OAMK
OULUN AMMATTIKORKEAKOULU

Jiska Parrila

VIRTUAALITIETOKONEEN JA ASSEMBLY-KIELEN KEHITYS

VIRTUAALITIETOKONEEN JA ASSEMBLY-KIELEN KEHITYS

Jiska Parrila
Opinnäytetyö 15 op
Kevät 2018
Tietotekniikan tutkinto-ohjelma
Oulun ammattikorkeakoulu

TIIVISTELMÄ

Oulun ammattikorkeakoulu
Tietotekniikan tutkinto-ohjelma, ohjelmistokehitys

Tekijä: Jiska Parrila
Opinnäytetyön nimi: Virtuaalitietokoneen ja assembly-kielen kehitys
Työn ohjaaja: Kari Laitinen
Työn valmistumislukukausi ja -vuosi: Kevät 2018
Sivumäärä: 30 + 2 liitettä

Työ käsittelee tietokoneen toimintaa matalalla tasolla virtuaalitietokoneen avulla. Virtuaalitietokoneelle on toteutettu käskykanta, assembly-kieli ja assembly-kääntäjä eli assembleri. Kaikki kolme rakennettiin Kari Laitisen C++:lla kirjoittamia IC8- ja IML-ohjelmia muokkaamalla.

Työn alkuperäinen tavoite oli kirjoittaa moniajoon kykenevä käyttöjärjestelmä IC8:lle. Johtuen kuitenkin IC8:n ja IML:n rajoituksista työ painottui niiden uudelleen kirjoittamiseen, joka johti lopulliseen aiheeseen. Muutoksia ohjelmiin tehtiin tarpeen vaatiessa. Työn lopputuloksena saatiin aikaan IC8:aa ominaisuuksiltaan rikkaampi IC8E, siistitty ja paranneltu IML-assembleri nimeltään IMLE ja joustavampi assembly-kieli.

Työn tulosten ansiosta IC8E:lle olisi mahdollista kirjoittaa alkuperäisen aiheen mukainen käyttöjärjestelmä. IMLE-kääntäjää täytyy kuitenkin kehittää hieman, jotta se kykenisi kääntämään uudelleensijoitettavia ohjelmia.

Asiasanat: virtuaalitietokone, assembly, assembleri, assembly-kieli, ohjelmointikieli, kääntäjä

ABSTRACT

Oulu University of Applied Sciences
Information technology, software

Author: Jiska Parrila
Title of thesis: Development of virtual machine and assembly language
Supervisor: Kari Laitinen
Term and year when the thesis was submitted: Spring 2018
Pages: 30 + 2 appendices

This thesis focuses on how computers operate on a low level and illustrates that behavior with the help of a virtual machine. An instruction set, assembly language and assembler were created alongside the virtual machine. All three are based on Kari Latitinen’s IC8 and IML programs, written in C++.

Originally the goal of this thesis was to create a small scale operating system for the IC8 virtual machine which would have been capable of multitasking. However, the original IC8 and IML were deemed to be too limited for an operating system implementation. This led to most of the work to go into rewriting the IC8 and IML which became the final subject of this thesis. Changes were made to IC8 and IML when needed. These changes resulted in a more feature rich version of IC8 called IC8E, a cleaned up and improved version of IML-assembler called IMLE and a more flexible assembly language.

Due to the results of this thesis, it would now be possible to write a functional operating system for IC8E using the updated IMLE-assembly. Some additional changes to IMLE are required for it to be able to compile relocatable programs.

Keywords: virtual machine, assembly, assembler, assembly language, programming language, compiler

SISÄLLYS

TIIVISTELMÄ 3
ABSTRACT 4
SISÄLLYS 5
1 JOHDANTO 6
2 IC8- JA IC8E-VIRTUAALITIETOKONEET 7
2.1 Rekisterit 8
2.2 IC8E:n rekisterit 9
2.3 Muistiavaruus 10
2.4 Käskykanta 14
2.5 Keskeytykset 16
2.6 I/O-laitteet 18
3 ASSEMBLY-KIELI 20
3.1 IMLE-assembly 20
3.2 Suunnittelu 22
3.3 Muuttujat 23
4 KÄÄNTÄJÄ 24
4.1 IMLE-kääntäjä 24
4.2 Tokenisointi 25
4.3 Parsinta 25
4.4 Semanttinen analysointi 26
4.5 Optimointi ja koodin generointi 26
5 PROJEKTIN JATKAMINEN 28
6 YHTEENVETO 29
LÄHTEET 30
LIITTEET
Liite 1. IC8E-käskykanta
Liite 2. IMLE-assemblyn kielioppi

1 JOHDANTO

Tämä opinnäytetyö toteutettiin oman osaamisen kehittämiseksi. Työn ohjaajana toimi lehtori Kari Laitinen, jonka kanssa myös aihe saatiin päätettyä. Henkilökohtaisena tavoitteena oli valita aihe, joka olisi haastava ja mielellään matalan tason ohjelmointia. Pohjaksi tälle työlle valikoituivat Laitisen C++:lla kirjoittamat IC8-virtuaalitietokone, sen IML-kääntäjä ja -assembly, jotka on kaikki kuvattu tarkemmin hänen kirjoittamassaan Java-ohjelmointikirjassa (1).

Työn tarkka tavoite oli pitkään auki ja eli paljon työstön aikana. Alun perin tavoitteena oli muokata virtuaalikone moniajokelpoiseksi ja kirjoittaa sille pieni käyttöjärjestelmä ohjelmien ajoa hallinnoimaan. Kääntäjään tai koneen käskykantaan ei ollut tarkoitus tehdä suuria muutoksia. Alkuperäisen koodin rakenteen vuoksi sen laajennuskelpoiseksi muokkaamisessa kesti odotettua pidempään. Tämä päti niin IC8:aan kuin IML:ään. Näistä syistä työn aihe siirtyi pitkälti olemassa olevan järjestelmän laajentamisesta lähes täyteen uudelleenkirjoitukseen. Uudelleenkirjoittamisen ohella toki lisättiin uusia ominaisuuksia ja muokattiin olemassa olevia, mutta niiden osuus työssä on huomattavasti pienempi, kuin mitä ensin kaavailtiin.

IC8:n ympärillä on hallintaohjelma, joka koki myös muutoksia, mutta on toissijaisessa asemassa. Hallintaohjelmalla pystytään muokkaamaan IC8:n muistia, lataamaan siihen ohjelmia ja purkamaan niitä takaisin komennoiksi. Jotkin toiminnoista rikkoontuivat laajojen koodimuutosten takia.

2 IC8- JA IC8E-VIRTUAALITIETOKONEET

Imaginary computer eli lyhyesti IC8 on Kari Laitisen kirjoittama 8-bittinen virtuaalitietokone, joka toteuttaa oman käskykantansa. Myös koneen muistiavaruus on 8-bittinen. Tätä ohjelmaa käytettiin tämän opinnäytetyön pohjana, josta syntyi IC8E (imaginary computer enhanced). Ohjelma koki suuria muutoksia koodillisesti ja ominaisuuksiltaan. Suurimmat muutokset ovat muistiavaruuden laajentaminen 16-bittiseksi, keskeytysten lisääminen ja käskykannan vaihto. Käskyt säilyivät 8-bittisinä. Koneen muisti on järjestelty little endianin mukaisesti, jolloin useatavuisen arvon vähiten merkitsevä tavu on muistissa ensimmäisenä.

Pienemmän mittakaavan, mutta sitäkin tärkeämpi, muutos IC8E:ssä oli sen prosessorin odotuslogiikan muuttaminen odottavasta kyseleväksi. Odotuslogiikkaa tarvitaan prosessorin kellotaajuuden asettamiseen, joka ajoittaa kaiken mitä prosessori tekee ja korvaa oikeassa prosessorissa käytettävän värähtelijäkristallin, josta käyntitaajuus muuten otettaisiin. Joka kerta kun prosessorin kello ”värähtää”, uuden prosessorisyklin suorittaminen aloitetaan. Prosessorin odotuslogiikan muuttaminen mahdollisti jo silloin muihin ohjelman osiin toteutetun vaiheistetun suoritusrakenteen.

Koko ohjelma ja kaikki sen osat oli nyt jaettu omiksi tilakoneen tiloiksi, joiden välillä liikuttiin käyttäjäsyötteen perusteella. Kulloinkin aktiiviselle tilalle kutsutaan päivitys-, syötteen luku- ja piirto-funktiota listatussa järjestyksessä. Tämä mahdollisti IC8E:n muuntamisen odottavasta ohjelmasta simulaatioksi, joka päivittää itseään jatkuvasti.

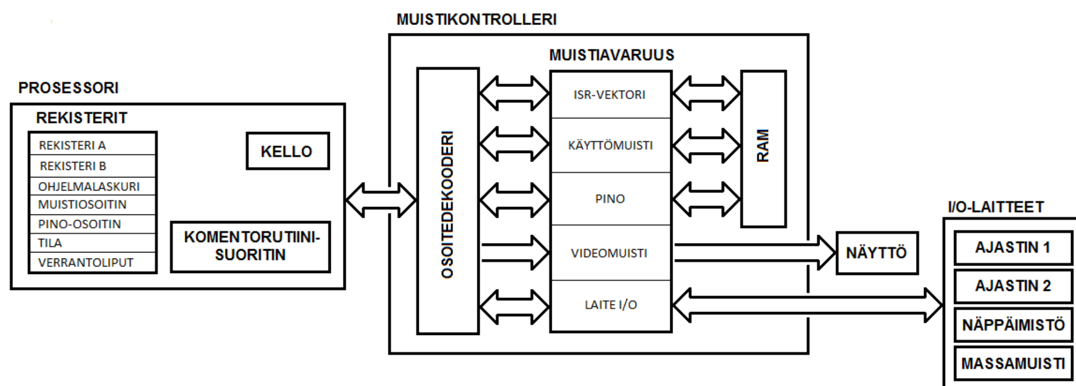
Taustatiedon kerääminen oli välttämätöntä työn jatkamiseksi. Suunnittelun ja tutkinnan edetessä sukellettiin hyvin syvälle tietokoneiden toimintaan. Prosessorin rekisterit, komentojen suorittaminen, keskeytykset ja laitteiden kanssa kommunikointi olivat keskeisiä aiheita. Erityisesti keskeytyksiin perehtyminen oli tarkkaa ja niiden toiminta tuli ehkä liiankin selväksi. Rekistereistä selvitettiin prosessoreille yleisimmät ja niiden käyttötarkoitus.

Komentojen suorittamiseen saatiin viitteitä jo koodin siistinnän aikana IC8:n komentorutiinista. Komentojen suorittamisesta haettiin tietoa myös muualta, jotta

optimaalinen ratkaisu saataisiin toteutettua (2). Lyhyehkön tutkinnon jälkeen elegantimmat ratkaisut komentojen suorittamiseen todettiin turhan kehittyneiksi ja niiden toteuttaminen IC8E:ssä olisi ollut yliampuvaa. Näinpä IC8:n tarjoama ratkaisu ei kokenut juurikaan muutoksia.

Laitteiden ja prosessorin välisen viestinnän tutkiminen vei suurimman osan tutkimusajasta. Korkean tason kuvan muodostamista varten täytyi tutkia useita aiheeseen liittyviä lähteitä, joista jotkin ovat hieman vanhentuneita, mutta sisällöltään edelleen voimassa olevia. (3; 4; 5; 6.)

Kuvassa 1 on kuvattu IC8E:n rakenne ja järjestelmän eri osien välinen kommunikointi.



KUVA 1: IC8E:n lohkokaavio

2.1 Rekisterit

Prosessorirekisteri on erittäin nopea ja pieni muistialue, joka sijaitsee useimmiten prosessorin sisällä. Jotkin rekisterit ovat vapaasti käytettävissä, kun taas jotkin ovat prosessorin itse hallinnoimia. Rekistereitä käytetään prosessorin tilan säilömiseen, matemaattisiin ja loogisiin operaatioihin sekä muistissa olevan datan manipuloimiseen. Rekisterien bittikoko on yksi, mutta ei ainut tekijä, joka vaikuttaa prosessorin bittisyyteen (8-, 16-, 32- tai 64-bittiset ovat yleisimpiä).

2.2 IC8E:n rekisterit

IC8E:n prosessorilla on kymmenkunta eri rekisteriä. Luku vaihtelee hieman riippuen siitä, mikä luetaan rekisteriksi. Kaikkien näiden sisältämiä arvoja voi muokata tavalla tai toisella. 8-bittisiä käyttörekistereitä on kaksi: A ja B. Näihin rekistereihin voi ladata arvoja muistista ja niiden sisältämät arvot voidaan kirjoittaa takaisin muistiin. Yleisiä käyttötapoja näille rekistereille ovat matemaattiset, vertailu- ja muunnosoperaatiot sekä koneen ulkoisten arvojen väliaikainen säilöminen, ennen kuin ne kirjoitetaan muistiin.

Program counter (PC) eli ohjelmalaskuri on 16-bittinen rekisteri, joka sisältää seuraavan suoritettavan komennon muistiosoitteen. IC8E:n käynnistyessä tämä rekisteri asetetaan heksadesimaaliarvoon 0x0100, joka on ensimmäinen ohjelmakoodille varattu muistiosoite. Jokaisen prosessorisyklin aikana prosessori lukee komennon tämän rekisterin osoittamasta muistiosoitteesta ja nostaa sen arvoa yhdellä. Riippuen suoritettavasta komennosta on mahdollista, että PC:n arvoa muokataan enemmänkin yhden prosessorisyklin aikana. Yksi tällainen komento on hyppy komento jump, joka asettaa PC:n arvon suoraan.

Memory pointer (MP) eli muistiosoitin on 16-bittinen rekisteri, jota käytetään yhden muistiosoitteen säilömiseen. IC8:n vastaava rekisteri toimii samalla tavalla, mutta on 8-bittinen. Tämän rekisterin käyttötarkoitukset ovat monipuolisia ja se mahdollistaa yllättävän monimutkaisten ohjelmien toteuttamisen. Sitä voidaan muokata sekä verrata ja sille voidaan suorittaa matemaattisia muutoksia A- ja B-rekisterien tavoin. Näiden lisäksi prosessorissa on komentoja, joilla on mahdollista lukea ja kirjoittaa muistia muistiosoittimen sisältämästä osoitteesta.

Stack pointer (SP) eli pino-osoitin on 8-bittinen rekisteri, joka sisältää pinon korkeuden. Pino on LIFO-tyyppinen (last in, first out) muistialue, johon talletetaan arvoja aina tarpeen vaatiessa. Pinoa käytetään yleisesti keskeytyksiä ja aliohjelmakutsuja suoritettaessa prosessorirekisterien sisältämien senhetkisten arvojen tallettamiseen. Kun aliohjelma tai keskeytys on suoritettu, arvot palautetaan pinosta lukemalla. Pino-osoittimen arvoa kasvatetaan joka kerta, kun pinoon talletetaan arvo ja vastaavasti lasketaan joka kerta, kun pinosta luetaan arvo. Käytön

helpottamiseksi IC8E:ssä kaikki arvot talletetaan pinoon 16-bittisinä ja muutetaan tarvittaessa 8-bittisiksi niitä sieltä luettaessa.

Interrupt priority (keskeytysprioriteetti) on yksi uusista uudelleenkirjoituksen yhteydessä lisätyistä rekistereistä. Rekisteri on 8-bittinen ja pitää sisällään mahdollisesti käynnissä olevan keskeytysrutiinin prioriteetin. Sitä ei pystytä muokkaamaan suoraan, vaan sen arvo määräytyy aina keskeytyksen mukaan. Tätä arvoa käytetään selvittämään, mikäli jokin keskeytys sallitaan, kun toinen keskeytysrutiini on jo käynnissä.

Toinen uusi rekisteri on vertailuoperaation tuloksen säilömiseen lisätty 8-bittinen verrantorekisteri. Rekisteri sisältää tietoa numeroiden välisestä suuruusvertailusta ja sitä käytetään ehdollisia hyppykomentoja suoritettaessa. Rekisterin arvoa laskettaessa tarkistetaan vain, onko ensimmäinen luku yhtä suuri, suurempi tai pienempi kuin toinen luku. Nämä kolme tarkistusta riittävät ja niistä voidaan juontaa loput yleisimpien verranto-operaatioiden tulokset.

Mainittujen rekisterien lisäksi prosessori sisältää myös suoritettujen komentosyklien määrän ja aktiivisuustilaa indikoivan arvon. Komentosyklien määrälle ei tällä hetkellä ole järkevää käyttötarkoitusta tai komentoja, joilla sitä voitaisiin hyödyntää.

Aktiivisuustila puolestaan kertoo, onko prosessori aktiivinen, horroksessa vai kokonaan pois päältä. Jokaiseen tilaan siirtymiseen on olemassa komento, mutta niiden välillä liikkuminen täysin vapaasti ei onnistu. Esimerkiksi jos prosessorin käsketään siirtyä horrokseen, siitä toiseen tilaan siirtyminen vaatii keskeytyksen suorittamista. Aktiivisena ollessaan prosessori suorittaa komentoja normaalista. Horroksessa prosessori ei suorita komentoja, mutta kuuntelee edelleen keskeytyksiä. Keskeytyksen sattuessa prosessori siirtyy aktiiviseen tilaan. Pois päältä ollessaan prosessori ei tee mitään.

2.3 Muistiavaruus

Muistiavaruudella tarkoitetaan osoitteita, joita prosessori voi käyttää. Muistiavaruutta ei tule sekoittaa tietokoneen käyttömuistiin eli RAMiin. RAM kyllä löytyy

muistiavaruudesta, mutta ei täytä sitä kokonaan. Prosessorin muistiavaruus jaetaan kaikkien tietokoneeseen liitettyjen laitteiden kesken. Jokainen laite saa sille varatun alueen muistiavaruudesta, jota prosessori käyttää laitteiden kanssa kommunikointiin. Jos jokin laite tai tietokoneen komponentti varaisi koko muistiavaruuden itselleen, koneesta ei olisi juurikaan hyötyä.

Monilla voi olla kokemusta 32-bittisistä tietokoneista ja niiden käyttömuistin määrän rajoittamisesta hieman alle 4 Gt:uun. Syy tähän on juurikin muistiavaruus, koska 32-bittinen prosessori pystyy osoittamaan noin 4,29 miljardiin eri muistiosoitteeseen, joka gigatavuiksi muutettuna on tasan 4. Jos koko 32-bittinen muistiavaruus täytettäisiin muistilla, tietokoneeseen ei pystyisi liittämään hiirtä, näppäimistöä, näyttöä tai mitään muutakaan laitetta. Monien onneksi 64-bittiset tietokoneet ovat yleistyneet ja niiden muistiavaruus kattaa 18,4 kvadriljoonaa muistiosoitetta, joka on 17,1 miljardia gigatavua tai tasan 16 exatavua.

IC8:n muistiavaruus on 8-bittinen ja pystyy osoittamaan 256:een eri muistiosoitteeseen. IC8E:ssä se laajennettiin 16-bittiseksi, jolloin käytettävissä olevien muistiosoitteiden määrä nousi 65 536:een. Laajennus itsessään ei tuottanut paljoakaan työtä, mutta sen käyttöönotto vaati joidenkin komentojen ja rekisterien muuttamista toimimaan 16-bittisillä arvoilla 8-bittisten sijaan. Tämä muutos heijastui myös IMLE-kääntäjään ja tuotti taas lisätyötä. Pelkkä muistiavaruuden laajentaminen koettiin kuitenkin riittämättömäksi. Syötteen ja tulosteen (I/O) käsittely muistikartoitusta käyttäen oli ollut pöydällä jo jonkin aikaa. Muistikartoitetun I/O:n tarkoituksena on yksinkertaistaa prosessoriarkkitehtuuria vähentämällä tarvittavien komentojen määrää. Sen sijaan että käytettäisiin tarkoituskohtaisia komentoja I/O:n hoitamiseen, muistikartoitetussa järjestelmässä I/O hoidetaan käyttäen samoja komentoja kuin käyttömuistin käsittelyssä.

Jotta muistikartoitettu käskykanta pystyy käyttämään kaikkia laitteita, tarvitaan osoitedekooderi. Osoitedekooderi on prosessorin kanssa kommunikoiva mikropiiri, jonka läpi kaikki laitteiden, muistin ja prosessorin välinen data kulkee. Se hoitaa prosessorista muistikartoituksen seurauksena poistuneet toiminnot ja selvittää, mille laitteelle prosessorilta saatu data tulee lähettää.

IC8:n käskykanta hoitaa I/O:n käsittelyn tarkoitukseen varatuilla komennoilla, joiden käyttötavat ovat jokseenkin rajoittuneet. Tästä johtuen IC8E:n I/O-toiminnot muutettiin käyttämään muistikartoitusta. Päätös johti IC8E:n muistiavaruuden osittamiseen, koska I/O vaatisi nyt oman alueensa muistiavaruudesta, jolloin koko muistiavaruutta ei voida enää varata käyttömuistille.

Mallia muistin osittamiseen otettiin vanhoista 8-bittisistä järjestelmistä, joista 90-luvulla valmistetun käsikonsoli Game Boyn muistiosiointi (7) valikoitui pohjaksi. Muutamien muutosten jälkeen saatiin aikaan tarkoitukseen sopiva muistiavaruus, joka mahdollisti muistikartoitetun I/O:n. Taulukko 1 listaa toteutetun muistiavaruuden osiot ja niiden sijainnin siinä. Kaikista osioista löytyy tarkempaa tietoa tämän dokumentin muista luvuista.

IC8E säilyttää koko muistiavaruutensa yhdessä taulukossa. Tämä ei vastaa todellista tietokonetta, jossa muistiavaruuden ja itse käyttömuistin välissä olisi osoitedekooderi, joka selvittää, mille koneeseen kytketylle laitteelle osoite kuuluu. Jos osoite sattuisi osumaan käyttömuistille varatulle alueelle, osoitedekooderi lähettäisi arvon muistille. Jos taas osoite osuu videomuistille varatulle alueelle, arvo lähetettäisiin näytönohjaimelle tai hyvin primitiivisissä tietokoneissa suoraan näytölle. Dekooderi on IC8E:ssä korvattu antamalla siihen liitetyille laitteille muistiavaruustaulukkoon osoittava osoitin, jota laitteet käyttävät prosessorin kanssa kommunikoidessaan. Lopputulos on sama kuin osoitedekooderia käytettäessä, mutta koodia ja arvojen kopiointia saadaan vähennettyä tällä ratkaisulla

TAULUKKO 1: Muistiavaruuden osiot

Käyttötarkoitus	**Alkuosoite**	**Loppuosoite**	**Koko tavuina**
ISR vektori	0x0000	0x00FF	256
Käyttömuisti	0x0100	0xF4FF	62464
Pino	0xF500	0xF6FF	512
Videomuisti	0xF700	0xFEFF	2048
Laite I/O	0xFF00	0xFFFF	256
Yhteensä	0x0000	0xFFFF	65536

TAULUKKO 2: Laite-I/O -osion sisäinen jaottelu

Käyttötarkoitus	Alkuosoite	Loppuosoite	Koko tavuina
Teksti monitori	0xFF00	0xFF04	5
Varattu	0xFF05	0xFF11	13
Näppäimistö	0xFF12	0xFF14	3
Kovalevy	0xFF15	0xFF1A	6
Varattu	0xFF1B	0xFF1F	5
Ajastin 1	0xFF20	0xFF22	3
Varattu	0xFF23	0xFF23	1
Ajastin 2	0xFF24	0xFF26	3
Varattu	0xFF27	0xFFFF	217
Yhteensä	0xFF00	0xFFFF	256

2.4 Käskykanta

Käskykanta tai komentokanta on kokoelma komentoja, jotka prosessori pystyy suorittamaan. Käskykanta on keskeinen osa mitä tahansa prosessoria ja voitaisiin ymmärtää sen kieleksi. Yksinkertaisimmillaan käskykannan komennot suorittavat hyvin selkeitä ja rajattuja toimintoja, kuten arvon lukeminen muistista tai jokin matemaattinen operaatio (+, -, /, *, ...). Komennot voivat olla myös hyvin monimutkaisia ja koostua useasta yksittäisestä toiminnosta. Käskykannan komentojen luonteen pohjalta se voidaan luokitella joko RISC- (reduced instruction set computing) tai CISC-tyyppiseksi (complex instruction set computing). Aiheesta

voi lukea tarkemmin englanninkielisestä Instruction set architecture -Wikipedia-artikkelista (8).

RISCin idea on pitää kaikki käskykannan komennot yksinkertaisina ja monikäyttöisinä, siinä missä CISC voi toteuttaa hyvinkin monimutkaisia ja erikoistuneita komentoja. Molemmilla tyypeillä on niille ominaisia piirteitä, jotka vaikuttavat pitkälti siihen, miten ja missä niitä käytetään. RISC on usein energiatehokas, joten se soveltuu hyvin mobiililaitteisiin. CISC sopii paremmin raskaaseen laskentaan ja sitä käytetäänkin paljon pöytäkoneissa ja palvelimissa. Mikään ei kuitenkaan estä RISC-pohjaista prosessoria olemasta energiasyöppö tai CISCiä toteuttavaa prosessoria olemasta hidas. Prosessorin ja käskykannan luonteeseen vaikuttaa moni asia, eikä RISC/CISC jaottelu ole useimmiten riittävä peruste valinnalle (8).

Yleinen esimerkki RISC-käskykannan toteutuksesta on ARM, jota käytetään lähes kaikissa kuluttajille tarkoitetuissa mobiililaitteissa. Vastaavasti laajalle levinnyt CISC-toteutus on x86, jota käytetään lähes kaikissa pöytä- ja kannettavissa tietokoneissa.

IC8E käyttää itsekehiteltyä RISCin tapaista käskykantaa. IC8:n alkuperäinen IML-käskykanta sisälsi 54 komentoa, joista moni oli suoraan sidottu johonkin tiettyyn rekisteriin. Lisäksi komento saattoi saada vain yhden operaattorin, jos sen vähäarvoisin bitti oli yksi. Uudelleenkirjoituksen aikana käskykantaa karsittiin ja komentoja muutettiin joustavammiksi. Kaikki rekisterikohtaiset komennot muutettiin siten, että ne saavat käyttämänsä rekisterin operaattorina ja mahdollisen toisen operaattorin, jota käytetään komennon suorittamisessa. Komentojen nimet muutettiin lyhyemmiksi ja päivittyneeseen toimintaansa sopiviksi. Moni olemassa oleva komento todettiin näiden laajennusten tuoman joustavuuden seurauksena turhiksi ja poistettiin kokonaan.

Näillä muutoksilla IML:stä saatiin kehitettyä IMLE-käskykanta, jonka IC8E toteuttaa. IMLE:n ymmärtämiä komentoja on 34, jotka on dokumentoitu tarkemmin liitteessä 1. Laskeneesta komentomäärästä huolimatta koneella on edelleen samat toiminnallisuudet ja se pystyy itseasiassa tekemään enemmän, kuin mitä vanhalla käskykannalla oli mahdollista saavuttaa.

Muutoksen mahdollistamiseksi IC8E:hen IC8:sta kopioitu komentoja suorittava koodi täytyi päivittää. Operaattorin lukeminen siirrettiin komennon suorittamisen yhteyteen, mikä toi joustavuutta. Kaikki komennot ovat edelleen 8-bittisiä. Koska komennot saattavat joutua käsittelemään 16-bittisiä arvoja, operaattoreita luettaessa luetaan aina kaksi tavua. Poikkeuksena tähän on tilanne, jossa operaattori voi olla ainoastaan rekisteri, jolloin luetaan vain yksi tavu.

Käskykannan päivittämisen aikana tiettyyn tarkoitukseen erikoistuneita komentoja pyrittiin välttämään. Tässä onnistuttiin hyvin, vaikka joitakin tarkoituskohtaisia komentoja jouduttiinkin kirjoittamaan. Arvojen siirtely muistin ja rekisterien välillä vaati muutaman eri version samasta komennosta. Siirtokomennot eroavat lähinnä arvojen käytön suhteen ja siinä, minne ne lopulta sijoittavat käsiteltävän arvon. Siirtokomentojen lisäksi keskeytykset vaativat omia komentojaan. Jälkikäteen ajateltuna keskeytysvektorin asetuskomento (load_isr) olisi voitu korvata tavallisella siirtokomennolla. Myös keskeytysrutiinista palaamista varten tarvittiin oma komento, koska keskeytyskutsu eroaa hieman tavallisesta aliohjelmakutsusta asynkronisuutensa vuoksi ja vaatii siksi muutaman lisätoiminnon suorittamista tavalliseen paluukomentoon verrattuna.

2.5 Keskeytykset

Keskeytykset ovat prosessoriin sen ulkopuolelta tulevia signaaleja. Nimensä mukaan ne keskeyttävät prosessorin sen hetkisen toiminnan, jotta keskeytystä varten kirjoitettu aliohjelma voidaan suorittaa. Tällainen aliohjelma tunnetaan nimellä keskeytysrutiini tai ISR (interrupt service routine). Keskeytykset mahdollistavat asynkronisten toimintojen suorittamisen. Niitä käytetään moniin eri tarkoituksiin, kuten datan vastaanottamiseen verkon yli, näppäimistön painallusten havaitsemiseen ja tiedostojen lukemiseen massamuistista. Lähestulkoon kaikki tietokoneen toiminnot, jotka voivat tapahtua millä tahansa ajanhetkellä tai joihin prosessorilla ei ole täyttä kontrollia, toteutetaan keskeytyksillä.

Keskeytykset voidaan jakaa kahteen pääkategoriaan: laitteisto- ja ohjelmistolähtöisiin. Laitteistolähtöisten keskeytysten lukumäärä on rajallinen ja määräytyy käytetyn prosessorin mukaan. Ohjelmistolähtöisiä keskeytyksiä voi olla huomattavasti enemmän ja niiden ainoat rajoitukset ovat prosessoriteho ja käyttömuistin

määrä. Jotta prosessori tietää, mikä aliohjelma sen tulee suorittaa mitäkin keskeytystä kohden, se yrittää etsiä sopivaa keskeytysrutiinia keskeytystaulusta (interrupt vector table). Keskeytystaulu on lyhyehkö muistialue, jossa säilytetään kunkin keskeytysrutiinin ensimmäistä muistiosoitetta. Jokaiselle laitteistolähtöiselle keskeytykselle on varattu paikka keskeytystaulussa, joka löytyy keskeytyskoodin perusteella.

Ohjelmistolähtöiset keskeytykset toimivat samaan tapaan, mutta ne eivät ole prosessorin hallitsemia, vaan tapahtuvat ohjelmatasolla, yleensä käyttöjärjestelmän sisällä, joka toimii niiden alustana ja hallinnoi niitä. Niissä keskeytystaulu on korvattu muistissa olevalla rakenteella, joka ajaa täysin saman asian, kuin prosessorin keskeytystaulu. Itse keskeytyksiä simuloidaan käyttöjärjestelmän ohjaamilla viesteillä.

IC8E:n keskeytystaulu varaa ensimmäiset 256 tavua sen muistiavaruudesta, joka jaetaan kaksitavuisiksi paloiksi, 16-bittisten muistiosoitteiden tallettamista varten. IC8E pystyy siis käsittelemään 128 eri keskeytystä. Prosessorin saamaa keskeytyskoodia käytetään sellaisenaan muistiosoitteena, kun keskeytystaulusta etsitään keskeytysrutiinia. Jos keskeytystaulusta löytyvä muistiosoite on käyttökelpoinen, prosessorin sen hetkinen tila talletetaan pinoon ja ohjelmalaskuri asetetaan osoittamaan keskeytysrutiinin muistiosoitteeseen.

IC8E toteuttaa prioriteetti järjestelmän keskeytyksiä varten, joka mahdollistaa joidenkin keskeytysten suorittamisen, vaikka jokin toinen keskeytysrutiini olisi jo suoritettavana. Saatua keskeytystä aletaan suorittaa, vain jos sillä on korkeampi prioriteetti kuin nykyisellä. Saman prioriteetin keskeytysten täytyy odottaa. Odottaminen on välttämätöntä, jotta prosessori ei jäisi keskeytyspyörteeseen jossa se ei voi tehdä muuta, kuin aloittaa uusia keskeytys rutiineja. Mikäli keskeytyspyörre pääsee syntymään se johtaa pinon ylivuotoon (stack overflow).

I/O toteutetaan IC8E:ssä keskeytyksiä käyttäen. Merkin lukeminen näppäimistöltä sekä datan lukeminen ja kirjoittaminen massamuistiin hoidetaan molemmat keskeytyksillä. Lisäksi IC8E:ssä on kaksi keskeytyksillä toimivaa ajastinta, jotka voidaan ohjelmoida erillään toisistaan.

2.6 I/O-laitteet

I/O on lyhenne sanoista input/output, syöte ja tuloste. Suurin osa tietokoneiden toiminnasta on mahdollista näinkin yksinkertaisten toimintojen ansiosta. I/O toimintoja suoritetaan isossa ja pienessä mittakaavassa: kuvan ilmestyminen ruudulle, yhden tavun lukeminen muistista prosessorirekisteriin, tiedoston lähettäminen verkon yli toiseen tietokoneeseen. I/O:n voidaan luonnehtia olevan datan liikettä järjestelmän eri osien välillä, jota prosessori orkestroi.

I/O laitteilla tarkoitetaan tietokoneeseen liitettäviä laitteita tai kaapeleita, jotka mahdollistavat prosessorin ja jonkin järjestelmän ulkopuolisen laitteen välisen kommunikoinnin. Prosessorin ja laitteen kommunikointi voi olla yksi- tai kaksisuuntaista. Yksisuuntaisessa kommunikoinnissa joko prosessori tai laite lähettää dataa, mutta ei saa takaisin mitään vastausta. Esimerkkejä tällaisista laitteista ovat 3,5 mm:n liitintä käyttävät kuulokkeet ja mikrofoni.

Moni tietokoneeseen kytketty laite käyttäytyy yksisuuntaisen laitteen tavoin, mutta todellisuudessa ei voisi toimia, jos ei saisi vastausta tietokoneelta. Tällaisia laitteita ovat näppäimistö, hiiri ja näyttö.

Kuten arvata saattaa, kaksisuuntaiset laitteet lähettävät ja vastaanottavat dataa kytkettynä ollessaan. Massa- ja USB-muistit, modeemit, toiset tietokoneet ja muut hiemankaan monimutkaisemmat laitteet käyttävät kaksisuuntaista I/O:a.

IC8E:ssä on kolme I/O-laitetta: näppäimistö, tekstinäyttö ja massamuisti. Taulukko 2:n pohjalta voidaan todeta, että näppäimistölle on varattu muistiavaruudesta osoitealue väliltä 0xFF12 - 0xFF14. Tätä muistialuetta käytetään näppäimistön kanssa kommunikointiin. Ensimmäinen tavu osoitteessa 0xFF12 kertoo näppäimistön tilan. Siitä voidaan tarkistaa, onko jotain nappia painettu. Näppäimistön toinen tavu osoitteessa 0xFF13 sisältää painetun näppäimen näppäinkoodin. Prosessori voi siis lukea näppäimen painalluksia lukemalla tässä muistiosoitteessa olevan arvon.

Näytölle varattu muistialue kattaa osoitteet 0xFF00 - 0xFF04. Tätä muistialuetta ei tällä hetkellä käytetä, mutta mahdollisia toimintoja sille voisi olla esimerkiksi näytön pyyhkiminen.

Video muisti, joka taulukko 2:n mukaan kattaa muistialueen väliltä 0xF700 - 0xFEFF, toimii tiivisti näytön kanssa. IC8E:n tapauksessa johonkin tällä muistialueella olevaan osoitteeseen kirjoittaminen saa arvon ilmestymään näytölle. Näyttö jakaa saman muistin prosessorin muistikontrollerin ja itsensä kanssa, jolloin arvojen kopioinnilta vältytään. Kyseisen muistialueen sisällön visualisointitapa riippuu näytöstä.

Nykyinen näyttö on tekstinäyttö, joten se piirtää muistialueen sisällön ASCII-merkkeinä. Merkkien sijainti näytöllä on suoraan sidonnainen sen muistiosoitteeseen. Osoite 0xF700 on näytön vasen yläkulma. Seuraavan osoitteen merkki tulostuu edellisen oikealle puolelle, kunnes näytöstä loppuu leveys ja uusi rivi alkaa. Olettaen, että näyttö on riittävän suuri, osoitteeseen 0xFEFF kirjoitettu merkki tulostuisi oikeaan alakulmaan.

Olisi myös mahdollista kirjoittaa toisenlainen näyttö, joka tulkitsisi videomuistin sisältämiä arvoja eri tavalla. Esimerkkinä jokainen muistiosoite voisi sisältää 8-bittisen väriarvon, joka tulostetaan samaa sijaintilogiikkaa käyttämällä. Tällöin saataisiin värinäyttö, jonka jokaisen pikselin sävyä pystytään kontrolloimaan kirjoittamalla videomuistiin.

Massamuistin kanssa kommunikointiin on varattu osoitteet 0xFF15 - 0xFF1A. Ensimmäiseen tavuun kirjoittamalla prosessori voi aloittaa luku- tai kirjoitustoiminnon tai keskeyttää käynnissä olevan toiminnon. Massamuisti käyttää samaa tavua oman tilansa ilmoittamiseen. Seuraava tavu kertoo käsiteltävän datan pituuden. Massamuisti pystyy kirjoittamaan tai lukemaan yhden toiminnon aikana korkeintaan 256 tavua. Seuraavaan kahteen tavuun määritellään muistiosoite, jota massamuisti käyttää. Dataa kirjoitetaan osoitteeseen levyltä luettaessa ja siitä luetaan dataa levylle kirjoitettaessa. Näitä seuraavat tavut kertovat massamuistin sisäisen osoitteen, joka toimii samaan tapaan kuin muistiosoite.

3 ASSEMBLY-KIELI

Assembly on prosessoriarkkitehtuurikohtaisesti toteutettu ohjelmointikieli. Se on alimman tason ohjelmointikieli, joka on tarkoitettu ”ihmisten” kirjoitettavaksi. Assemblysta alempi taso on kaikkein primitiivisin tietokonekieli eli raa’at konekomennot, joita prosessori ymmärtää ilman mitään muutoksia. Assembly-komennot kartoittuvat konekomentoihin melkein yksi yhteen. Assembly-komennot ovat muutaman merkin mittaisia lyhenteitä sanoista, joista jokainen kääntyy vain yhdeksi tai kouralliseksi konekomentoja.

Assemblyä kirjoitetaan hyvin harvoin, koska se on erittäin työlästä ja virhealtista. Sen osaamisesta tai ymmärtämisestä on tosin hyötyä ohjelmien takaisinmallintamisessa (reverse engineering), tietoteknisenä yleissivistyksenä tai jos jokin järjestelmä vaatii erittäin kovatasoista optimointia. Optimointikin on nykyään kyseenalainen syy assembly-kielen käyttämiseen, koska C-kääntäjät kykenevät generoimaan tehokkaampia komentorakenteita, kuin mitä ihmiselle on mahdollista. Ohjelmaa, joka kääntää assembly-koodia konekomennoiksi, nimitetään assembleriksi. Assembleri on assembly-kielikohtainen.

3.1 IMLE-assembly

IMLE on IC8E:tä varten kirjoitettu assembleri joka osaa kääntää IMLE-assemblyllä kirjoitetun ohjelman IC8E:n ymmärtämiksi konekomennoiksi. Liitteessä 1 on listattu kaikki IMLE-assemblyn komennot ja niiden konekieliset arvot. Toisin kuin useimmissa assembly-kielissä, IMLE:ssä ei käytetä sanojen lyhenteitä komentoina vaan lyhyitä sanoja. Tämä helpottaa sillä kirjoitettujen ohjelmien ymmärtämistä. Joitakin yleisiä lyhenteitä käytetään, jotta komennot eivät veny pitkiksi.

Kuvat 2 ja 3 sisältävät alkuperäisellä IML:llä ja uudistetulla IMLE-kielellä kirjoitetut versiot samasta ohjelmasta. Kuvista näkee, että IMLE:llä kirjoitettu ohjelma on hieman pitempi. Tämä johtuu tiettyyn tarkoitukseen erikoistuneiden komentojen puuttumisesta, joita IML:ssä esiintyy. Esimerkiksi komento jump_if_register_a_zero on jouduttu korvaamaan kolmella eri komennolla, jotta sama toiminnallisuus voidaan toteuttaa. Tarvittavat komennot ovat move_in, compare ja

jump_ne. Kasvaneesta ohjelmakoosta huolimatta korvaavat komennot ovat huomattavasti alkuperäistä ratkaisua joustavampia, mutta vaativat useamman prosessorisyklin.

```
1
2   //  abcde.iml   (c) 1998-2000 Kari Laitinen
3
4   //  This program prints the letters ABCDE to the screen.
5   //  Register A is used to count how many characters have
6   //  been printed. The character code being output to the
7   //  screen is held in register B. The first code is 'A'
8   //  which means 41H, the ASCII code of letter A.
9   //  As the ASCII code in register B is incremented after
10  //  each printing, a different letter will be printed
11  //  each time.
12
13  beginning_of_program:
14        load_register_a_with_value       5
15        load_register_b_with_value       'A'
16
17  print_next_letter:
18        output_byte_from_register_b
19        increment_register_b
20        decrement_register_a
21        jump_if_register_a_zero          end_of_program
22        jump_to_address                  print_next_letter
23
24  end_of_program:
25        stop_processing
26
27
28
```

KUVA 2: Alkuperäisellä IML-kielellä kirjoitettu abcde-ohjelma (9.)

```
1  /*
2
3      abcde.iml                11.02.2018
4
5      Prints letters ABCDE. This is an updated version of
6      abcde.iml program written in the original IML-language
7
8  */
9
10 //                First arg       Second arg
11 var output        @F700h
12 var letter        'A'
13
14 move_in           M               output
15 move_in           A               5
16
17 print:
18   move_in         B               letter
19   move_m_out      output          B
20   increment       M
21   increment       B
22   decrement       A
23   move_out        letter
24   move_in         B               0
25   compare         A               B
26   jump_ne         print
27
28 stop
```

KUVA 3: Uudistetulla IMLE-kielellä kirjoitettu abcde-ohjelma

3.2 Suunnittelu

IMLE-assemblyn suunnittelu aloitettiin, kun IC8E:hen lisättävät ominaisuudet oli päätetty ja suurimmat koodimuutokset tehty. Ohjelmointikielen suunnittelu on haastava prosessi. Tästä johtuen suunnittelu aloitettiin käyttäen olemassa olevaa IML-käskykantaa pohjana. Komentoja karsittiin ja yhdisteltiin, niiden käyttöä luonnosteltiin ja lopulta alettiin pohtia syntaksia. Syntaksi pidettiin yksinkertaisena ja erikoismerkkien käyttöä vältettiin. Syntaksi myös pyrkii välttämään tarvetta tarkastella jo käsiteltyjä merkkejä tai symboleita, kun päätellään käsiteltävän symbolin merkitystä.

Alustavan syntaksisuunnittelun pohjalta luotiin kielioppi IMLE-assemblylle. Kielioppi vaaditaan, jotta assembleri voidaan kirjoittaa. Koska kyseessä on Assembly-kieli, kielioppi pysyi erittäin yksikertaisena. Kielioppi on kuvattu liitteessä 2. Jokainen IMLE:llä kirjoitettu ohjelma koostuu n -kappaleesta lausekkeita (statement). Lauseke voi olla joko komento (instruction), merkintä (label) tai muuttuja (variable). Kommentit on jätetty pois kielioppikaaviosta, koska niitä voidaan lisätä mihin tahansa kohtaan, kuinka monta kertaa tahansa. IMLE tukee yhden ja monen rivin kommentteja (//, /**/).

3.3 Muuttujat

IMLE tukee kolmentyyppisiä muuttujia: kokonaislukuja, tekstiä ja muistiosoitteita. Muuttuja alkaa aina var-avainsanalla jota seuraa muuttujan nimi. Nimen jälkeen tulee muuttujan arvo. Kun muuttujaan halutaan viitata, vain sen nimeä käytetään. Muuttujia käytetään jonkin arvon tallettamiseen muistiin. Muistista se voidaan lukea prosessori rekisteriin, jossa sitä voidaan tarkastella ja muokata. Rekisterissä oleva arvo voidaan muuttuneena kirjoittaa takaisin muuttujan muistiin, jolloin se korvaa muuttujan aiemman arvon.

Kokonaislukuja ja muistiosoitteita voidaan käyttää desimaaleina tai heksadesimaaleina. Kokonaisluku koostuu pelkästään numeroista. Muistiosoite koostuu myös numeroista, mutta alkaa aina @-merkillä. Jos kokonaisluku tai muistiosoite halutaan ilmaista heksadesimaali muodossa, tulee sen perään lisätä h-kirjain.

Tekstin käsittely IMLE:ssä on erittäin primitiivistä. Tekstillä ei ole lopetusmerkkiä eikä sen pituutta tiedetä. Näistä syistä tekstin käyttöä kannattaa vältellä ja käyttää apumuuttujia silloin, kun tekstiä joudutaan käsittelemään.

4 KÄÄNTÄJÄ

Kääntäjä on ohjelma, joka muuntaa ohjelmakoodin tietokoneen ymmärtämiksi konekomennoiksi. Kääntäjän kirjoittaminen ei ole kovin vaativaa, vaan haaste tulee käännättävästä kielestä. Vuosien saatossa kääntäjiin on vakioitunut seitsemän vaihetta: tokenisointi, parsinta, semanttinen analysointi, välikielen generointi, välikielen optimointi, konekoodin generointi ja konekoodin optimointi. (10.) Jokainen näistä vaiheista toteutetaan itsenäisesti ja jotkin voidaan ajaa useampaan kertaan. Ohjelmakoodin kulkiessa kääntäjän läpi sitä muokataan ja tutkitaan jokaisen vaiheen aikana.

4.1 IMLE-kääntäjä

Kun IML-kääntäjää alettiin työstää, sen koodit siistittiin ja jaettiin omiin tiedostoihinsa. Senhetkinen toteutus oli kankea ja sekavan oloinen. Koodin alta paistoivat alkuperäisten kääntäjävaiheiden toteutukset, jotka olivat sotkeentuneet vuosien saatossa, kun uusia ominaisuuksia oli lisätty pieniä muutoksia tekemällä. Tämä oli johtanut siihen, että kääntäjä sekoitti eri vaiheita siellä täällä. Näistä syistä koko IML-kääntäjä kirjoitettiin uusiksi. Uudelleenkirjoitus toteutettiin vaihekohtaisesti. Tiettyyn vaiheeseen kuuluvat koodit kerättiin, siistittiin ja koottiin loogiseksi ja itsenäiseksi kokonaisuudeksi. Kun kaikki vaiheet oli koottu, niiden välinen kommunikointi yhtenäistettiin, jotta seuraava voisi ottaa edellisen tuotokset vastaan.

Lopputulos on siistitty kääntäjä IMLE-assemblylle, jossa on toteutettu tokenisointi, parsinta, semanttinen analysointi ja konekoodin generointi. Välikielen generointi sekä optimoijat jätettiin toteuttamatta tarpeettomina, koska kääntäjän ei tarvitse optimoida ohjelmakoodista generoimiaan konekomentosarjoja. Tämä myös helpottaa generoitujen konekielisten ohjelmien tulkintaa, koska komentojen järjestys pysyy suurilta osin muuttumattomana.

4.2 Tokenisointi

Ohjelmakoodin tokenisointi on kääntäjän ensimmäinen vaihe. Se muuttaa ohjelmakoodin tokeneiksi tai symboleiksi. (Kontekstista ja seurassa olevista henkilöistä riippuen termi saattaa vaihdella.) Tokenisoinnin hoitavasta ohjelman osasta käytetään nimitystä lekseri. Lekserillä on riittävästi tietoa ohjelmointikielestä, jotta se pystyy erottelemaan symbolit toisistaan. Yleensä symbolit erotetaan toisistaan tulostumattomilla merkeillä (välilyönti, rivinvaihto, sisennys) tai merkkien kategorian perusteella (kirjain, numero, erikoismerkki, kielikohtaisesti määritelty merkkijoukko). Jokaiselle luetulle symbolille annetaan kategoria, johon se kuuluu. Kategoriatietoa tarvitaan myöhemmissä käännösvaiheissa. IMLE:n käyttämät kategoriat ovat englanninkielisillä nimillään identifier, opcode, string, integer, address, comment, colon, var ja register. Symboleita ja kategorioita yhdistelemällä lekseri luo symbolijonon, joka syötetään eteenpäin parserille.

Tärkeä piirre lekserille on se, että se ei muuta symbolien järjestystä. Symbolit ovat siis samassa järjestyksessä, kuin missä ne esiintyvät ohjelmakoodissa. Tämä piirre myös mahdollistaa tarkkojen virheilmoitusten generoimisen, koska virhe voidaan sijoittaa tarkasti suhteessa lähdekoodiin. Lekseri ei ota kantaan symbolien järjestykseen tai siihen, onko generoitu symbolijono merkityksellinen, oikein muotoiltu tai muutenkaan toimiva ohjelma.

4.3 Parsinta

Parseri, suomeksi jäsentäjä (suomenkielistä termiä ei juurikaan käytetä), saa työstääkseen lekserin generoiman symbolijonon. Parserin tehtävä on varmistaa, että symbolit muodostavat ohjelmointikielen kieliopin mukaisia lauseita. Jos symbolijonossa ilmenee virheitä, ilmoitetaan syntaksivirheestä. Symbolijonon lauseet kootaan jäsennyspuuksi. Jäsennyspuu määrittää symbolien suoritusjärjestyksen. Lauseen sisäisen jäsennyspuun muoto määräytyy symbolien välisen hierarkian pohjalta. Mitä korkeammalla ohjelmointikielen hierarkiassa jokin symboli on, sitä latvemmaksi se sijoittuu puussa. Tämä mekanismi tarvitaan, jotta monimutkaisen lauseen suoritusjärjestys voidaan selvittää. Arkinen esimerkki vastaavasta järjestelmästä on laskujärjestyssopimus. Parsinta ei ota kantaa kasaamiensa lauseiden järjestykseen tai siihen, onko sisältö merkityksellinen.

4.4 Semanttinen analysointi

Semanttisen analysoinnin aikana parserin generoima jäsennyspuu analysoidaan. Jokaisen ohjelmalauseen on oltavat ohjelmointikielen semantiikan mukainen, eli sillä täytyy olla kielen tuntema merkitys. Semanttinen analyysi on melko yksinkertainen vaihe, mutta sen laajuus ja monimutkaisuus ovat suoraan suhteessa käännettävään ohjelmointikieleen. Kielen komentojen määrä vaikuttaa suoraan siihen, kuinka monta eri tilannetta semantiikka-analysoinnin pitää pystyä todentamaan virheettömäksi.

IMLE:n semanttinen analyysi on yksinkertainen, koska kieli on suppea. IMLE ei salli lauseita, jotka yhdistävät eri komentoja. Tämä piirre yksinkertaistaa analyysiä huomattavasti. IMLE:n analyysin aikana varmistetaan, että jokainen muuttuja ja merkintä julistetaan vain kerran, jokainen komento saa määrällisesti ja tyypillisesti oikeat operaattorit ja merkintöjen oikea muotoilu. Eräs IMLE:llä esiintyvä piirre on suoraan komennoissa käytettävien numero-, osoite- ja tekstiarvojen muuttaminen muuttujiksi tämän vaiheen aikana. Tästä seuraa, että samat useaan kertaan esiintyvät, suoraan koodiin kirjoitetut arvot eli ”magic numberit”, korvataan kaikki samalla piilomuuttujalla. Toinen harvinainen piirre IMLE:lle on muuttujien ja merkintöjen jälkietsintä, joka mahdollistaa muuttujien julistamisen niiden ensimmäisen käyttökerran jälkeen.

4.5 Optimointi ja koodin generointi

Välikielen generointi mahdollistaa kaikkien tähän mennessä suoritettujen vaiheiden prosessoriarkkitehtuuririippumattomuuden. Semanttisella analyysilla oikein kirjoitetuksi varmistettu jäsennyspuu käännetään kääntäjän sisäiseksi välikieleksi. Välikieltä voidaan optimoida välittämättä itse prosessoriarkkitehtuurista, jolle ohjelmaa ollaan kääntämässä. Tämä mahdollistaa samojen optimointirutiinien käytön, vaikka konekoodigeneraattori, -optimoija ja prosessoriarkkitehtuuri vaihtuisivatkin. Seurauksellisesti näiden vaihtaminen myös muuttuu helpommaksi, koska aiempien käännösvaiheiden ei tarvitse välittää kohdealustasta. Tämä myös mahdollistaa usealle eri prosessorialustalle tarkoitetun konekoodigeneraattorin ja -optimoijan sisällyttämisen kääntäjään.

Vastaavasti välikieli voidaan hyödyntää myös kääntäjän alkuvaiheissa. Samoja konekohtaisia vaiheita voidaan käyttää eri ohjelmointikielille, kunhan kääntäjän lekseri, parseri ja semantiikka vaihdetaan kielen mukaan. Välikieli siis tekee kääntäjästä huomattavasti joustavamman ja monitarkoituksellisemman, kuin mitä on mahdollista saavuttaa, jos se jätetään pois.

Väli- ja konekielten optimoijien tarkoitus on nimensä mukaan optimoida koodia. Optimoijat käyvät läpi niille generoituja komentojonoja ja pyrkivät yksinkertaistamaan, karsimaan, korvaamaan ja nopeuttamaan niitä. Näiden tavoitteiden saavuttamiseksi kääntäjä joutuu käyttämään monimutkaisia algoritmeja, hyödyntämään laajalti laskentateoriaa, loogista yhdenmukaisuutta ja monia muita tekniikoita. Hyödynnettäviä tekniikoita on lukemattomat määrät, eikä kaikkia pysty mitenkään käyttämään.

Optimointi voidaan suorittaa useita kertoja antamalla sille edellisen optimointikerran tuottama koodi käsiteltäväksi. Joissain tilanteissa tämä mahdollistaa tehokkaamman koodin generoinnin verrattain primitiivisellä optimoijalla.

Luonnollisesti välikielen optimointirutiinit suoritetaan, ennen kuin konekoodigeneraattoria voidaan käynnistää. Konekoodigeneraattori ottaa välikoodioptimoijan optimoiman koodin, kääntää sen tuntemakseen konekoodiksi ja antaa generoidun koodin konekoodioptimoijalle työstettäväksi. Konekoodioptimoija toistaa optimointirutiininsa välikielen optimoija tavoin, mutta itse optimointirutiini on täysin erilainen välikielen ja konekoodin erojen vuoksi.

5 PROJEKTIN JATKAMINEN

Kaikissa kolmessa projektin osassa on puutteita ja parannettavaa. IC8E- ja IMLE-ohjelmat ovat sen verran yksiselkoisia, että pienellä dokumentoinnilla niistä saisi helposti sisäistettävät kokonaisuudet.

IC8E:tä on mahdollista laajentaa kirjoittamalla sille uusia I/O-laitteita ja suunnittelemalla niiden kommunikointi. Käskykannan vaihtumisen seurauksena rikkoutuneet näkymät pitäisi korjata. Syötteen käsittelyä isäntäohjelmassa voitaisiin parantaa kirjoittamalla niiden ympärille tapahtuma-luokka, jotta niiden välittäminen IC8E:lle ja sen I/O-laitteille saataisiin hoidettua paremmin. Keskeytysten prioriteettijärjestelmän katselmointi olisi aiheellista. Nykyinen järjestelmä ei ole saanut tulikastetta ja on toimiva vain teoriassa.

IMLE:hen pitää lisätä tarkat virheilmoitukset jokaiseen vaiheeseen. Puuttuvat käännösvaiheet voitaisiin myös lisätä. Näiden lisäksi isompi IMLE:hen lisättävä ominaisuus olisi uudelleen sijoitettavien ohjelmien kääntäminen.

IMLE-assemblyn katselmointi olisi myös aiheellista. Assemblyn kielioppi, syntaksi ja komennot tulisi kaikki katselmoida, jotta mahdollisesti parannettavat aspektit löydettäisiin.

Korjattavaa ja parannettavaa löytyy, mutta mitä projekti olisi, jos sen tuloksista ei olisi mitään hyötyä? Projektin aikana toteutettujen parannusten ja lisäominaisuuksien ansiosta alkuperäiset tavoitteet on nyt mahdollista toteuttaa. Käyttöjärjestelmän ja ohjelmien moniajon toteuttaminen on mahdollista, kunhan IMLE-assembleri saadaan kääntämään uudelleensijoitettavia ohjelmia. Ajastinkeskeytykset mahdollistavat aikapohjaisen moniajohallinnoinnin käyttöjärjestelmätasolla ja massamuistista voidaan ladata käyttömuistiin suoritettavia ohjelmia. Näissä on enemmän kuin riittävästi aineksia kokonaan uuden projektin toteuttamiseen.

6 YHTEENVETO

Työn alussa tarkoituksena oli kirjoittaa moniajokykyinen käyttöjärjestelmä Kari Laitisen IC8-virtuaalitietokoneelle. Työn edetessä todettiin, että IC8:n sisäiset ratkaisut eivät taivu käyttöjärjestelmätoteutukseen. Tästä syystä IC8, IML ja IML-assembly kirjoitettiin suurilta osin uusiksi, ja siitä tuli tämän työn lopullinen aihe.

Virtuaalikoneeseen ja sen kääntäjään toteutetut uudelleenkirjoitukset paransivat koodin laatua suorituskyvyn, ymmärrettävyyden ja hallinnoitavuuden kannalta. Koodikannasta saatiin yhtenäinen ja johdonmukainen sekä huomattavasti aiempaa joustavampi. Koodia on jatkossa helppo laajentaa ja muokata. IC8E:n laajennettu muistiavaruus avaa kokonaan uusia mahdollisuuksia, koska muistia ja laitteita on käytettävissä huomattavasti aiempaa enemmän.

IML-assemblyn uudelleentoteutus IMLE:ksi toi kieleen uusia ulottuvuuksia ja monikäyttöisempiä komentoja muuttuneen kielirakenteen myötä. Useamman operaattorin antaminen yhdelle komennolle oli suurin tekijä kieltä päivitettäessä. Juuri sen toteuttaminen mahdollisti joustavamman kielirakenteen.

Laajojen päivitysten seurauksena rikkoutuneet ohjelmien osat ovat ikävä sivuseuraus. Niiden päivittäminen yhteensopiviksi nykyisten ratkaisujen kanssa on mahdollista, mutta ei useimpien osalta maksa vaivaa.

Työn aikana ohjelmointikielen, kääntäjän ja virtuaalikoneen suunnitteluun liittyvät prosessit ja tekniikat tulivat tutummiksi. Syvempää tietämystä tietokoneiden laitteistosta ja erityisesti keskeytyksistä saatiin kerättyä tutkintavaiheen aikana. Tätä tietoa voidaan jatkossa hyödyntää monella eri tavalla.

IC8:aan ja IML:ään toteutetut parannukset mahdollistavat alkuperäisen suunnitelman toteuttamisen eli moniajokelpoisen käyttöjärjestelmän kirjoittamisen.

LÄHTEET

1. Laitinen, Kari. 2006. A Natural Introduction to Computer Programming with Java. Trafford Publishing.
2. The Instruction Cycle. Bristol Community College. Saatavissa: http://www.c-jump.com/CIS77/CPU/InstrCycle/lecture.html. Hakupäivä: 18.3.2018
3. Computer port (hardware). 2018. Wikipedia. Saatavissa: https://en.wikipedia.org/wiki/Computer_port_(hardware). Hakupäivä: 18.3.2018
4. Difference between port mapped and memory mapped access?. 2014. Varaquilex. Saatavissa: https://superuser.com/questions/703695/difference-between-port-mapped-and-memory-mapped-access/703705. Hakupäivä: 18.3.2018
5. Memory and I/O buses. 2007. Stanford Computer Science Department. Saatavissa: http://www.scs.stanford.edu/07au-cs140/notes/l11.pdf. Hakupäivä: 18.3.2018
6. Memory-mapped I/O. 2018. Wikipedia. Saatavissa: https://en.wikipedia.org/wiki/Memory-mapped_I/O. Hakupäivä: 18.3.2018
7. Mongenel, Randy. GameBoy Memory Map. Saatavissa: http://gameboy.mongenel.com/dmg/asmmemmap.html. Hakupäivä: 18.3.2018
8. Instruction set architecture. 2017. Wikipedia. Saatavissa: https://en.wikipedia.org/wiki/Instruction_set_architecture. Hakupäivä: 18.3.2018
9. Laitinen, Kari. 1998. abcde.iml. Saatavissa: http://www.naturalprogramming.com/javabookprograms/icom/abcde.iml. Hakupäivä: 6.4.2018
10. Compiler Design Tutorial. Tutorials Point. Saatavissa: https://www.tutorialspoint.com/compiler_design/index.htm. Hakupäivä: 18.3.2018

IC8E virtual machine instruction set **11.03.18**

Unlike most assembly languages, IC8 assembly does not abbreviate its instructions to make it easier to understand.

Term	Explanation
Register	A CPU register
Address	A memory address. Valid range is from 0x0100 to 0xFFFF (all but ISR vector reserved memory).
Location	A memory address or register
Label	A memory address identified by a jump label.
Value	A value in memory address or register (Contents of a location)
ISR address	A memory address pointing to interrupt vector table (0x0000 – 0x00FF).

Instruction	Opcode	Operands Primary	Secondary	Function
nop	0x00	-	-	No operation, uses a single processor cyckle.
stop	0xB2	-	-	Stops the processor.
idle	0xD1	-	-	Puts the processor into a sleep like state. Interrupts still function.
end_idle	0xD2	-	-	Wakes the processor from its idle sleep.
shutdown	0xDD	-	-	Shuts processor down.
get	0x04	address	-	Gets address in register M and writes it to address.
set	0x05	address	-	Sets register M to an address.
move_in	0x06	register	address/register	Moves value in memory or register to register.
move_out	0x07	address	address/register	Moves value in register to address or register.
move_m_in	0x08	register	address/register	Moves value in location identified by secondary operand to memory location pointed by register.
move_m_out	0x09	address	address/register	Moves value in location identified by secondary operand to memory location pointed by address.
move_p_in	0x10	register	address/register	Moves value in location identified by secondary operand to register.
move_p_out	0x11	address	address/register	Moves value in location identifed by secondary operand to address.
push	0xA2	register	-	Push value in register to stack.
pop	0xA4	register	-	Pop value from stack to register.
call	0x81	address	-	Calls a subroutine and pushes current register states to stack.
return	0x82	-	-	Return from suroutine and pulls previous register states from stack.
return_isr	0x84	-	-	Return from interrupts service routine (ISR).
load_isr	0x83	ISR address	address	Sets ISR vector item to a subroutine start address.
add	0x30	register	register	Adds value to register.
subtract	0x32	register	register	Subtracts value from register.
divide	0x34	register	register	Divides register with value.
multiply	0x36	register	register	Multiplies register with value.
increment	0x38	register	-	Increments register or value behind memory address by one.
decrement	0x3A	register	-	Decrements register or value behind memory address by one.
not	0x3C	register	-	Negates value in given register using binary operations.
compare	0x40	register	register	Compares two values and sets processor's internal compare flags which can be used for jumps.
jump	0x41	label	-	Jumps to given memory address.
jump_eq	0x43	label	-	Jumps to given memory address if compare flags denote equal.
jump_lt	0x45	label	-	Jumps to given memory address if compare flags denote less than.
jump_gt	0x47	label	-	Jumps to given memory address if compare flags denote greater than.
jump_le	0x49	label	-	Jumps to given memory address if compare flags denote less or equal.
jump_ge	0x4B	label	-	Jumps to given memory address if compare flags denote greater or equal.
jump_ne	0x4D	label	-	Jumps to given memory address if compare flags denote not equal.

```
IMLE Grammar expressed in extended Backus-Naur form
'''''''''''''''''''''''''''''''''''''''''''''''''''

letter          = "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" | "J"
                | "K" | "L" | "M" | "N" | "O" | "P" | "Q" | "R" | "S" | "T"
                | "U" | "V" | "W" | "X" | "Y" | "Z" | "a" | "b" | "c" | "d"
                | "e" | "f" | "g" | "h" | "i" | "j" | "k" | "l" | "m" | "n"
                | "o" | "p" | "q" | "r" | "s" | "t" | "u" | "v" | "w" | "x"
                | "y" | "z" ;

digit           = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;

opcode          = "nop"         | "get"          | "set"          | "move_in"
                | "move_out"    | "move_m_in"    | "move_m_out"   | "move_p_in"
                | "move_p_out"  | "add           | "subtract"     | "divide"
                | "multiply"    | "increment"    | "decrement"    | "not"
                | "compare"     | "jump"         | "jump_eq"      | "jump_lt"
                | "jump_gt"     | "jump_le"      | "jump_ge"      | "jump_ne"
                | "push"        | "pop"          | "call"         | "return"
                | "load_isr"    | "return_isr"   | "stop"         | "idle"
                | "end_idle"    | "shutdown" ;

identifier      = letter , {letter | digit | "_" } ;

integer         = { digit } , [ "H" | "h" ] ;

address         = "@" , integer ;

register        = "A" | "B" | "M" | "S" | "PC" ;

string          = "'" , { character | digit | ? any ASCII character ? } , "'"
                | '"' , { character | digit | ? any ASCII character ? } , '"' ;

operand         = register | identifier | letter | address | integer ;

label           = identifier , ":" ;

variable        = "var" , identifier , integer | address | string

instruction     = opcode , [ operand , [ operand ] ] ;

statement       = label | variable | instruction;

grammar         = { statement } ;
```