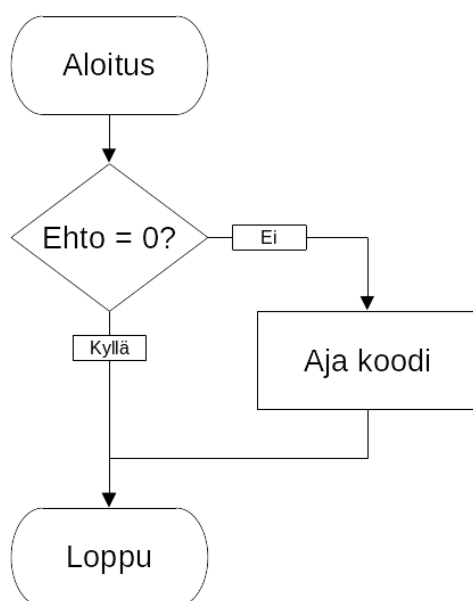


Kuva 21. While-silmukan vuokaavio.

If-lause

Palautus	1. tokeni	Parametrit	Huom.
Ei	if	EXPRESSION, PBLK	

Lauseen ehtona käytetään ensimmäistä ilmaisuparametria, ja lauseen koodina toista PBLK-parametria. Jos ehdon tuloksena on nollasta poikkeava arvo niin lauseen koodi ajetaan. Jos ehdon tulos on nolla niin lauseen koodi ohitetaan. Tämä on kuvattu kuvassa 22.



Kuva 22. If-lauseen vuokaavio.

Ohjelman lopetus ja arvon palautus.

Palautus	1. tokeni	Parametrit	Huom.
Ei	return	EXPRESSION	Saa olla useita vaikka ZVM ei tue tätä. Katso toteutus

return-lause lopettaa ohjelman ja palauttaa jonkin arvon ZVM-ohjelman käynnistäneelle C++-koodille.

Yhteen- ja vähennyslaskut.

Palautus	1. tokeni	Parametrit	Huom.
KYLLÄ	add	EXPRESSION, EXPRESSION	
KYLLÄ	sub	EXPRESSION, EXPRESSION	

Vertailu

Palautus	1. tokeni	Parametrit	Huom.
KYLLÄ	eq	EXPRESSION, EXPRESSION	Yhtä suuri
KYLLÄ	neq	EXPRESSION, EXPRESSION	Erisuuri
KYLLÄ	gt	EXPRESSION, EXPRESSION	Enemmän
KYLLÄ	lt	EXPRESSION, EXPRESSION	Vähemmän
KYLLÄ	gte	EXPRESSION, EXPRESSION	Enemmän tai yhtä suuri
KYLLÄ	lte	EXPRESSION, EXPRESSION	Vähemmän tai yhtä suuri

Vertailut palauttavat arvon yksi, jos kyseinen ehto on totta. Jos ehto ei ole totta, niin vertailut palauttavat arvon nolla.

Muistin varaus ja vapautus.

Palautus	1. tokeni	Parametrit	Huom.
KYLLÄ	malloc	EXPRESSION	Kutsuu malloc-funktiota.
EI	free	EXPRESSION	Kutsuu free-funktiota.

Muistin luku ja kirjoitus.

Palautus	1. tokeni	Parametrit	Huom.
KYLLÄ	mr	EXPRESSION	
EI	mw	EXPRESSION (ptr), EXPRESSION (value)	Kirjoittaa toisen ilmaisun arvon ensimmäisen osoittamaan paikkaan.

Konsolille kirjoittaminen

Palautus	1. tokeni	Parametrit	Huom.
EI	print	EXPRESSION	Kutsuu std::cout. Parametri tulkitaan osoittimen kahdeksantavuisiin merkkeihin.
EI	printnumber	EXPRESSION	Kutsuu std::cout.