TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond
Thomas Johann Seebecki elektroonikainstituut

IE40LT
Jaan Hendrik Murumets 142951IALB

C-KEELE KOMPILAATORI REALISEERIMISE ANALÜÜS MIKROKONTROLLERILE PISKE

Bakalaureusetöö

Juhendaja: Eero Haldre
Dipl. Ins.
Kaasjuhendaja: Peeter Ellervee
PhD

Tallinn 2017

Autorideklaratsioon

Kinnitan, et olen koostanud antud lõputöö iseseisvalt ning seda ei ole kellegi teise poolt varem kaitsmisele esitatud. Kõik töö koostamisel kasutatud teiste autorite tööd, olulised seisukohad, kirjandusallikatest ja mujalt pärinevad andmed on töös viidatud.

Autor: Jaan Hendrik Murumets

22.05.2017

Annotatsioon

Töö eesmärk on analüüsida, millised võimalused on (pehmetuumalisele) protsessorile PISKE C-keele kompilaatori saamiseks. Töös vaadeldakse esmalt PISKE ehitust, eelkõige kompilaatori ja programmeerija vaatenurgast. Seejärel kompileerimise protsessi nii tervikuna kui ka üksikute sammudena, jällegi sihitud arhitektuuri vaatenurgast. Jõudes järeldusele, et vaadelda tasub vaid korralikke, hästi toetatud kompilaatoreid, võrreldakse kahte enim kasutatavat kompilaatorite perekonda – GCC ja LLVM. Seejärel selgitatakse, miks töös püstitatud eesmärgi täitmiseks on just LLVM projekt sobivaim. Viimaks selgitatakse ülevaatlikult milliseid sammud tuleb selle kompilaatori portimiseks astuda.

Lõputöö on kirjutatud eesti keeles ning sisaldab teksti 28 leheküljel, 7 peatükki, 12 joonist, 1 tabelit.

Abstract

Analysis of implementing a C-language compiler for PISKE MCU

This thesis analyses methods to compile the C programming language for a softcore target named PISKE. It starts by covering the architecture of PISKE from a compiler's and programmer's perspective. It goes on to describe the process of compiling step-by-step from C source files down to binaries with added emphasis on the intended target – PISKE. It explains, by example, the concepts of preprocessing source files by the C preprocessor, the compilation phase itself glancing over tokens, abstract syntax trees, intermediate representation, optimization and lowering intermediate representation into assembly code. It continues with an overview of the work done by the assembler and linker.

The author then concludes the best course of action to be porting one of the more performant and popular compiler projects, nominating both GCC and LLVM and giving a cursory introduction to them and their history. The thesis then goes on to compare the two in aspects relevant in the set context: the structures of the projects, use of intermediate language, extensibility both in terms of a more complete toolchain and for use on the intended target.

They conclude that LLVM is far better suited for the task, but may lead to extra work having to be performed should there be a need for porting a debugger for the PISKE MCU. Lastly, the author gives an overview of the steps necessary to retarget the LLVM project's compiler to PISKE.

The thesis is in Estonian and contains 28 pages of text, 7 chapters, 12 figures, 1 table.

Lühendite ja mõistete sõnastik

AST	*Abstract Syntax Tree*, abstraktne süntaksipuu
BP	*Base Pointer*, indeksregister
DMA	*Direct Memory Access*, otsene mälupöördus
FIFO	*First In First Out*, pinulaadne andmestruktuur
FPGA	*Field Programmable Gate Array*, väliprogrammeeritav väravamassiiv
GAS	*GNU Assemble*, GNU assembler
GCC	*GNU Compiler Collection*, GNU kompilaatori kollektsioon
GNU	*GNU's Not Unix*
GPIO	*General Purpose Input-Output*, digitaalne sisend-väljund
GPL	*(GNU) General Public License*. GNU litsents
HDL	*Hardware Design Language*, riistvara kirjeldamis keel
HSA	*Heterogeneous System Architecture*, heterogeenne süsteemiarhitektuur
IR	*Intermediate Representation*, vahepealne esitus
LTO	*Link Time Optimisation*, sidumisaegne optimisatsioon
LUT	*Look Up Table*, otsingutabel
PC	*Program Counter*, programmiloendur
PS4	*PlayStation 4*
RISC	*Reduced Instruction Set Computing*, kärbitud käsustikuga arvuti
RTL	*Register Transfer Language*, vahepealse esituse vorm
SDK	*Software Development Kit*, Tarkvaraarenduskomplekt
SPIR	*Standard Portable Intermediate Representation*, vahepealse esituse vorm
UART	*Universal Asynchronous Receiver/Transmitter*, standardne jadaühendus
USB	*Universal Serial Bus*

Sisukord

Jooniste loetelu

Tabelite loetelu

1 Sissejuhatus

Antud bakalaureusetöö eesmärk on uurida ja võrrelda erinevaid viise, kuidas saada mõne laialdaselt kasutatava programmeerimiskeele kompilaator väljastama ühele spetsiifilisele mikrokontrollerile arusaadavat masinkoodi. Silmas peetava protsessori näol on tegemist kodumaise *softcore* ehk pehmetuumalise PISKE-nimelise mikrokontrolleriga (vt Lisa 1), millel hetkel puudub täielikult võimalus kõrgema taseme keeles programmeerida.

Arvestades asjaolu, et tegemist on vägagi minimalistliku ja piiratud ressurssidega protsessoriarhitektuuriga, jäävad realistlikult kasutatavaks vaid üksikud keeled, millest enim kasutust leiab kindlasti C-keel [1]. Seda nii sardsüsteemide valdkonnas kui ka üldisemalt. Viimase tõttu on uurimustöö kitsendatud vähemalt C-keelt sisendina kasutavatele kompilaatoritele.

Autori esialgne eesmärk oli ise (tõenäoliselt) piiratud funktsionaalsusega C-keele [2] kompilaatori kirjutamine, kuid selline kodukootud tööriist ei omaks kuigi palju praktilist väärtust, jäädes vanematele ja professionaalsematele tegijatele alla funktsioonirikkuse, optimeerituse, optimeerimisvõime ja ilmselt nii mõnegi muu aspektiga.

Kuna aga (pehmetuumaline) protsessor koos talle suunatud pädeva C kompilaatoriga on oluliselt väärtuslikum kui uus kompilaator või protsessor iseseisvalt, on otstarbekas vaadelda olemasolevaid tööriistu ja võimalusi juba tehtud tööd eesmärgipäraselt rakendada. Niisiis tuli hakata vaatama juba eksisteerivate kompilaatorite poole ning välja filtreerida vaid parimad kandidaadid. Selleks oli vaja piiritleda, mille alusel valik teha.

Selged on paar asja – esiteks: nii piiratud ressurssidega mikrokontrolleri puhul on väga oluline, et kompilaator oleks võimalikult tark - vastasel juhul kulub programmeerijal ebaproportsionaalselt palju aega ja vaeva käsitsi masinkoodi ehk *assembleri* optimeerimise ja teelehtedelt kompilaatori kavatsuste lugemise peale. Teiseks on kompilaatori *retargetimine* ehk ümbersuunamine küllalt töömahukas protsess, mida saab oluliselt leevendada kui jääda populaarsemate ja paremini dokumenteeritud tööriistade juurde.

Kolmas, kuid kindlasti mitte vähem tähtis aspekt on valitud mikrokontrolleri arhitektuurist tulenev lisasamm – nimelt on valitud pehmetuumaline protsessor Harvardi arhitektuuriga, mis tähendab, et kood ja muutujad on eraldi adresseeritavates mäludes. Kuigi tegemist ei ole millegi eksootilisega, on enamus laiatarbe kompilaatoreid mõeldud eelkõige von Neumanni tüüpi protsessoritele, kus muutujad ja kood on samas mälus. Viimane tingib mõningate modifikatsioonide tegemist olemasolevates kompilaatorites.

Just selliste (ja teiste) eripärade tõttu osutus mõistlikumaks kasutada LLVM projektide teekide kogu ja vaadelda spetsiifiliselt nendes kitsendustes C keele kompilaatori uuele platvormile portimise protsessi.

2 PISKE mikrokontroller

PISKE on Eestis arendatud pehmetuumaline mikrokontroller. Nagu tema nimi võib vihjata on tegemist küllalt pisikese mikrokontrolleriga. Autori sõnul on tegemist maailma väikseima kompilaatorisõbraliku 16-bitine pehmetuumalise protsessoriga (vt Lisa 1) . Protsessor mahub ära kõigest 260 4-sisendiga LUT-i (*Look Up Table*) sisse, mis võimaldab seda kasutada nii väga odavate ja väikeste (seda nii füüsiliste mõõtmete kui ka LUT-de arvu poolest) FPGA-de (*Field Programmable Gate Array*) sees kui ka oluliselt suuremate sees, juhul kui on tarvis rakendada kas väga keerukaid või mahukaid perifeeriaseadmeid. Mikrokontrolleri väiksusest olenemata on tegemist igati võimeka pehmetuumalise protsessoriga. PISKE on implementeeritud Verilog HDL-s (*Hardware Design Language*).

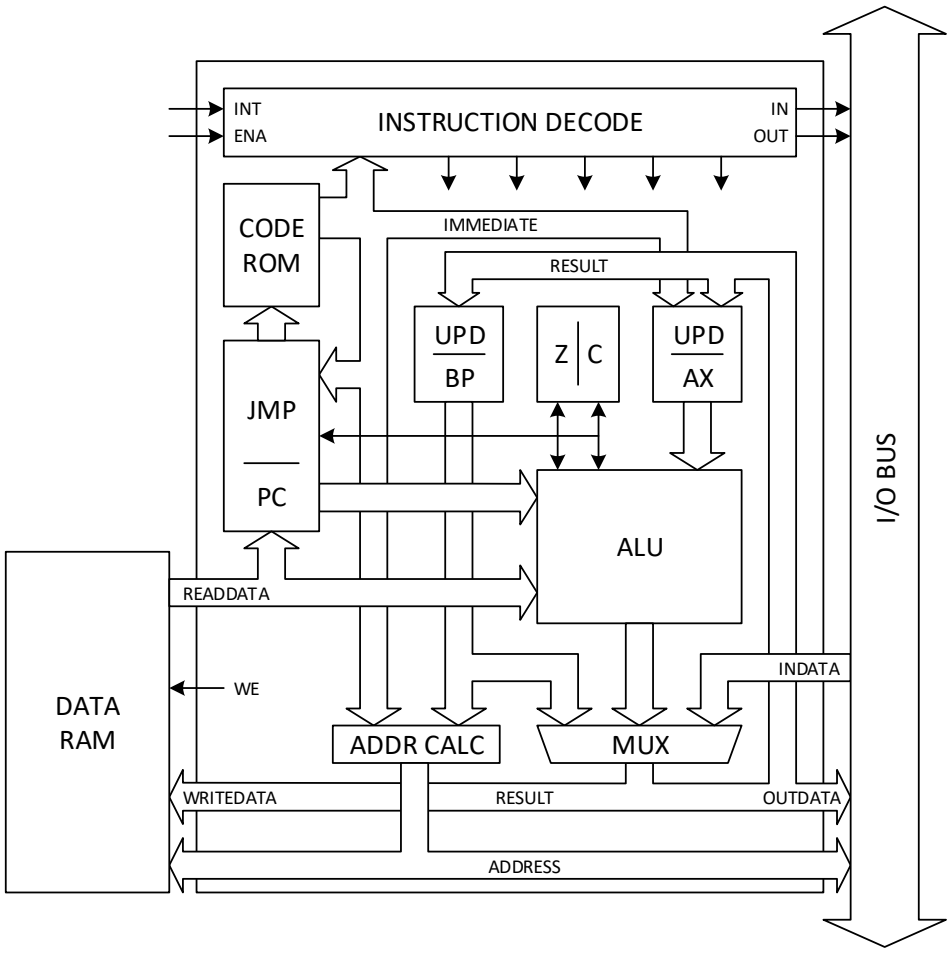
Tema väiksuse tõttu on PISKE rakendatav näiteks Lattice iCE40 seeria ~$1.5 maksva FPGA peal, positsioneerides ta 3-4 korda odavamaks sarnase väljaviikude arvu ja USB (*Universal Serial Bus*) võimekusega Atmega16U2-st, pakkudes samal ajal oluliselt suuremat USB läbilaskevõimet tänu USB välisseadme võimekusele mäluga otse töötada. Lisaks on PISKE võimeline töötama väga energiasäästlikult tänu katkestuskontrolleri funktsioonile seisata protsessori töö täielikult kuni esimese katkestuseni. See omadus teeb PISKE kasutamise ideaalseks näiteks sellistes andmehõive rakendustes, mille puhul on vaja tegutseda vaid mõne välise seadme poolt genereeritud katkestuse peale.

2.1 Arhitektuur

PISKE-l on 16 biti laiused sõnad ja (tinglikult) kolm erinevat aadressisiini. Need on teineteisest täielikult sõltumatud koodi-, andmete- ja perifeeriasiinid, kusjuures iga siin lubab adresseerida kuni 2^{16} * 16 bitti ehk 128 kilobaiti andmeid. Loomulikult on maksimaalne adresseeritav mälu suurus teoreetiline, tegelikkuses on mälude suurused piiratud eelkõige FPGA vaba sisemälu ressursside ja nende jaotusega.

Nagu eelnevast välja võib lugeda on PISKE näol tegemist puhta Harvardi arhitektuuriga protsessoriga, millega kaasneb omajagu häid külgi just tavalise sardsüsteemi rakenduses. Tavaliste protsessori käskudega ei ole võimalik koodimälu lugeda ega kirjutada, näiteks ei ole võimalik pinu üle jooksmise korral seadmel olevat koodi kahjustada. Koodimälu ligipääsu koha pealt on erandiks sobivad perifeeriamoodulid, millel saab võimaldada ka koodimälu modifitseerimist. Ülevaatlikkuse huvides on Joonis 1 kujutatud PISKE arhitektuuriskeem.

PISKE-l on implementeeritud minimalistlik RISC (*Reduced Instruction Set Computing*) arhitektuur, mis hõlmab endas 49 instruktsiooni. PISKE teeb küllalt eriliseks fakt, et iga instruktsioon saab täidetud ühe taktsageduse tsükliga, mille saavutamiseks kasutatakse ära taktsageduse nii tõusvad kui langevad ääred. PISKE on disainitud jooksma 24 MHz juures, kuid teda on edukalt simuleeritud jooksmas ka kuni 183 MHz taktsagedusega.



Joonis 1. PISKE arhitektuuriskeem (vt Lisa 1).

PISKE-l on kolm baasregistrit: *pc* (*Program Counter*) ehk programmiloendur, *ax* ehk akumulaator ja *bp* (*Base Pointer*) ehk indeksregister. Otse adresseeritavad on 512 esimest sõna (16 bitti lai) mälust, mida võib kohelda kui universaalseid registreid, kuna aritmeetika instruktsioonid võimaldavad ühe tsükliga teha „loe-muuda-kirjuta“ operatsiooni.

Tavapäraseid registreid (nagu AVR-s R0-R31 [3]) PISKE-l ei eksisteeri, kuna nende olemasoluks ei ole põhjust – pöördumise kiirus suvalisele mälupesale võtab alati ühe tsükli. Lisaks on PISKE olekuregistris kaks lippu: Z ehk nulli lipp ja C ehk ülekande lipp.

Ühe argumendiga aritmeetikakäsud (nihked) saavad kasutada vaid akumulaatorit ning kahe argumendiga käskude üheks argumendiks peab olema akumulaator, teiseks argumendiks mälupesa. Operatsiooni tulemus salvestatakse kas akumulaatorisse või mälupesasse.

Lisaks otse adresseeritavatele esimesele 512-le mälupesale on võimalik relatiivselt adresseerida indeksregistrile (*bp*) 256 eelnevat ja 255 järgnevat mälupesa. Suhtelist adresseerimist rakendades on võimalik täita pinu, funktsioonide lokaalsete muutujate ja funktsioonile argumentide edastamise vajadused. Viimast asjaolu arvestades ei ole PISKE-l tavapärast pinuviita üldse olemas. Funktsiooni väljakutsumise (*call)* peale salvestatakse automaatselt indeksregistri *bp* järgi väljakutsuva funktsiooni naasmisaadress välja kutsuva funktsiooni lokaalmuutujana.

PISKE-l puudub sisse ehitatud võimekus teha ujukomatehteid.

2.2 Perifeeriaseadmed

Lisaks protsessorile sisaldab PISKE mikrokontroller ka talle arendatud komplementaarset minimalistliku välisseadmete moodulite perekonda.

PISKEs on 1-16 sisendit katkestuste kontrollerisse, mis on suuteline 11 taktiga hüppama kõige kõrgema prioriteediga katkestuse täitmise rutiini peale. Neist 10 kulub konteksti salvestamiseks ja aktiivse katkestuse valimiseks tarkvaras implementeeritud katkestuse halduri poolt. Katkestuste teenindamine on kooperatiivne, seega ei saa üks katkestus teise teenindamise ajal katkestuse täitmist uuesti katkestada. Uus katkestus teenindatakse peale eelnevaga lõpetamist.

PISKE toetab mitut erinevat energiasäästlikkuse taset. Taktsageduse kontroller toetab kahte erinevat taktsagedust ja lubab neid tõrgeteta suvalisel hetkel ümber lülitada.

Koos katkestusekontrolleri mooduliga on võimalik PISKE viia sellisesse režiimi, kus ei tööta ükski protsessori osa. Sellisel juhul saab protsessor tööga jätkata vaid välise katkestuse saabumise peale. Antud režiim on kõige kokkuhoidlikum ja ei tarvita peaaegu üldse voolu.

Lisaks kolmepordisele põhimälule (protsessor, DMA (*Direct Memory Access*) ja USB/*bus-master*) ja protsessori standardvarustusse (taktsageduse, katkestuste ja GPIO (*General Purpose I/O*) kontroller ning taimer) kuuluvatele perifeeriaseadmetele on PISKE-s implementeeritud ka striimide DMA I/O kontroller, FIFO (*First In First Out*) ja UART (*Universal Asynchronous Receiver/Transmitter*) I/O (*Input/Output*) kontroller, I2S (*Inter Integrated Circuit Sound*) liides FIFO kontrollerile ning USB funktsiooni kontroller, mis toetab *Low-Speed*, *Full-Speed* ja *High-Speed* standardeid *bulk*, *interrupt* ja isokroonse lõpp-punkti režiimis.

2.3 Käsustik

PISKE on oma struktuurilt väga lihtne ja väike protsessor. See väljendub ka tema käsustikus, mis kirjutamise hetkel koosnes 49 käsust. PISKE käsustik on paljus inspiratsiooni saanud x86 arhitektuurist, seetõttu kattuvad näiteks kõik käskude mnemoonilised nimed x86 instruktsioonide omadega.

Kõik PISKE käsud koosnevad 5-6 bitisest käsukoodist. Aritmeetika- ja loogikakäsud koosnevad 6 bitist käsukoodist ning kasutavad lisaks veel suhtelise adresseerimise lippu, ülejäänud bitid on jäetud operandidele. *ax* registri laadimiseks mõeldud käsud lippe ei vaja, selle asemel on tegemist 11-bitise operandiga ja 5-bitise käsukoodiga. Juhul kui on tarvis teha operatsioone 16-bitiste muutujatega on olemas spetsiaalne käsk, mis laeb *ax* registrisse muutuja ülemised 5 bitti. Sellisel juhul on operandi jaoks kasutusel bitid D7:D3. *ax* registri laadimise käskude struktuuriga on ka tingimuslikud ja tingimusteta siirdekäsud, mille puhul laetakse operandi 11-bitine suhteline aadress. Vahetu argumendiga siirdekäsud ei toeta absoluutset adresseerimist. Täpsem informatsioon käskude kohta, mida PISKE täita suudab on toodud Lisas 1 koos ammendava kirjeldusega.

Nagu eelnevalt mainitud puudub PISKE-l riistvaraline pinu. Sellise hädavajaliku funktsionaalsuse tagab *bp* register ja *call* instruktsiooni riistvaraline implementatsioon. Aktiivsele alamprogrammile kuuluvale pinupiirkonnale viitab üldjuhul bp. Juhul kui bp registrit on tarvis kasutada muudeks otstarveteks saab *bp* registri sisu liigutada mõnda otseadresseeritavasse registrisse, kust selle saab vajadusel ka ühe tsükliga tagasi laadida. Tihti on selleks otstarbeks määratud üks spetsiifiline register, mida võib tinglikult kutsuda *fp*-ks ehk *frame pointer*-ks.

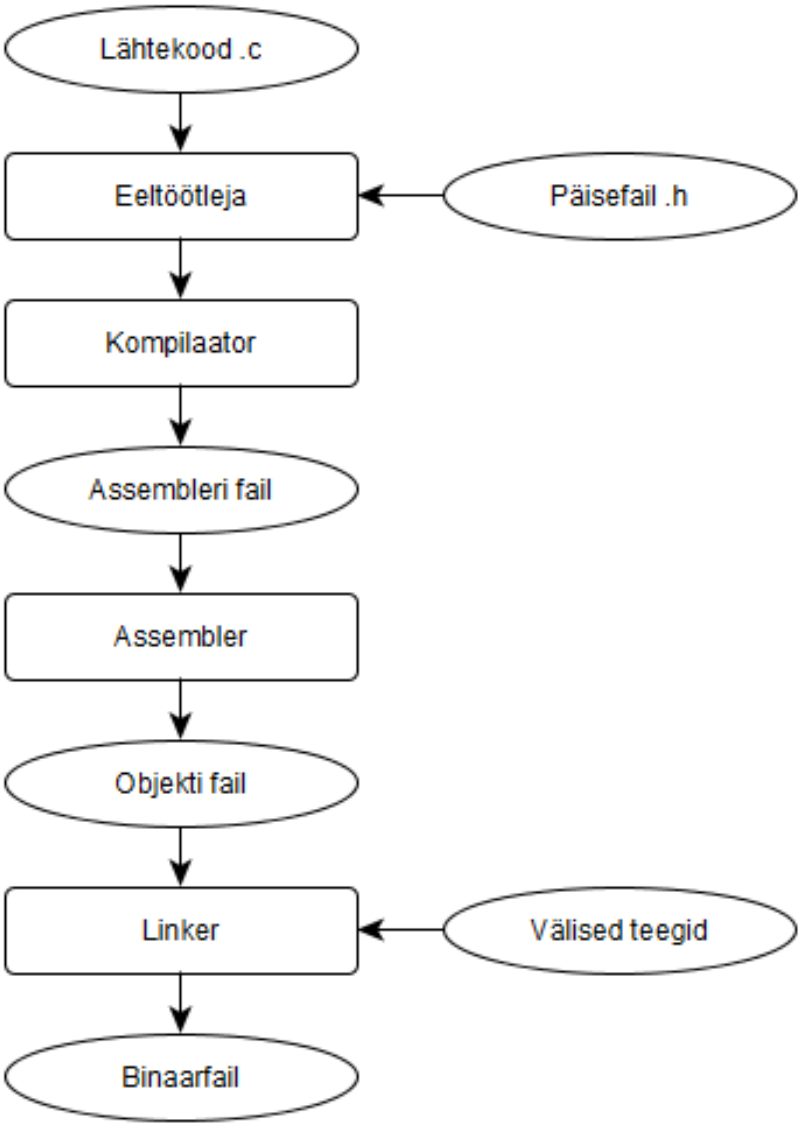
PISKE puhul on alamprogrammi sisenedes tarvis eraldada pinus iga alamprogrammi lokaalmuutujate ja argumentide jaoks vajalik ruum ning omistada viit pinupiirkonnale *bp* või *fp* registrisse. Pinu puudumisest tingitult ei eksisteeri PISKE-l *push* ja *pop* instruktsioone. Nende funktsionaalsusega samaväärne on *bp* suhtes argumentide liigutamine *mov* instruktsiooniga. PISKE peal salvestab *call* käsk ise alamprogrammist tagasipöördumise aadressi aadressile [bp+0] välja kutsutud funktsiooni suhtes. Sellest tulenevalt on vajalik, et *bp* registris asuks viit alamprogrammile kuuluvale mälupiirkonnale. Iga alamprogramm, milles kutsutakse veel omakorda välja alamprogramme, peab eraldama endale pinupiirkonna. Selleks tuleb eraldada endale pinupiirkond lahutades alamprogrammi sisenedes *bp* registrist selle alamprogrammi poolt kasutatava pinupiirkonna suuruse. Alamprogrammist väljudes tuleb *bp* väärtus taastada selleks, mis ta oli alamprogrammi sisenedes. Seejärel tuleb kasutada *ret* instruktsiooni, mis tegelikult vastab käsule „jmp [bp+0]“. Selleks, et lihtsustada masinkoodi analüüsimist, genereerimist ja optimeerimist salvestatakse aadressile [bp-1] pinupiirkonna suurus.

3 Kompileerimise protsess

Selleks, et adekvaatselt selgitada, mida bakalaureusetöö autor kokkuvõttes teha kavatseb, tuleb esmalt natuke kirjeldada kompileerimise protsessi. Täpsustuseks – kuna eesmärk on eelkõige C-keelt valitud mikrokontrollerile kompileerida, tuleb ka selgitus C-keele keskne.

Terminit „kompilaator“ kasutatakse kahes kontekstis. „Kompilaator“ võib viidata nii tarkvarale, mis viib läbi esialgse tekstifaili sisu teisenduse lähtekoodist masinkoodi kui ka tööriistade kollektsioonile, mis viib läbi terve kompileerimise protsessi alustades eeltöötluse ehk *preprocessing*-ga ning lõpetades *link*-mise ehk sidumisega. Autor kasutab spetsiifilisemat terminit kui on tarvis mingit osa kompileerimise protsessist eristada või mingit erinevust täpsustavalt esile tuua. Kui erisus ei ole aga antud kontekstis oluline kasutab autor sõna „kompilaator“ selleks, et kirjeldada tööriista või nende komplekti, mis viivad läbi terve kompileerimise protsessi.

Kompileerimine toimub mitmes järgus, kuid laias laastus taandub protsess neljale faasile: eeltöötlus ehk *preprocessing*, *parse*-mine ehk töötlemine, *assembly*-mine ehk kokkupanek ning *link*-mine ehk sidumine. Joonis 2 kirjeldab tavapärast C keele kompileerimise käiku.



Joonis 2. Kompileerimise sammud

Järgnevalt selgitab autor lähemalt, mida mingi samm endast kujutab ning, kuidas ühe faasi väljundist saab järgne faasi sisend. Illustratiivseteks näideteks kasutab autor ka C keeles kirjutatud näidiskoodi ning selle erinevaid transformatsioone läbi *clang* (LLVM teekide ja tööriistade põhjal ehitatud C kompilaator) ja GCC (*GNU Compiler Collection*) kompilaatorite kompileerimisfaaside.

3.1 Näidiskood

Selleks, et paremini selgitada, millised transformatsioonid programmeerija kirjutatud lähtekood kompileerimise käigus läbib, kasutab autor all näidatud koodi.

Joonis 3 Naidis.h fail. kujutatud päisefail sisaldab funktsiooni prototüüpe ning ühte *#define* direktiivi, et paremini illustreerida eeltöötluses läbi viidavaid samme.

```
#ifndef NAIDIS_H
#define NAIDIS_H

#define PARAMEETRITE_ARV  4

int add(int a, int b);
int subtract(int a, int b);
int multiply(int a, int b);
int divide(int a, int b);

#endif /* NAIDIS_H */
```

Joonis 3 Naidis.h fail.

Joonis 4 kujutatud koodifailis on kasutatud ülal *#define* direktiiviga defineeritud konstanti, funktsioonide deklaratsiooni ning C keele standardile vastavalt *main* funktsiooni. Koodis kasutatakse *#include* direktiivi selleks, et C standardteekides olevate funktsioonide prototüübid kompilaatorile tuttavaks teha. Nende tegeliku paigutamise eest kompileerimisel saadavasse koodi vastutab *linker*.

```c
// Programm võtab parameetriteks tehte formaadis "a tehe b", nt "a + b".
#include <stdio.h>
#include <stdlib.h>
#include "naidis.h"
int main(int argc, char **argv) {
        // Kontrollime, et sisestatud oleks õige arv parameetreid
        if (argc != PARAMEETRITE_ARV) {
                printf("Sisestasite vale arvu parameetreid\n");
                return -1;
        }

        int A = atoi(argv[1]);
        char tehe = *argv[2];
        int B = atoi(argv[3]);
        int tulemus;

        switch (tehe) {
        case '+':
                tulemus = add(A, B);          break;
        case '-':
                tulemus = subtract(A, B);     break;
        case '*':
                tulemus = multiply(A, B);     break;
        case '/':
                tulemus = divide(A, B);       break;
        default:
                printf("Tundmatu tehe!\n");
                return -2;
        }

        printf("%d %c %d = %d\n", A, tehe, B, tulemus);
        return 0;
}

int add(int a, int b) {      // Tagastab parameetrite a ja b summa
        return a + b;
}

int subtract(int a, int b) { // Funktsioon tagastab parameetrite a ja b vahe
        return a - b;
}

int multiply(int a, int b) { // Tagastab parameetrite a ja b korrutise
        return a * b;
}

int divide(int a, int b) {   // Tagastab parameetrite a ja b jagatise
        return a / b;
}
```

Joonis 4 Naidis.c fail.

3.2 Eeltöötlus

Käsuga „*clang naidis.c -o naidis -E“* saab paluda kompilaatoril (antud juhul LLVM teekide ja tööriistade põhjal ehitatud *clang,* kuid GCC annab sisuliselt sama väljundi) alustada kompileerimise protsessi ning lõpetada see kohe peale eeltöötlemise faasi. Tekstifaili *naidis* salvestatakse lähtekood nii nagu selleks hetkeks kompilaator seda näeb.

Siin ei ole veel tegelikult tegemist puhta C keelega vaid C *preprocessor*-ga. C *preprocessor* on osa C keele standardist ISO/IEC 9899. Viimane on C-ga väga tihedalt seotud „keel“, mida kasutatakse selleks, et erinevate tingimuste (sihitud platvorm, soovitud funktsionaalsus, olemasolevad teegid jne) põhjal viia sisse muudatusi lähtekoodi, mis jõuab kompilaatorini. Näiteks asendatakse kõik *#define* direktiiviga deklareeritud konstandid koodis olevates *token*-tes ehk tähistes. Nii saab reast „*#define PARAMEETRITE_ARV 4“* saadud vihje põhjal tähisest „PARAMEETRITE_ARV“ konstant 4. Samuti asendab C *preprocessor* kõik *#include* direktiivid päisefailide sisuga, tingimuslike direktiivide #if, #ifdef jms hindamise tulemusega, asendab kommentaarid tühikutega jne.

Antud näite põhjal saab Joonis 4 kujutatud koodist Joonis 5 kujutatud kood. Joonisel ei ole kujutatud eelnevat 5803 rida koodi, mis kujutab endast sihitud platvormile (antud juhul Windows x64) standardteekide päiste <stdio.h> ja <stdlib.h> eeltöödeldud sisu.

Näites on iga kommentaar asendatud üksiku tühikuga, sisse on viidud ülalmainitud tähiste asendused, näidis.h päisefaili sisu on pärast eeltöötlemist otse asendatud reale #include "naidis.h" ning kompilaatorile on jäetud vihjed, mis algavad „#“-ga. Näiteks tähendab #pragma pack(pop), et kõik järgnev tuleks pakkida vastavalt varem *push*-tud joondamisreeglitele (et nii kood kui ka muutujad oleksid mälus järjestikku). Kuna eeltöötlemine muudab paratamatult ka lähtekoodi failide sisu aga kompilaatori kasutajad tahaksid leitud vigade ja hoiatuste korral teada nende täpseid asukohti originaalses failis, jäetakse sisse ka tähised lisatud/kustutatud ridade kohta. Selliseid tähiseid nimetatakse *linermarker*-teks [4].

Samuti hinnati #ifndef-i tõesust ning tehti sellele vastavad asendused – antud juhul tuli alles jätta #ifndef ja #endif direktiivide vahele jäänud read, kuna märget NAIDIS_H varemalt ei olnud defineeritud.

```
#pragma pack(pop)
# 3 "naidis.c" 2
# 1 "./naidis.h" 1
int add(int a, int b);
int subtract(int a, int b);
int multiply(int a, int b);
int divide(int a, int b);
# 4 "naidis.c" 2
int main(int argc, char **argv) {
        if (argc != 4) {
                printf("Sisestasite vale arvu parameetreid\n");
                return -1;
        }
        int A = atoi(argv[1]);
        char tehe = *argv[2];
        int B = atoi(argv[3]);
        int tulemus;
        switch (tehe) {
        case '+':
                tulemus = add(A, B); break;
        case '-':
                tulemus = subtract(A, B); break;
        case '*':
                tulemus = multiply(A, B); break;
        case '/':
                tulemus = divide(A, B); break;
        default:
                printf("Tundmatu tehe!\n");
                return -2;
        }
        printf("%d %c %d = %d\n", A, tehe, B, tulemus);
        return 0;
}

int add(int a, int b) {
        return a + b;
}

int subtract(int a, int b) {
        return a - b;
}

int multiply(int a, int b) {
        return a * b;
}

int divide(int a, int b) {
        return a / b;
}
```

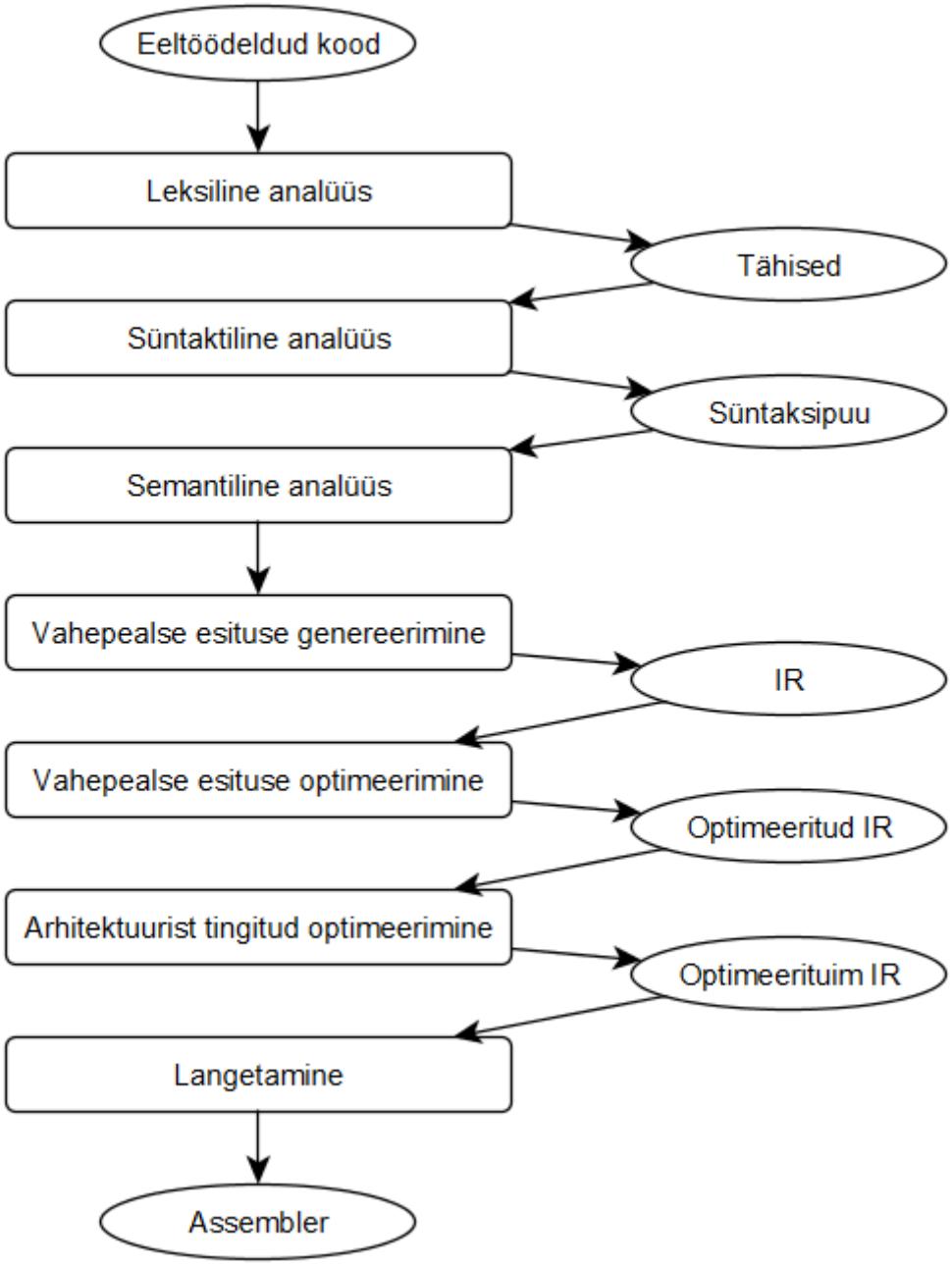
Joonis 5. Eeltöötluse tulemus

3.3 Kompileerimine

Pärast eeltöötlemise faasi suunatakse igale .c failile vastav eeltöödeldud fail edasi kompilaatorile, mille ülesandeks on koodifailist teha nii-öelda *assembler*-i fail. Antud ülesande edukaks lõpule viimiseks käib töö mitmes faasis: [5] [6]

- Tähthaaval sisse loetud lähtekoodi põhjal tehakse leksiline analüüs, st sisestatud sõnadest ja sümbolitest genereeritakse spetsiaalsed tähised, mis vastavad lähtekoodi keele spetsifikatsioonile. Siin avalduvad vead nagu kaitstud nimede kasutamine muutujate nimedeks, näiteks „int int = 0;“ Seda sammu kutsutakse *parse*-miseks
- Süntaktiline analüüs, mille käigus luuakse tähistest abstraktne süntaksipuu ehk AST (*Abstract Syntax Tree*). Viimane tähistab sisendkoodi keele suhtes agnostilist programmivoo kirjeldust. Siin avalduvad süntaksi vead, näiteks sulgude vale paigutus. Tegemist on viimase sammuga, mille puhul lähtekoodi keel on oluline.
- Semantiline analüüs, mis kujutab endast eelneval sammul genereeritud AST põhjal sisendkeele reeglitele vastavuse kontrolli. Siin avalduvad näiteks tüüpidega tehtud vead, näiteks täisarvu ja sõne liitmine, deklareerimata muutuja kasutamine jne.
- *Intermediate representation*-i ehk vahepealse esituse genereerimine. IR on üldjuhul kompilaatorile spetsiifiline keel, mis on küllalt lähedal masinkoodile. Viimane tuleneb sellest, et tegemist on viimase universaalse sammuga, st sihitud arhitektuuri eripärad ei ole veel olulised.
- IR-i optimeerimine. Viiakse läbi optimisatsioonid, mis ei mõjuta programmeerija kirjeldatud loogikat. Näiteks eemaldatakse siin kasutud operatsioonid, arvutatakse välja lähtekoodis tehetena esitatud konstandid jne.
- Sihitud arhitektuurile vastav optimeerimine. Siin optimeeritakse veel IR kujul olevat koodi selliselt, et järgnevas sammus saaks genereerida sihitud arhitektuuri eripärasid silmas pidav kood. See ja eelmine samm on võrdlemisi häguselt piiritletud.
- Viimase sammuna toimub protsess nimega „langetamine“, mille käigus genereeritakse IR kujul olevast koodist sihitud protsessorile vastav assemblerifail, mis vastab peaaegu üks-üheselt lõplikule binaarkoodile. Puudu on veel vaid lingitud sümbolid. Assembleri fail on veel inimloetav.

Parema ülevaate huvides on Joonis 6 näidatud millised, millises järjekorras ja milliste sisendite-väljunditega on ülal toodud sammud.



Joonis 6. Kompileerimise faasi vahesammud koos sisendite ja väljunditega.

Osakest Clangi poolt genereeritud (erakordselt ebaefektiivsest) vahepealsest esitusest kujutab Joonis 6. Väljundi saamiseks jooksutas autor käsku „clang naidis.c -o naidis_out_llvm_ir_no_opt.txt -S -emit-llvm“.

```
; Function Attrs: noinline nounwind
define i32 @add(i32 %a, i32 %b) #0 {
       entry:
       %b.addr = alloca i32, align 4
       %a.addr = alloca i32, align 4
       store i32 %b, i32* %b.addr, align 4
       store i32 %a, i32* %a.addr, align 4
       %0 = load i32, i32* %a.addr, align 4
       %1 = load i32, i32* %b.addr, align 4
       %add = add nsw i32 %0, %1
       ret i32 %add
}
```

Joonis 7. Add funktsiooni Clangi optimeerimata sisemine esitus.

Pärast optimeerimise läbi viimist saab Joonis 7 kujutatud koodist aga Joonis 8 kujutatud kood:

```
; Function Attrs: norecurse nounwind readnone
define i32 @add(i32 %a, i32 %b) local_unnamed_addr #4 {
entry:
  %add = add nsw i32 %b, %a
  ret i32 %add
}
```

Joonis 8. Add funktsiooni efektiivsem esitus.

GCC kasutab sisemiseks ideaalse masina käsustiku kujutamiseks aga GIMPLE-nimelist keelt, mis on toodud võrdluseks LLVM-i IR-ga. Saadud käsuga „gcc naidis.c -o naidis_out_gcc_ir.txt -S -fdump-tree-gimple-raw“, kujutatud Joonis 9.

```
add (int a, int b)
gimple_bind <
  int D.2374;
  gimple_assign <plus_expr, D.2374, a, b, NULL>
  gimple_return <D.2374 NULL>
>
```

Joonis 9. GCC GIMPLE esituses Add funktsioon.

Selleks, et nii-öelda langetamist läbi viia, on eelnevalt vajalik, et eksisteeriks sihitud arhitektuuri käsustikule ja võimekusele vastav masina kirjeldusfail. GCC kasutab selleks puhuks .md laiendiga *Machine Description* faili, LLVM aga .td laiendiga *Target Description* faili. Vastavates failides on kirjeldatud kõik operatsioonid, mida sihitud arhitektuur teha oskab ning nendega kaasnevad eripärad, registrite arv, nimed, kirjeldused jne. *Target Description*-st on natuke lähemalt juttu ümber suunamise peatükis.

Antud operatsioonide põhjal viiaksegi läbi lõplik assembleri teisendus. Joonis 10 on toodud Clangi genereeritud masinkood funktsioonid „Add“.

```
_add:                                   # @add
# BB#0:                                 # %entry
        pushl   %ebp
        movl    %esp, %ebp
        subl    $8, %esp
        movl    12(%ebp), %eax
        movl    8(%ebp), %ecx
        movl    %eax, -4(%ebp)
        movl    %ecx, -8(%ebp)
        movl    -8(%ebp), %eax
        addl    -4(%ebp), %eax
        addl    $8, %esp
        popl    %ebp
        retl
```

Joonis 10. Clangi poolt genereeritud inimloetav masinkood.

3.4 Assembler

Assembleri ülesanne on võtta eelneva sammu väljund ning see masinkoodi teisendada. Teisenduse väljundiks on nö objektifailid, tihtipeale .o laiendiga. Saadud failis ei ole veel kõike vajalikku jooksutatava binaarkoodi saamiseks – puudu on veel nö linkimise sümbolid, näiteks viidad funktsioonidele, mis asuvad välistes teekides. Samuti on veel tegemata konstantide ja nihete arvutamine, failisiseste sümbolinimede asendamine neile vastavate aadressidega ja mäluaadresside teisendamine. Assembleri ülesandeks on lisaks instruktsioonide ja operandide teisendamisele just viimati loetletud asjade tegemine. Väliselt lingitud sümbolite asendamisega tegeletakse viimases ehk nö sidumise faasis.

3.5 Sidumine

Viimase sammuna koostab *linker* talle antud objektifailidest ja väliselt lingitud sümbolitest ühe suure *binary* ehk binaarkoodist koosneva faili. Linkeri ülesandeks on ka mälusektsioonide paigutamine mällu ning lõplik konstantide arvutamine. Assembleri väljundis paigutatakse muutujad, konstandid ja operatsioonid tihti kindlale mäluaadressile, mis võib kattuda teistes objektifailides kirjeldatud väljadega. Linkeri ülesanne on neile tinglikult öeldes uus koht leida.

Linkeri mälupaigutuse muutmiseks kasutatakse linkeri skripte, mis kirjeldavad kuhu millised sektsioonid panna tuleb, näiteks asuvad globaalsed muutujad ja sõned, mida koodi jooksutamise käigus ei muudeta, tihti spetsiifilistes kohtades. Mikrokontrollerite puhul võib seal olla kirjeldatud ka näiteks *heapi* ehk kuhja asukoht ja pinu alguse aadress.

Linkeri töö lõppedes on saadav binaarkood sihitud masina peal otse jooksutatav.

4 Kompilaatori kandidaadid

Autori lõpliku eesmärgi täitmiseks on vajalik langetada otsus ümber suunatava kompilaatori osas. Valik ei olnud esialgu lihtne - Wikipedia kompilaatorite lehekülg [7] andis uurimistöö läbiviimise hetkel levinud C kompilaatorite arvuks 64. Kui neid aga lähemalt uurima hakata selgub, et enamus neist ei ole antud rakenduse jaoks tegelikult kõlblikud. Pärast modifitseeritavuse, litsentsi, kättesaadavus, efektiivsuse ja suutlikkuse tõttu ebasobivate kandidaatide välja arvamist jäid lauale vaid üksikud. Viimastest said suurema soosingu osaliseks need, millel on sarnasele arhitektuurile (näiteks AVR) juba *backend*-d olemas, kuna nende lähtekoodi ja tööpõhimõtte uurimine võib osutuda äärmiselt kasulikuks. Viimasele kriteeriumile vastavad kaks kompilaatorit – LLVM ja GCC.

Valik LLVM ja GCC vahel ei osutunud enam kuigi lihtsaks kuna mõlemad kandidaadid on pidanud vastu aja survele ja on seeläbi muutunud vägagi usaldusväärseteks ja hästi toetatud projektideks. Valiku tegemiseks tuli objektiivselt võrrelda mõlema projekti häid ja halbu külgi ning eraldi hinnata nende sobivust eesmärgi täitmiseks. Väga oluliseks osutus ka autori subjektiivne vaade dokumentatsiooni olukorrale ja kättesaadavusele, olemasolevate sarnaste joontega protsessoriarhitektuuride tugi ning nii-öelda ökosüsteemi terviklikkus.

Järgnevalt vaadeldakse nende kompilaatorite häid ja halbu külgi ning tuuakse võrdluseks kummagi tugevad ja nõrgad aspektid. Analüüsi tulemusel selgub, miks autor lõpuks LLVM-i infrastruktuuri kasuks otsustas.

4.1 Ülevaade

Et paremini mõista, mida GCC ja LLVM projektid endast täpsemalt kujutavad ja kust nad alguse said, annab autor neist lühikese ülevaate.

4.1.1 GCC

GCC on tõenäoliselt tuntuim kompilaator maailmas. GCC esimene versioon nägi ilmavalgust 1987. aastal. GCC sai alguse vajadusest kompilaatori järele GNU (*GNU's Not Unix*) projekti raames. Vaba tarkvara liikumise suurkuju Richard Stallman hakkas esialgu uurima erinevaid kompilaatoreid, mida GNU tarbeks kasutada kuid järeldas lõpuks, et kõige otstarbekam on kirjutada projekti vajadustele ja võimalustele vastav kompilaator. Nii lasigi Stallman 22. märtsil 1987 välja esimese versiooni GCC-st, mille kaasautoriteks sai mainitud veel mitmeid inimesi. [8]

GCC-d kirjeldatakse kui esimest tõeliselt edukat vaba tarkvara. Esimese versiooni väljalase kattus Sun Microsystemsi otsusega hakata oma tarkvara arenduse tööriistade komplekti küllalt kallilt müüma ning viis nii mõnedki kasutajad tasuta samalaadset tarkvara kasutama. Vaid kolme aastaga oli GCC-d laiendatud töötama 13 erineva arvutiarhitektuuri peal, kusjuures töötades tihti paremini kui tasulised alternatiivid.

Tänu GNU projektiga tuntuks saanud GPL (*GNU General Public License*) litsentsile võis igaüks võtta GCC lähtekoodi ning seda modifitseerida vastavalt enda vajadustele, eeldusel et kinni peetakse GPL-i tingimustest. Neist olulisim on nõue, et nii-öelda uus *fork* ehk haru peab ka olema lihtsasti kättesaadava lähtekoodiga ning kasutama sama litsentsi.

Tänapäeval on GCC tõenäoliselt enim kasutatav kompilaator, GCC dokumentatsioon [9] ütleb, et seda saab kasutada 48-le erinevale arhitektuurile binaarkoodi genereerimiseks. GCC jookseb sisuliselt iga laiatarbe operatsioonisüsteemi peal, lisaks paljudele vähem tuntutele. GCC on enamike Linuxi distributsioonide standardkompilaator ning (2010. a seisuga) oli ainuke kompilaator, millega oli võimalik tervet Linuxi kernelit kompileerida.

4.1.2 LLVM – clang

LLVM projekt on kollektsioon modulaarsetest ja hõlpsasti kasutatavatest kompilaatori ja tööriistaahela tehnoloogiatest. Kuigi lühend võib viidata LLVM-le kui virtuaalmasinale on tegelikkus sellest kaugel. Viimane ei tähenda aga kindlasti seda, et LLVM-i põhjal ei saaks virtuaalmasinat teha. Projekti skoobi muutudes kaotas esialgu akronüümina alguse saanud nimi ka oma esialgse tähenduse, tänapäeval ongi „LLVM“ projekti nimi ning selles terendavad tähed ei oma sügavamat tähendust.

LLVM sai 2000. aastal alguse Vikram Adve ja Chris Lattneri uurimistööna Illinois Ülikoolis. Uurimistöö eesmärk oli põhjalikumalt uurida staatiliste ja dünaamiliste programmeerimiskeelte dünaamilist kompileerimist. Töö käigus arendatud tarkvarast sai alguse laiaulatuslik tööriistade ja teekide komplekt, mida kasutatakse nii kommerts, vabavara kui ka akadeemilistel otstarvetel [10].

2005. aastal Apple-sse palgatud Chris Lattner sai ülesandeks arendada firmas kasutatavaid tööriistu ning SDK-d (*Software Development Kit*). Lattner valis platvormiks talle enam kui tuttava LLVM projekti, mida kavatses esialgu kasutada koos GCC Objective-C keele *frontend*-ga. Selgus aga, et viimase arendamine ei olnud GCC meeskonnale kuigi suur prioriteet, ei olnud kuigi hästi laiendatav ega integreerunud kuigi hästi Apple-i XCode tarkvaraga. Seetõttu otsustas Apple alustada täiesti uue C, Obj-C ja C++ keelte *frontend*-i arendamist. Nii sündiski LLVM-i projektiga väga tihedalt seotud Clang, mille lähtekood muudeti avalikult kättesaadavaks 2007. aastal [10].

LLVM hõlmab endas arvukaid alamprojekte, millest autorile pakuvad huvi eelkõige:

- LLVM Core - eeskätt kogum teeke, mille eesmärgiks on pakkuda modernset lähtekeelest ja sihitud arhitektuurist iseseisvat optimeerijat ning tuge arvukatele protsessoriarhitektuuridele koodi genereerimiseks. LLVM Core kasutab sisemiselt ülal tutvustatud IR-i koodi kujutamiseks.
- Clang – C keele *frontend* ehk tarkvara, mis genereerib C-keele perekonna programmeerimistkeeltest IR-i
- lldb – arendusjärgus olev *debugger*
- lld - linker

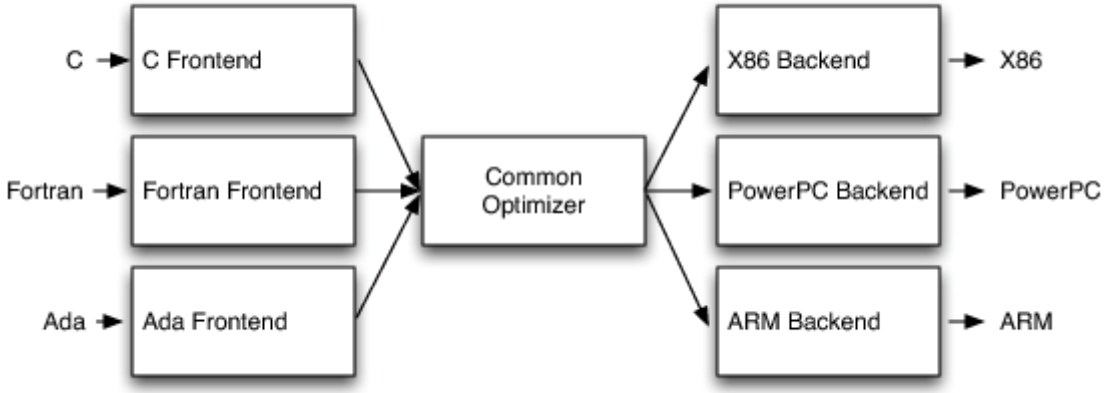
Kuigi LLVM on GCC-st 15 aastat noorem on tegemist siiski igati tõsiseltvõetava ja küpse projektiga, millel on kasutajale omajagu pakkuda. Viimast kinnitab asjaolu, et LLVM ja Clang leiavad kasutust lisaks Apple-i ökosüsteemile veel ka näiteks Sony PS4 SDK-s.

4.2 GCC ja LLVM-i võrdlus

4.2.1 Struktuur

Kummagi projekti struktuur on suuresti määratud tema ajalooga – GCC kui väga spetsiifilisse raamistikku disainitud ja arendatud tarkvara on kuni viimase paari aastani olnud võrdlemisi monoliitne tükk. Sellega vastandub LLVM, mis on aja jooksul suutnud hoida talle omast modulaarset joont.

Täpsemalt iseloomustab LLVMi ülesehitust Joonis 11 kujutatud skeem.



Joonis 11. LLVM struktuur [10].

„C Frontend“ kohal terendab lugejale tuttav Clang, „Common Optimizer“ on ulatuslik IR-i peal töötav LLVMi optimeerija ning „Backendiks“ saab autori töö tulemusel PISKE protsessorile sobiv *backend.* Joonise pealt tuleneb ka LLVM-i kiiresti tõusnud menu põhjus. Soovi korral saab kas ühe või mitu alamprojekti võtta ning need suhteliselt vaevata mõne muu projektiga siduda – nii on Clang leidnud kasutust nii mõneski arenduskeskkonnas *linter*-i ja koodilõpetajana.

Sellele vastandub väga tugevalt GCC, mis on ajalooliselt olnud pigem üks suur tükk tarkvara kui LLVM-le sarnase ehitusega kogumik. Tänu LLVMi konkurentsile on ka GCC liikunud suurema modulaarsuse suunas. Paraku on tegemist veel poolelioleva tööga, kuid GCC v7.1 dokumentatsioon annab lootust, et olukord on kiiresti paranemas.

4.2.2 IR võrdlus

LLVM projekt on algusest peale kasutanud nende enda kodukootud vahepealset esitust, mida saab kujutada nii teksti kui baidijadana. Viimased on omavahel ekvivalentsed kõiges muus kui esitusviisis. Selline omadus lubab kompilaatoriarendajal hõlpsasti genereeritavas koodis vigu leida ning samal ajal tagab oluliselt suurema arvutusliku töötlemisvõimekuse, kui seda lubaks vaid esitus teksti kujul. LLVM-i IR on väga selgelt defineeritud reeglitega keel, mille kohta on lihtne leida dokumentatsiooni [11] (ja ka kiidusõnu [12]).

GCC seevastu kannab endaga kaasas aastakümneid tagasi tehtud otsuseid. Üle erinevate versioonide ja alaliikide on GCC-d õpetatud töötama RTL-s, GENERIC-s, GIMPLE-s, SPIR-s, LLVM IR-s ja HSA *Intermediate Layer*-s. Neist viimases laiatarbe versioonis leiavad kasutust RTL, GENERIC ja GIMPLE. Segaduse tekitamiseks kasutavad erinevad GCC *front end*-d kas GENERIC-t, GIMPLE-t või mõlemat. GIMPLE on küllalt sarnane LLVM IR-ga ja põhineb laias laastus samadel printsiipidel. GENERIC seevastu on matemaatilist puud kirjeldav keel. C keelt kompileerides genereerib GCC alguses lähtekoodi esituse GENERIC-s, seejärel muudetakse ta GIMPLE-ks, millest omakorda saab RTL. Viimaks saab langetamise teel RTL-st assembler.

Kuigi LLVM-i IR ja GCC GIMPLE on küllalt sarnased keeled nii süntaksi kui lähenemise poolest, jääb siin peale LLVM-i oluliselt standardiseeritum vahepealne esitus.

4.2.3 Laiendatavus

LLVM on programmeerimiskeeles C++ kirjutatud ja juba algusest peale ehitatud väga modulaarseks tööriistaks. Tänu sellele on võimalik sama arhitektuuriga, kuid väikeste erinevustega protsessoreid hõlpsasti lisada ja muuta. Nii saaks PISKE puhul luua erinevaid kompilaatori konfiguratsioone, mis vastaksid ühele või teisele PISKE konfiguratsioonile, näiteks juhul kui vaja peaks minema ujukomatehete riistvaralist tuge või kui registrite asukoht, võimekus jne peaksid muutuma. Niinimetatud *subtarget*-te loomine GCC peal on oluliselt tülikam protsess.

LLVM projekt sai valitud ühe teise vägagi huvitava projekti – Rust programmeerimiskeele - kompilaatoriks. [13] Rust on võrdlemisi uus programmeeriskeel, mis toob endaga kaasa selliseid omadusi nagu jooksmise ajal „tasuta“ abstraktsioonid, garanteeritud mäluturvalisus, ohutu mitmelõimelisus ja peaaegu olematu *runtime.* Just need omadused teevad temast selliste piiratud ressurssidega platvormide peal nagu seda on PISKE ideaalse programmeerimiskeele. Tänu LLVM-i modulaarsele ehitusele on võimalik vaid *backend*-i ümber kirjutades tuua Rusti tugi ka PISKE protsessorile, nagu seda tehakse praegu sarnase AVR arhitektuuriga [14]. Rust on praegu veel võrdlemisi noor keel, mida ei ole kuigi palju sellistes rakendustes kasutatud, kuid siiamaani tundub, et aja jooksul hakatakse Rusti tõsisemalt võtma ka muude platvormide peal kui x86.

4.2.4 Muud tööriistad

Kompilaator võib olla küll C-keeles programmeerimise jaoks kõige olulisem tööriist, kuid märkimata ei saa jätta ka näiteks *linker*-t ja *debugger*-t. Nii GNU kui ka LLVM-i projektid hõlmavad endas lisaks kompilaatorile ka muid vajalikke tööriistu. Nii on GNU projektis LLVM-i lld-le vastavad *linker*-d nimega ld ja gold, lldb-le vastav *debugger* gdb jne. Oma baasfunktsionaalsuse poolest on need tööriistad võrdväärsed.

Linkerid gold, ld ja lld täidavad kompileerimise protsessi viimast sammu ehk sidumise jaoks vajalikku funktsiooni. lld lubab olla oluliselt kiirem kui ld [15], mis nagu muud GNU projekti osad, kannab endaga kaasas aastakümneid ajalugu. gold seevastu on võrdlemisi uus liige GNU perekonnas ja on võimeline ka lld-ga sammu pidama [16]. lld suur eelis üle teiste alternatiivide on võime töötada LLVM-i *bitcode*-ga, lubades seeläbi ka sidumiseaegset optimeerimist ehk LTO-d (*Link Time Optimisation*). Vastukaaluks – lld-d võib tootmiskõlbulikuks ja küpseks pidada vaid väga levinud platvormidel ning võib uue sihitud platvormi peal endaga ootamatuid probleeme kaasa tuua.

Viimane on probleem ka lldb-ga, mis kirjutamise hetkel ei oma openOCD (*On-Chip Debugger*) tuge ja seega ei ole sobilik *cross-platform debugging*-u jaoks. gdb-d seevastu on koos openOCD projektiga töötanud juba aastaid ja on ilmselt üks enim levinud lahendusi sellist sorti silumiseks ning lubab kasutada väga suurt hulka erinevaid riistvara *debugger*-d.

Laiema ökosüsteemi mõistes on GNU tööriistad kindlasti paremini juurdunud ja toetatud ning seetõttu oluliselt kasutatavamad.

4.2.5 Kokkuvõttev tabel

Tabel 1. GCC ja LLVM võrdlus.

Omadus	**GCC (GNU)**	**LLVM / Clang**
Esimene väljalase (aasta)	1987	2003
Programmeerimiskeel	C, C++	C++
Litsents	GPLv3	University of Illinois / NCSA Open Source License
Toetatud keeled (*frontend*)	C, C++, Objective-C, Fortran, Ada ja Go	ActionScript, Ada, C#, Common Lisp, Crystal, D, Delphi, Fortran, OpenGL Shading Language, Halide, Haskell, Java bytecode, Julia, Lua, Objective-C, Pony, Python, R, Ruby, Rust, CUDA, Scala, Swift ja Xojo
Vahepealne esitus	RTL, GENERIC, GIMPLE	LLVM IR
Toetatud arhitektuurid (*target, backend*)	ARM, AVR, lm32, M68k, MicroBlaze, Mips, MSP430, PDP11, RISCV, Sparc, xtensa, x86 (kokku 48)	AMDGPU, ARM, AVR, MSP430, Mips, PowerPC, RISCV, Sparc, WebAssembly, x86 (kokku 17)
Toetatud C keele standardid	C89, C99, C11	C89, C99, C11
C keele laiendused	Arvukalt laiendusi, paindlik	Minimaalselt laiendusi, range
Debugger	gdb	lldb
Linker	ld, gold	lld
Debuggeri openOCD tugi	Olemas	Puudu

5 Portimine

LLVM-s uuele arhitektuurile binaarkoodi genereerimise kulminatsioon oleks *backend*, mis Joonis 4 tuttava *Add* funktsioonist genereeriks Joonis 12 kujutatud assemblerikoodi ning sellest seejärel *linker*-t kasutades sellest omakorda binaarkoodi genereeriks.

```
; Liidame 1 + 2, ja omistame tulemuse ax-i
       mov ax, 1
       mov [bp-2], ax      ; Laeb esimese operandi alamfunktsiooni pinusse
       mov ax, 2
       mov [bp-3], ax      ; Laeb teise operandi alamfunktsiooni pinusse
       call funktsioon_add ; Kutsub välja funktsiooni
       mov ax, [bp-4]      ; Loeb välja alamprogrammi return väärtuse

funktsioon_add:
       mov ax, 5           ; Stack frame-i suurus
       sub bp, ax          ; Vähendab bp-d stack frame-i suuruse võrra
       mov [bp-1], ax      ; Salvestab stack frame-i suuruse pinusse
       mov ax, [bp+3]      ; Argument 1
       add ax, [bp+2]      ; Argument 2
       mov [bp+1], ax      ; Tulemus
       add bp, [bp-1]      ; Suurendab bp stack frame suuruse võrra
       ret                 ; Pöördub tagasi [bp+0]
```

Joonis 12. Add funktsioon PISKE assembleris.

Joonis 12 on kujutatud soovitud assembleri kood, kus terendavad lisaks hädavajalikele liigutamisele ja liitmistehetele ka alamfunktsioonide sisenemise ja väljumise kood. Viimaste ülesandeks on allokeerida funktsioonile vajaminev ruum mälus ning veenduda, et viidaregister (*bp*) näitaks õigesse kohta, kuna muutujate adresseerimine toimub enamikel juhtudel selle suhtes. Väljumise korral vastutatakse viidaregistri eelneva väärtuse taastamise eest.

LLVM-i dokumentatsiooni [17] põhjal on uue *backend*-i implementeerimiseks vajalikud vaid üksikud küllalt töömahukad sammud.

Nendeks on:

- *TargetMachine* alamklassi loomine, antud juhul failides nimega „PiskeTargetMachine.cpp“ ja „PiskeTargetMachine.h“. Selles alamklassis on kirjeldatud sihitud arhitektuuri karakteristikud, mida tutvustatakse ülejäänud kompilaatori koodile.
- Kirjeldada protsessori registreid. „RegisterInfo.td“ failis peavad olema *Target Description* tüüpi keeles täpselt märgitud registrite definitsioonid, aliased ja klassid. Neist genereeritakse *TableGen*-nimelist tööriista kasutades kood, mis vastab .td failis kirjeldatule.
- *TargetRegisterClass* alamklassis tuleb kirjeldada erinevate registrite interaktsioonid ja nende allokeerimise eeskiri.
- „TargetInstrFormats.td“ ja „TargetInstrInfo.td“ failides kirjeldada sihitud arhitektuurile tuttavad instruktsioonid ja implementeerida *TargetInstrInfo* alamklassis nende LLVM-ga liidestamiseks vajalikud ülejäänud funktsioonid.
- Kirjeldada mille alusel ja milliseid instruktsioone kasutada LLVM IR-i muundamiseks sihitud arhitektuurile vastavaks. Taaskord *TableGen* tööriista kasutades genereerida kood, mis vastavalt ette antud mustritele ja muu sihitud arhitektuurile relevantset informatsiooni (kirjeldatud „TargetInstrInfo.td“-s) kasutades valib spetsiifilisi instruktsioone. Täpsustav informatsioon kirjeldada failides „PISKEISelDAGToDag.cpp“ ja „PISKEISelLowering.cpp“-s vastavaid alamklasse implementeerides.
- Kirjutada kood *assembly printer*-i jaoks, mis muundab LLVM IR-i soovitud arhitektuuri GAS või llvm-as formaati, implementeerides *AsmPrinter*-i ja *TargetAsmInfo* alamklassid. Kasutab „TargetInstrInfo.td“-s täpsustatud sõnesid assembleri käskude jaoks.
- Lisada nö *subtarget* tugi, mis lubab väikeste erinevustega protsessorite eripärasid arvestada vastavalt valitud *subtarget*-le. Kasulik näiteks juhul kui mingitel PISKE variantidel ei ole mõni käsk toetatud või on lisakäske.

6 Kokkuvõte

PISKE protsessorile C-keelest binaarkoodi kompileerimiseks on mitu võimalust. Ühena neist võiks täiesti algusest alustada ning kirjutada terve kompilaatori *toolchain*-i. Ent sellise tööriista võimekus oleks tõenäoliselt naeruväärne ilma massiivse ajainvesteeringuta. Oluliselt parem lahendus on vaadata laiemalt levinud kompilaatorite poole ning hinnata neid vastavalt PISKE poolt seatud kriteeriumitele. Filtreerimise tulemusel järeldub, et kõige paremini sobivad LLVM ja GCC projektid.

Neist perspektiivikamaks osutus LLVM projekt. Viimase põhjal arendatud clang kompilaatori mugandamine uuele arhitektuurile võiks anda PISKE kitsendusi ja LLVM konkurente arvesse võttes parima tulemuse. LLVM-i kasuks räägivad väga hea optimeerija, hea vahepealne koodiesitus ja uute rakenduste jaoks laiendamist soosiv struktuur. Veel võib ühe hea küljena välja tuua LLVM-i dokumentatsiooni, mis on tunduvalt kompaktsem ja paremini navigeeritavam kui alternatiivide oma.

LLVM-i põhjal töö jätkamine võib aga tulevikus paratamatult tähendada seda, et soovi korral mõni *debugger* PISKE-t toetama panna tähendab väikses mahus sama töö kordamist uues vormis. Arvestades lldb projekti hetkeseisu tuleb *debugger*-i jaoks tõenäoliselt hakata vaatama juba GNU projekti gdb poole, mis kirjutamise hetkel oli ainuke tõsiseltvõetav variant.

Portimise protsess on LLVM projekti puhul võrdlemisi hästi kaardistatud koos arvukate näidetega, millest on võimalik vajadusel inspiratsiooni ja näiteid leida. LLVM-i peal on ka AVR arhitektuuri, millel on PISKE-ga sarnaseid jooni, toetav *backend* olemas, mis võib samuti kasulikuks osutuda.

7 Kasutatud kirjandus

[1] C. Walls, „Embedded Systems Programming Languages,“ AspenCore Media, 12 September 2014. [Võrgumaterjal]. Saadaval: http://www.eetimes.com/author.asp?doc_id=1323907.

[2] International Organization for Standardization, „ISO/IEC 9899:2011,“ Detsember 2011. [Võrgumaterjal]. Saadaval: https://www.iso.org/standard/57853.html.

[3] Atmel Corporation, „AVR Libc Reference Manual,“ [Võrgumaterjal]. Saadaval: http://www.atmel.com/webdoc/avrlibcreferencemanual/FAQ_1faq_reg_usage.html.

[4] G. Project, „The C Preprocess: Preprocessor Output,“ [Võrgumaterjal]. Saadaval: https://gcc.gnu.org/onlinedocs/cpp/Preprocessor-Output.html.

[5] H. E. B. C. J. H. J. K. G. L. Dick Grune, Modern Compiler Design, Chichester: John Wiley & Sons, Ltd, 2000.

[6] A. W. Appel, Modern Compiler Implementation in C, Cambridge: Cambridge University Press, 2004.

[7] Wikipedia, „List of compilers,“ Wikimedia Foundation, [Võrgumaterjal]. Saadaval: https://en.wikipedia.org/wiki/List_of_compilers#C_compilers. [Kasutatud 11 May 2017].

[8] G. C. Collection, „A Brief History of GCC,“ [Võrgumaterjal]. Saadaval: https://gcc.gnu.org/wiki/History.

[9] GCC, „Status of Supported Architectures from Maintainers' Point of View,“ FSF, [Võrgumaterjal]. Saadaval: https://gcc.gnu.org/backends.html. [Kasutatud 7 May 2017].

[10] C. Lattner, „The Design of LLVM,“ LLVM Project, [Võrgumaterjal]. Saadaval: http://www.drdobbs.com/architecture-and-design/the-design-of-llvm/240001128.

[11] L. Project, „LLVM Language Reference Manual,“ [Võrgumaterjal]. Saadaval: http://llvm.org/docs/LangRef.html.

[12] majek04, „IR is better than assembly,“ [Võrgumaterjal]. Saadaval: https://idea.popcount.org/2013-07-24-ir-is-better-than-assembly/.

[13] R. P. Developers, „Rust FAQ,“ Rust Project Developers, [Võrgumaterjal]. Saadaval: https://www.rust-lang.org/en-US/faq.html#how-fast-is-rust.

[14] avr-rust, „avr-rust repository,“ [Võrgumaterjal]. Saadaval: https://github.com/avr-rust/rust.

[15] l. projekt, „LLD - The LLVM Linker,“ LLVM Projekt, [Võrgumaterjal]. Saadaval: https://lld.llvm.org/#performance.

[16] M. Larabel, „Trying Out LLVM 4.0's LLD Linker On Ubuntu 17.04 vs. GNU LD, GNU Gold,“ phoronix, 16 Märts 2017. [Võrgumaterjal]. Saadaval: http://www.phoronix.com/scan.php?page=article&item=lld4-linux-tests&num=1.

[17] L. Projekt, „Writing an LLVM Backend,“ LLVM Project, [Võrgumaterjal]. Saadaval: http://llvm.org/docs/WritingAnLLVMBackend.html.

[18] L. S. Sterling, The Art of Agent-Oriented Modeling, London: The MIT Press, 2009.
[19] GNU Project, „The C Preprocessor,“ [Võrgumaterjal]. Saadaval: https://gcc.gnu.org/onlinedocs/gcc-2.95.3/cpp_1.html.

Lisa 1 – PISKE 16-bitine mikrokontroller

Järgnevalt lehelt algab PISKE tooteleht.

PISKE 16bitine Mikrokontroller

Ülevaade

Peamised omadused

- Maailma väikseim kompilaatorisõbralik 16bit „softcore“ protsessor: 260 LUT (4 sisendiga)
- Puhas Harvardi arhitektuur:
  - Sõltumatud koodi, andmete ja perifeeriasiinid
  - Kõik siinid toetavad 2^16 x 16bitist aadressiruumi
  - Andmete ja koodimälu mälu suurus on enamasti piiratud FPGA vaba sisemälu ressursiga
  - Protsessori käskudega koodimälu lugeda ja kirjutada ei saa
- Minimalistlik RISC arhitektuur:
  - 49 käsku, kõik täidetakse 1 taktsageduse tsükliga
  - Protsessor kasutab nii taktsageduse tõusvat kui langevat fronti
  - Kolm baasregistrit: pc, ax ja bp
  - Kaks lippu: Z ja C
  - Ühe argumendiga aritmeetikakäsud (nihked) saavad kasutada vaid akumulaatorit
  - Kahe argumendiga käskude üks argument on akumulaator, teine argument on mälupesa. Tulemus salvestatakse akumulaatorisse või mälupessa.
  - Mäluargumente saab adresseerida nii otse kui ka relatiivselt bp registri suhtes:
    - Otse saab adresseerida esimest 512 sõna mälust
    - Osa otseadresseeritavaid mälupesi saab kasutada sobiva mahuga „registrite“ salvena kuna aritmeetikakäsud võimaldavad 1 tsükliga „read-modify-write“.
    - Suhteliselt saab adresseerida bp-256 kuni bp+255 aadressruumi
    - Suhteline adresseerimine võimaldab täita nii pinu, funktsioonide lokaalmuutujate kui funktsioonide argumentide salve vajadused
  - „Return aadress“ salvestatakse automaatselt bp indeksregistri järgi väljakutsuva funktsiooni lokaalmuutujana
- Minimaalne mälu- ja perifeeriapöördumine: 16bit
- Arendatud koos kokkusobivate minimalistlike perifeeriamoodulitega perekonnana
- Katkestuste kontroller:
  - 1-16 sisendit
  - Katkestuste kooperatiivne („non-preemptive“) teenindamine
  - 11 takti (<0.5us) sisenemislatents katkestuse signaalist kõrgeima prioriteediga katkestust täitva koodini, millest 10 kulub konteksti salvestamiseks ja aktiivse katkestuse valimiseks tarkvaralise katketuste halduri poolt
  - 9 takti katkestushalduri väljumislatents mis kulub katkestuste „tail-chaining“-u teostamisele ja konteksti taastamisele
- Protsessor toetab mitut energia kokkuhoiu taset:
  - Taktsageduse kontroller toetab kaht taktsagedust ja nende tõrgeteta ümberlülitamist
  - Katkestuskontroller võimaldab protsessorit seisata esimese saabuva katkestuseni
- Implementatsioon: Verilog HDL

- Optimeeritud madala voolutarbe ja väikese mahuga FPGA platvormidele:
  - Protsessor koos kella- ja katkestuskontrolleriga vajab <300 LUT
  - Aeglaseimate, Lattice iCE40UL FPGA-dega on saavutatav 24MHz taktsagedus. Teistel FPGA platvormidel saavutatavad taktsagedused on tunduvalt kõrgemad.

Sissejuhatus

Viimastel aastatel on turule tulnud mitmeid väikese mahuga, odavaid, eriti väikese staatilise voolutarbega ja füüsiliste mõõtmetega FPGA-sid nagu Lattice/SiliconBlue iCE40, Lattice MachXO3L ja Altera Max10.

PISKE mikrokontroller sai välja töötatud võimaldamaks ka selliste toodete kasutamist System-On-Chip filosoofia järgi: lisades mikrokontrolleri tuumale tema jaoks väljatöötatud standardseid perifeeriakomponente ning konkreetse rakenduse tarbeks arendatud HDL lahendusi. Seni nõudis selline lähenemine oluliselt suuremate ja rohkem energiat tarbivate FPGA-toodete kasutamist.

Pakutakse tervet klassi väiksemaid FPGA-sid mahuga 640-8000 4-sisendilist LUT-i. Saada olevate korpuse mõõdud algavad 1.4x1.4mm-st ja hinnad $1.2-st ka väiksemate koguste juures. Sellise suurusklassi FPGA-de tarbeks on turul olemas olevate 32bitiste protsessorite (Altera Nios-II, LatticeMico32, Xilinx MicroBlaze) kasutamine ebapraktiline. Kui neid ka oleks võimalik selliste FPGA-de sisse sünteesida, siis haaraksid nad enamiku nende FPGA-de mahust mitte jättes ruumi kasutaja lisatavate lahenduste ja perifeeria jaoks.

Samas on väiksema mahu jaoks optimeeritud sünteesitavate 8-bit mikrokontrollerite (LatticeMico8, Xilinx PicoBlaze etc) võimekus väga madal. Vabavaraline alternatiiv OpenMSP430 pole parem: vajab rohkem ressurssi kui Altera Nios-II ja Xilinx MicroBlaze (tüüpiliselt >2000 LUT), pakkudes samas nimetatutest oluliselt madalamat võimekust.

Sellest tulenevalt sai välja töötatud uus 16bit ISA ja sünteesitav mikrokontrolleri arhitektuur. Eesmärgiks oli saavutada väikseim võimalik ressursivajadus millega on veel võimalik teostada selline ISA (Instruction Set Architecture) mille jaoks on võimalik mõistlikult mugandada mõni C kompilaator. Eesmärgiks on samas tavapärastest 8bit mikrokontrolleritest oluliselt laiemate võimalustega ja võimekam protsessor.

Protsessori juurde on välja töötatud ka perekond sobivaid perifeeriakomponente:

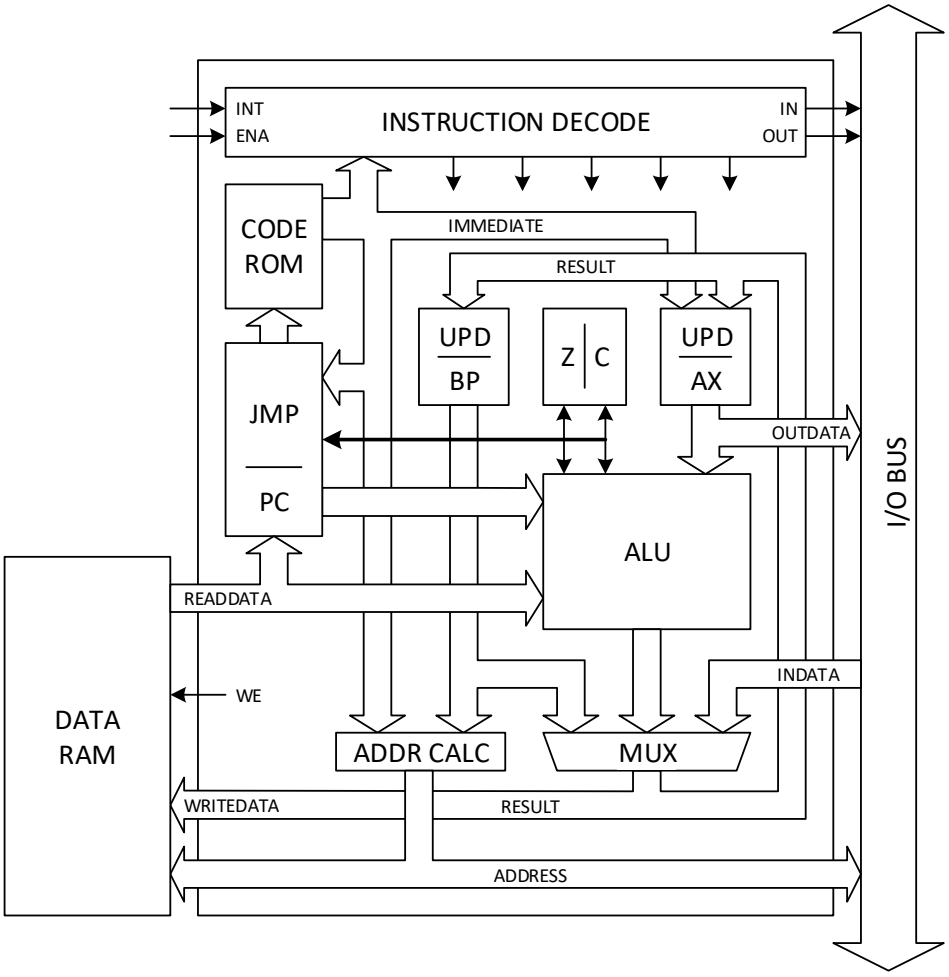
- 3-pordine põhimälu (protsessor, DMA ja USB/bus-master)
- Kellakontroller
- Katkestuste kontroller
- GPIO kontroller
- Taimer
- Striimide I/O DMA kontroller
- Striimide FIFO ja serialiseerimise liides
- I2S konverter FIFO/serial I/O liidesele
- USB LS/FS/HS funktsioonikontroller (bulk, interrupt ja isokroonsed lõpp-punktid)

Kõigi perifeeriakomponentide jaoks on arendatud ka minimalistlikud draiverid ja assembleris teostatud näidisrakendus mis neid kasutab. Näidisprojekt mis koosneb kõigest ülalnimetatust mahub parasjagu töötama 1280 LUT mahulisse Lattice ICE40-LP FPGA-sse ning on täielikult USB standarditele vastav.

Protsessorit arendades oli eesmärgiks saavutada ca 1 DMIPS/MHz jõudlus ning ca 24MHz taktsageduse võimekus Latice iCE40-LP ja UL seeria FPGA-l. Kui suurt jõudlust PISKE tegelikult võimaldab on hetkel veel kontrollimata kuna C-kompilaatori tugi on alles arendamisel.

Tööriistadest on hetkel olemas Python-is realiseeritud assembler ning hierarhiline netlisti analüsaator mis näitab milliste süsteemi komponentide jaoks kui palju ja milliseid FPGA ressursse kulub. Süsteemisisest debuggerit ega tarkvara simulaatorit hetkel välja arendatud ei ole. Terviksüsteemi saab aga simuleerida FPGA simulaatoritega ning selle tarbeks on teostatud põhjalik Verilog testbench mis muuhulgas analüüsib kogu USB enumereerimisprotsessi ning peamisi perifeeria poolt võimaldatavaid funktsioone. Selliste testbenchidega saab ja on edukalt simuleeritud ka protsessori tööd reaalse koodiga.

Arhitektuur



PISKE keskmeks on aritmeetika-loogika seade mis saab enamasti ühe argumendi mälust ja teise akumulaatorist. Ühe argumendiga loogikatehted (nihked) kasutavad argumendina vaid akumulaatorit. Lisafunktsioonina kasutatakse aritmeetika-loogikaseadet alamprogrammide väljakutsumise ajal jooksva käsu aadressile 1 otsaliitmiseks mille tulemus siis mällu salvestatakse.

Selleks et FPGA ressursse võimalikult optimaalselt ära kasutada on osad aritmeetika ja loogikafunktsioonid viidud aritmeetika-loogika seadmest välja, sisemiste registrite (ax, bp ja pc) juurde. Käsud ja sündmused mis neid registreid muudavad (näiteks add bp,<arg>) teostatakse tegelikult nende registrite juurde kuuluva loogika abil. Selle illustreerimiseks on plokkskeemis põhiregistrite juures lühend „UPDate“.

Käsustiku dekodeerimise loogika on PISKEs väikeste tükikestena muude moodulite sees ja vahel teostatud. Eraldiseisvat moodulit selleks tegelikult ei ole, aga kontseptuaalselt on see hajus loogika joonisel kujutatud ühe moodulina mis juhib kõikide ülejäänud moodulite käitumist.

Diskreetseid juhtsignaale on protsessoril vaid kaks:

- „ENA“ signaal lubab protsessoril töötada. Näiteks on katkestuskontrolleri abil realiseeritud võimalus seisata protsessor järgmise maskimata katkestuseni. Seda kasutatakse ka siis kui mitmepordine mälu on näiteks hõivatud kõrgema prioriteediga moodulite teenindamisega.
- „INT“ signaali abil katkestatakse jooksva programmi töö ning siirdutakse katkestuste haldurisse mis salvestab jooksva konteksti ning annab juhtimise teenindamist vajava katkestuse haldurile. Kui „INT“ signaal on aktiivne rohkem kui 1 takt järjest, siis tõlgendab protsessor seda kui „RESET“-i.

PISKE suhtleb välismaailmaga läbi kahe liidese:

- „DATA RAM“ põhimälu mis võib erinevates rakendustes mitmel eri viisil teostatud. Enamik käske loevad argumente põhimälust ja tihti salvestavad tulemused sinna. Kuna põhimälu poole pöördumine on FPGA sees väga kiire, siis kasutatakse mälu algusosa (kuni 256 sõna) ka üldkasutatavate registrite salvena.
- „I/O BUS“ mille kaudu suheldakse protsessorist välja jääva perifeeriaga. Seda liidest kasutavad käsud IN ja OUT. I/O liidese aadressruum on täiesti eraldiseisev nii koodi kui andmete ruumist.

Aadressikalkulaator suudab mälu ja I/O liidese jaoks arvutada ühe aadressi takti kohta. See tähendab, et protsessor ei saa salvestada tulemust teisele aadressile kui see millelt argument loetakse. See tähendab samuti, et IN käsu poolt I/O aadressruumist loetud info saab minna vaid AX registrisse. Ja I/O aadressruumi saab kirjutada OUT käskudega vaid AX registrist võetud argumenti.

Protsessoris toimuvad kõik ühe käsuga seotud protsessid ühe kellatakti jooksul. Käsu ilmumisest koodimälu väljundisse kulub 1 takt tulemuse salvestamiseks mällu või registrisse. See tähendab et mitte ainult pole käskude läbilaskevõime 1/takti kohta vaid ka seda, et kõigi käskude latents on samuti 1 takt.

Selleks, et selle ühe takti jooksul saaks protsessoris toimuda kaks sündmust: argumendi lugemine mälust ja sellest töödeldud tulemuse salvestamine ühe ja sellesama takti jooksul, kasutatakse argumendi lugemiseks mälust kella langevat fronti, kõik muud sündmused (s.h. tulemuse salvestamine) toimuvad aga traditsiooniliselt kella tõusva frondi ajal. Sellise lähenemise teeb võimalikuks ka kõige väiksemate, lihtsamate, odavamate FPGA arhitektuuride mälumoodulite struktuur: kirjutamise ja lugemise portidel võib kasutada erinevat kella.

Kui PISKE-t peaks kasutatama FPGA arhitektuurides või projektides kus kella mõlema frondi kasutamine pole võimalik, siis saab teda kasutada ka ainult kella tõusvaid fronte kasutades, aga kaks korda kõrgema taktsagedusega.

Sellisel juhul kulub iga käsu täitmiseks kaks takti: esimene kulub argumendi lugemiseks mälust ning teine aritmeetika-loogika tehteks ja tulemuse salvestamiseks. Protsessori võimekuses kaotatakse sellise lähenemise korral küllalt vähe sest maksimaalne taktsagedus millega protsessor sellisel juhul töökindlalt töötada saab on peaaegu 2x kõrgem. Voolutarve on aga sellise lähenemise korral veidi kõrgem, mistõttu pole see PISKE jaoks standardlahendus.

Aadressruumid

PISKE jaoks eksisteerib kolm täiesti eraldiseisvat aadressruumi. Kõik kolm aadressruumi on maksimaalselt 64K 16-bitise sõna suurused. Igas konkreetses rakenduses on tegelikud mälude suurused enamasti väiksemad kui nimetatud maksimumid kuna reaalsetes FPGA-des on mälumoodulite ressurss piiratud. Kõikide aadressruumide poole saab pöörduda ainult 16biti kaupa:

- Andmete mälu, millest esimese 512x 16-bitise sõna poole saab otse, BP registrit kasutamata absoluutsete aadressidega pöörduda. Seda ruumi saab kasutada „registriteks“, tihti kasutatavateks konstantideks ning programmi ja draiverite globaalseteks muutujateks.
  Andmete mälu pesa [0x0000] on ainsana PISKE poolt fikseeritud rakendusega: sinna salvestatakse katkestuste korral katkestatud programmi täitmata jäänud käsu aadress +1. Selle kompromissi tõttu ei toeta PISKE raudvaraliselt katkestuste katkestamist.

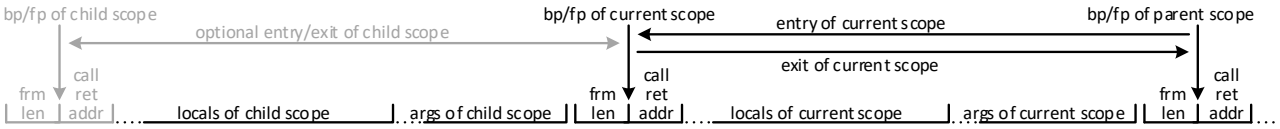
  Ülejäänud andmete mälu poole saab pöörduda relatiivse adresseerimisega BP registri suhtes. Kõik kahe operandiga käsud võimaldavad ühe operandi jaoks mälu kasutada nii otse (esimest 512 sõna) kui BP suhtes relatiivselt. Relatiivse adresseerimise vahemik on BP-256 .. BP+255 16-bitist sõna.

- I/O (sisendi/väljundi) aadressruum on mõeldud protsessorist väljas asuvate perifeerseadmetesse kirjutamiseks ja sealt lugemiseks. Ka I/O aadressruumis saab esimese 512x 16-bitise sõna poole pöörduda otse, BP registrit kasutamata. I/O käsud (IN ja OUT) võimaldavad perifeeriat samuti adresseerida relatiivselt BP registri suhtes. Ainult BP suhtes relatiivse adresseerimisega on võimalik pöörduda I/O aadressruumis esimesest 512-st sõnast kaugemale.

- Koodimälu ainus fikseeritud vektor on 0x0000, kuhu pöördub protsessor alati nii katkestuse kui ka „RESET“-i järel. Peale konteksti salvestamist peab katkestuste haldur kontrollima andmete mälu pessa [0x0000] kirjutatud katkestuse „return“ aadressi ning kui see on 0x0001 („INT“ sisend oli aktiivne rohkem kui 1 takt järjest), siis on tegemist „RESET“-iga. Kui see on aga 0x0001-st erinev, siis on tegemist katkestusega ja tuleb anda juhtimine sobivale draiverile katkestuse teenindamiseks.

Pinu

PISKE kuulub protsessorite hulka millel pole raudvaras teostatud pinu („stack“). Selle funktsioone täidab üldkasutatav viidaregister, milles hoitakse enamasti viita aktiivsele alamprogrammile kuuluvale pinupiirkonnale („stack frame“). Ajal mil viidaregistrit vajatakse muuks otstarbeks, hoitakse pinupiirkonna viita otseadresseeritavas mälus asuvas „fp“ registris kust teda saab vajadusel kergesti viidaregistrisse (bp) laadida.



Tänapäevased kompilaatorid liidavad nagunii peaaegu alati alamprogrammide algustesse ja lõppudesse alamprogrammi sisenemise ja väljumise koodijupid. Näiteks x86 platvormil teevad need jupid täpselt sedasama mida nad peavad PISKE puhul tegema: eraldama pinus iga alamprogrammi lokaalmuutujate ja argumentide jaoks vajaliku ruumi ning omistama viida pinupiirkonnale bp/fp registrisse. Erinevus tuleb PISKE jaoks sisse sellega, et tarkvaraliselt hallatud pinupiirkondade kõrval tavapärast raudvaras hallatud pinu ei olegi. Seda polegi tegelikult vaja, sest kompilaatori jaoks on täpselt sama kerge salvestada argument „mov“ käsuga [bp] viida suhtes mällu, selle asemel et seda salvestada „push“ tüüpi käsuga raudvaraliselt hallatud pinusse.

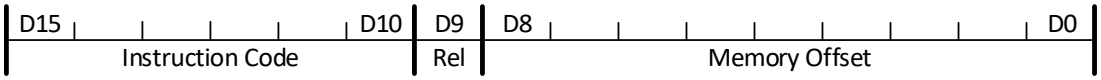
Kuna PISKE-l tavapärast pinu ei ole salvestab „call“ käsk alamprogrammist naasmise aadressi [bp+0] aadressile. Seetõttu *peab* enne „call“ käsu kasutamist olema „bp“ registris viit alamprogrammile kuuluvale pinupiirkonnale. Seega peavad kõik alamprogrammid mis plaanivad omakorda välja kutsuda alamprogramme, eraldama endale oma pinupiirkonna lahutades alamprogrammi sisenedes bp registrist selle alamprogrammi poolt vajatava pinupiirkonna suuruse. Alamprogrammist väljudes tuleb „bp“ ennistada selleks mis ta oli alamprogrammi sisenedes (liites bp-le sama väärtuse). Seejärel saab kasutada „ret“ käsku mis tegelikult on alias käsule „jmp [bp+0]“, et naasta väljakutsunud programmi.

Eelnevast järeldub ka, et alamprogrammid mis ei kutsu omakorda välja alamprogramme ei pea omale tingimata pinupiirkonda arvutama/eraldama. Sellised alamprogrammid saavad ikkagi kasutada pinu oma lokaalmuutujate tarbeks, aga nende jaoks asuvad argumendid ja lokaalmuutujad bp-st allpool. Alamprogrammide jaoks kes eraldavad omale oma pinupiirkonna, asuvad lokaalmuutujad ja argumendid bp-st kõrgemal.

Et võimaldada silumisprogrammidele pinu ja alamprogrammide hierarhia analüüs ja näitamine ning lihtsustada veidi alamprogrammide väljumise koodijuppide tööd salvestatakse pinupiirkonna kindlasse kohta [bp-1] ka selle pinupiirkonna pikkus.

Käsustik

Kuna PISKE on üle kõige optimeeritud võimalikult vähe FPGA ressurssi vajama, siis on ka tema käsustik lihtne ja minimalistlik. Enamik käske koosnevad 6-bitisest käsukoodist ja ühe argumendi adresseerimiseks absoluutsest aadressist või BP suhtes relatiivsest ofsetist:



Lisaks käsukoodile on sellistes käskudes juhised mälu adresseerimiseks:

- 9-bitine „Memory Offset“ mis võib kujutada:
  - absoluutset (unsigned) aadressi millega saab pöörduda mälu esimese 512 sõna poole
  - suhtelist (signed) ofseti mis liidetakse BP registrile ja saadakse aadress kuhu pöörduda
- „Relative“ lipp, mis määrab kas 9-bitist ofseti käsitleda absoluutse (0) või suhtelise (1) aadressina

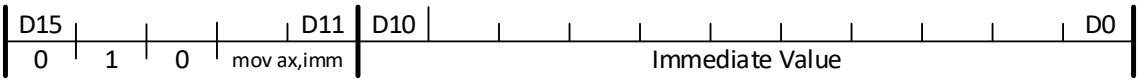
Sellised käsud on:

Käsukood	Mnemoonika	Toime
000001	jmp <mem>	hüpatakse 16bit aadressile mis loetakse mälust
011000	mov ax,<mem>	ax register loetakse mälust
011001	mov <mem>,ax	ax register salvestatakse mällu
011010	mov flags,<mem>	Z ja C lipud loetakse mälust
011011	mov <mem>,flags	Z ja C lipud salvestatakse mällu
011100	mov bp,<mem>	bp register loetakse mälust
011101	mov <mem>,bp	bp register salvestatakse mällu
011110	mov bp,ax	ax register kopeeritakse bp registrisse (mälu ofset ja „Relative“ bitt peavad olema 0)
011111	mov ax,bp	bp register kopeeritakse ax registrisse (mälu ofset ja „Relative“ bitt peavad olema 0)
100000	out <io>,ax	ax register saadetakse absoluutselt või relatiivselt adresseeritud I/O porti
100010	in ax,<io>	ax register loetakse absoluutselt või relatiivselt adresseeritud I/O pordist
100100	add bp,<mem>	bp registrile liidetakse mälust loetud väärtus
100101	add bp,ax	bp registrile liidetakse ax registri sisu
100110	sub bp,<mem>	bp registrist lahutatakse mälust loetud väärtus
100111	sub bp,ax	bp registrist lahutatakse ax registri sisu
101000	shl ax	ax registrit nihutatakse 1 biti võrra vasakule ja LSB täidetakse 0-ga
101001	shl <mem>,ax	nagu eelmine käsk, aga tulemus salvestatakse mällu

Käsukood	Mnemoonika	Toime
101010	rol ax	ax registrit nihutatakse vasakule ja LSB täidetakse C lipuga
101011	rol <mem>,ax	nagu eelmine käsk, aga tulemus salvestatakse mällu
101100	sar ax	ax registrit nihutatakse paremale ja MSB täidetakse märgibitiga (endise ax väärtuse D15 bitt)
101101	sar <mem>,ax	nagu eelmine käsk, aga tulemus salvestatakse mällu
101110	ror ax	ax registrit nihutatakse paremale ja MSB täidetakse C lipuga
101111	ror <mem>,ax	nagu eelmine käsk, aga tulemus salvestatakse mällu
110000	xor ax,<mem>	ax registri sisu XOR-itakse mälust loetud argumendiga ja tulemus pannakse ax registrisse
110001	xor <mem>,ax	nagu eelmine käsk, aga tulemus salvestatakse mällu tagasi
110010	or ax,<mem>	ax registri sisu OR-itakse mälust loetud argumendiga ja tulemus pannakse ax registrisse
110011	or <mem>,ax	nagu eelmine käsk, aga tulemus salvestatakse mällu tagasi
110100	and ax,<mem>	ax registri sisu AND-itakse mälust loetud argumendiga ja tulemus pannakse ax registrisse
110101	and <mem>,ax	nagu eelmine käsk, aga tulemus salvestatakse mällu tagasi
110110	test ax,<mem> test <mem>,ax	ax registri sisu AND-itakse mälust loetud argumendiga ja uuendatakse Z lippu vastavalt AND tehte tulemusele (argumentide järjekord assembleri mnemoonikas käsu koodi ega käitumist ei mõjuta)
110111	test bp,<mem> test <mem>,bp	bp registri sisu AND-itakse mälust loetud argumendiga ja uuendatakse Z lippu vastavalt AND tehte tulemusele (lahutamise tulemust ei salvestata ning argumentide järjekord assembleri mnemoonikas käsu koodi ega käitumist ei mõjuta)
111000	add ax,<mem>	ax registri sisu liidetakse mälust loetud argumendile tulemus pannakse ax registrisse
111001	add <mem>,ax	nagu eelmine käsk, aga tulemus salvestatakse mällu tagasi
111010	adc ax,<mem>	ax registri sisule liidetakse mälust loetud argument ja C lipu väärtus ning tulemus pannakse ax registrisse
111011	adc <mem>,ax	nagu eelmine käsk, aga tulemus salvestatakse mällu tagasi
111100	sbb ax,<mem>	ax registrist lahutatakse mälust loetud argument ja inverteeritud C lipp ning tulemus pannakse ax registrisse. C lipu väärtuseks saab „0“ kui järgmistest kõrgematest bittidest tuleks 1 laenata ja „1“ kui laenamist ei ole vaja teha (lahutamise ajal alatäitumist ei toimunud).
111101	sbb <mem>,ax	mälust loetud argumendist lahutatakse ax registri sisu ja inverteeritud C lipp ning tulemus pannakse ax registrisse. C lipu väärtuseks saab „0“ kui järgmistest kõrgematest bittidest

Käsukood	Mnemoonika	Toime
		tuleks 1 laenata ja „1“ kui laenamist ei ole vaja teha (lahutamise ajal alatäitumist ei toimunud).
111110	cmp ax,<mem>	ax registrist lahutatakse mälust loetud argument aga tulemust ei salvestata (Z ja C lippe uuendatakse sarnaselt ilma laenamiseta tehtud sbb käsule)
111111	cmp <mem>,ax	mälust loetud argumendist lahutatakse ax registri sisu aga tulemust ei salvestata (Z ja C lippe uuendatakse sarnaselt ilma laenamiseta tehtud sbb käsule)

Veidi teistsuguse struktuuriga on käsud ax registri laadimiseks 16-bitise käsu sees asuvast vahetust 11-bitisest argumendist:

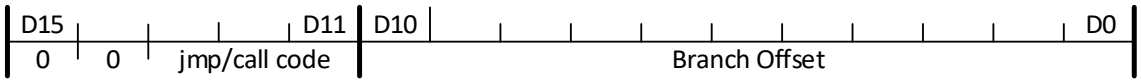


Käsukood	Mnemoonika	Toime
01000	mov ax,<imm>	ax registri alumised 11 bitti laetakse vahetust argumendist, ülemised 5 bitti täidetakse nullidega („zero extended“)
01001	mov ax,<imm>	ax registri alumised 11 bitti laetakse vahetust argumendist, ülemised 5 bitti täidetakse märgiga („sign extended“)
01010	mov ax,<imm>	ax registri ülemised 5 bitti laetakse vahetust argumendist (käsu bittidest D7-D3), alumised 11 bitti täidetakse nullidega
01011	mov al,<imm>	ax registri alumised 11 bitti laetakse vahetust argumendist, ülemised 5 bitti jäetakse muutmata

Esimese kolme ax registrile vahetu argumendi omistamise käsu mõte on võimaldada valdaval enamikul juhtudest omistada ax registrile ühe 16bit käsuga uus väärtus hoolimata sellest, et käsu sisse mahub ära vaid 11-bitine vahetu argument. Puhkudel kui on vaja omistada väärtust mida ühe käsuga omistada ei saa, tuleb vaja kasutada kaht käsku järjest, näiteks väärtuse 0AAAAh omistamiseks:

```
mov     ax,0A800h
mov     al,02AAh
```

Tingimuslikud ja tingimusteta siirdekäsud on järgmise struktuuriga:



Kõik vahetu argumendiga siirdekäsud on suhtelised: hüpata saab jooksva käsuaadressi suhtes -1024 .. +1023 ulatuses:

Käsukood	Mnemoonika	Toime
00010	call <rel.addr>	Salvestab call käsule järgneva käsu aadressi andmemällu aadressile [bp] ning teeb tingimusteta relatiivse hüppe
00011	jmp <rel.addr>	Teeb tingimusteta relatiivse hüppe
00100	jz <rel.addr>	Teeb relatiivse hüppe kui Z lipp on 1 (kui eelmise aritmeetika-loogikatehte tulemus oli 0)
00101	jnz <rel.addr>	Teeb relatiivse hüppe kui Z lipp on 0 (kui eelmise aritmeetika-loogikatehte tulemus ei olnud 0)
00110	jc <rel.addr>	Teeb relatiivse hüppe kui C lipp on 1 (kui eelmise aritmeetikatehte ajal toimus ületäitumine)
00111	jnc <rel.addr>	Teeb relatiivse hüppe kui C lipp on 0 (kui eelmise aritmeetikatehte ajal ei toimunud ületäitumist)

Taktsagedus

Näidisprojekt PISKE maksimaalse taktsageduse illustreerimiseks teostab USB Full-Speed seadme, mille PHY on integreeritud FPGA sisse ning mille kontrollerite tegevust juhib ning mille tarkvara töötab PISKE peal. Projekt koosneb lisaks protsessorile mitmest perifeerseadmest, aga maksimaalse taktsageduse piirid seab siiski PISKE, kuna väikese suuruse saavutamiseks on tegu „non-pipelined“ arhitektuuriga mis kasutab kella mõlemat fronti. Seda võib käsitleda taktsageduse x2 korrutamisega ilma selleks PLL-i tarvitamata. Sellise arhitektuuri eelis on ka väikseim võimalik voolutarve arvutusvõimsuse kohta võrreldes 2x kõrgema taktsagedusega ja 2 takti/käsk arhitektuuriga.

Kui aga FPGA-s 2x kõrgem taktsagedus saadaval siis on väga lihtne kasutada PISKE-t ainult kella tõusvate frontidega nii, et iga käsu täitmiseks kulub 2 takti.

Mõlemas järgnevas tabelis toodud kiirused on sellised mille puhul FPGA implementatsiooni „timing analysis“ annab tulemuseks ajastusvigade puudumise ka halvimas ja aeglaseimas protsessi nurgas ehk õnnetuimate integraalskeemidega mis napilt tootmise kvaliteedikriteeriume täidavad, kõrgeima temperatuuri ja madalaima lubatud toitepinge juures.

Esimeses tabelis on toodud optimeeritud platvormidel saavutatud taktsagedused. Nii RTL lähtekood kui loogika paigutus FPGA sees on neil käsitsi optimeeritud:

Platvorm	**Taktsagedus**	**PISKE + kellakontroller + katkestuskontroller (5 sis.) suurus**
Lattice iCE40 UltraLite	24 MHz	290 LUT
Lattice iCE40 HX	30 MHz	290 LUT

Teises tabelis on toodud platvormid kus platvormist sõltumatu, käsitsi optimeerimata RTL lähtekood on sünteesitud mitmete erinevate platvormide peale:

Platvorm	**Taktsagedus**	**PISKE + kellakontroller + katkestuskontroller (5 sis.) suurus**
Altera Max10 10M04SAU169C8G	46 MHz	340 LUT
Altera Cyclone IVE *(odav)* EP4CE6E22C8	45 MHz	340 LUT
Altera Cyclone IVE *(kiire)* EP4CE6E22C6	61 MHz	340 LUT
Altera Cyclone V *(odav)* 5CEBA2F17C8	50 MHz	170 ALM
Altera Cyclone V *(kiire)* 5CEBA2F17C6	63 MHz	170 ALM
Xilinx Artix-7 *(odav)* XC7A15TFTG256-1	62 MHz	226 LUT
Xilinx Artix-7 *(kiire)* XC7A15TCPG236-3	84 MHz	226 LUT
Xilinx Kintex-7 XC7K70TFBG484-3	109 MHz	226 LUT
Xilinx Kintex Ultrascale XCKU035-FBVA676-3-E	139 MHz	226 LUT
Xilinx Kintex Ultrascale+ XCKU3P-FFVA676-3-E	183 MHz	226 LUT

Suhteliselt väikese vaevaga õnnestuks ka neil platvormidel optimeerimise läbi PISKE implementatsiooniks vajalikke ressursse vähendada. Kindlasti tuleks Cyclone III/IV arhitektuuride jaoks mõistlikult optimeeritud implementatsioon väiksem kui iCE40 implementatsioon kuna Altera Cyclone III/IV seeria LUT-id on märksa võimekamad kui iCE40 LUT-id.

Terminid

FPGA Field Programmable Gate Array – programmeeritav (loogika)integraalskeem, millesse laetakse peale voolu sisselülitamist (enamasti FLASH mälust) loogikalülitustest, trigeritest ja muudest komponentidest koostatud skeem. Seejärel on FPGA integraalskeem valmis täitma kasutaja soovitud ülesandeid samuti nagu tehases toodetud integraalskeem (ASIC).

HDL Hardware Description Language – mõni paljudest keeltest millega kirjeldatakse elektroonikat (ja selle käitumist). Tüüpilised näited on Verilog ja VHDL.

LUT Abstraktne loogikaelement (Look-up Table) milledest (koos trigeritega) koosnevad FPGA integraalskeemid. LUT-ide hulk FPGA-s on üks mitmest parameetrist mis kirjeldab selle mahtu ja võimekust. Samuti on vajatavate LUT-ide arv indikaatoriks mis iseloomustab FPGA jaoks sünteesitud skeemi mahtu ja ressursivajadust.

Süntees Protsess mille abil muundatakse HDL keeles kirjeldatud „raudvara“ FPGA konfigureerimiseks vajalikuks andmefailiks mis salvestatakse FLASH mällu millest andmete lugemisega FPGA sisselülitamise järel ennast tööks valmis seab.

ISA Instruction Set Architecture – käsustik mida konkreetne protsessor täita oskab.

Autoritest

Protsessori ISA, Verilog RTL implementatsiooni, perifeeriakomponentide, perifeeriakomponentide draiverite, näidisprojekti, Verilog testbenchi, assembleris realiseeritud näidisrakenduse ja käesoleva dokumendi autor on Sulo Kallas.

C-kompilaatori arendamisega tegeleb Jaan Hendrik Murumets.