TAMPEREEN
AMMATTIKORKEAKOULU

TEOREETTINEN NELIBITTINEN PROSESSORI

Tino Nordlund

Opinnäytetyö
Tammikuu 2017
Tieto- ja tietoliikennetekniikka
Sulautetut järjestelmät

TIIVISTELMÄ

Tampereen ammattikorkeakoulu
Tieto- ja tietoliikennetekniikka
Sulautetut järjestelmät

NORDLUND TINO
Teoreettinen nelibittinen prosessori

Opinnäytetyö 31 sivua, joista liitteitä 9 sivua
Maaliskuu 2017

Tietotekniikan saadessa yhä suurempaa jalansijaa yhteiskunnassamme, sen rakenteet, taustalla oleva toiminta vajoaa itsestäänselvyytenä mysteeriksi. Mysteeri on hyvin haudattu. Sitä ei käsitellä edes sulautettujen järjestelmien koulutusohjelmassa kuin pienin vihjein ja viittauksin. Lukemattomien myöhäisiltojen jälkeen saatiin vihi kahden transistorin ja vastuksen rakentamasta NAND-portista, jota monistamalla voidaan rakentaa toimivia loogisia kokonaisuuksia, jopa toimivaksi prosessoriksi asti. Ajatusta pyöriteltiin jonkin aikaa, kunnes löytyi helppo tapa testailla logiikkaa puhelimella käyttäen selaimen javascript-tulkkia. Tämä mahdollisti ajatuksen välittömän testaamisen heti sen tultua ja toimi sanoinkuvaamattomana apuna.

Myöhemmin loogisten operaatioiden rakentuessa tuli eteen kysymyksiä bittimäärästä ja käskykannan laajuudesta. Esoteeriset kielet ja vanhemmat kotitietokoneet, kuten Commodore 64, tulivat apuun rakenteidensa kanssa. Kuitenkin huomattiin kahdeksan bitin olevan rakenteeltaan hieman turhan laaja tuoden paljon turhaa toistuvuutta, joka ei vaikuta loogiseen pätevyyteen mitenkään. Täten supistettiin rakennetta neljään bittiin, sillä käytettiinhän Intelin nelibittistä 4004-prosessoriakin aikanaan.

Assembleri hyppykohtineen (goto-labels) antoi lisäarvoa prosessorille mahdollistamalla aliohjelmien helpon suunnittelun. Maininnan arvoinen disassembler myös purkaa ohjelman takaisin pelkistetyksi assembleriksi, joten ohjelmapätkien tai koko ohjelman tarkistaminen onnistuu samalla näkymällä.

Simulaattori mahdollistaa suunnitellun ohjelman ajamisen ja muunmuassa koodin ajamisen yksi komento kerrallaan, jolloin toiminnan tarkastelu helpottuu. Näkymän taulukossa on myös rekisterien tämänhetkiset arvot, jolloin mikään toiminnallinen osuus ei jää näkemättä.

Asiasanat: nand, transistori

ABSTRACT

Tampere University of Applied Sciences
Degree Programme in ICT Engineering
Embedded Systems

TINO NORDLUND
Theoretical four bit processor

Bachelor's thesis 31 pages, appendices 9 pages
March 2017

Computers and computer science has gained it's footstep in our everyday life but its structure and details are hidden in the background and taken for granted. There's little to no clues what goes on before taking huge amount of time to study the matter, mostly in late night when motivation has its highest peak. At one of these nights there was an article talking about the NAND gate which can be used to form logical structures, even up to working processor.

The thought was being processed for some time, mostly using mobile phones browser's build in Javascript interpreter that made it possible to test any ideas instantly at the time of the innovation.

Later when logical operations were formed, more theoretical questions arrived about bit count and instruction set. Esoteric languages and old home computers like Commodore 64 proved themselves as a huge benefit when inventing the way of this processor. However, the 8 bit system deemed to have too much repetition and didn't have much to bring in terms of logical operation, so 4 bit system was chosen. After all, Intel 4004 was 4 bit processor that was used greatly and it proves the capabilities of a 4 bit system.

Goto-labels added a huge value to assembler making it possible to program scalable subprograms so that programmer won't have to calculate the length himself. Also disassembler makes a tremendious tool for debugging part of the or even the whole code.

About debugging, the most valuable tool must be the simulator. The simulator was formed to be able to run code one or many steps at a time with real time data of register's current value.

Key words: nand. transistor

SISÄLLYS

1 JOHDANTO 5
2 FYYSINEN TOTEUTUS 6
3 SIMULOINNIN POHDINTA 8
4 SIMULOINNIN TOTEUTUS 9
5 HALF-ADDER JA LIPPUREKISTERI 14
6 FULL-ADDER 15
7 PROSESSORIN VALINTA 16
8 REKISTERIT 19
9 MUISTIN RAKENNE JA KÄSKYKANTA 20
10 SIMULAATTORI 29
11 ASSEMBLER 30
12 ASSEMBLERIN TOIMINTA 32
13 POHDINTA 35

LÄHTEET 36
LIITTEET 37
Liite 1. Tiedosto hdl.js 37

JOHDANTO

Eräänä sateisena päivänä säännönmukaisesti komentoja suorittavaa sekvensseriä, siis kaupan elektronista valojärjestelmää seuratessa huomattiin valojen toimivan säännönmukaisesti. Kuitenkin, kun sade lisäsi oman tulosignaalinsa järjestelmään, valojen nopeuden huomattiin hieman kasvavan. Sekvensseri, joka muuttaa lopputulemaansa perustuen tulosignaaleihin, on kuin prosessori. Sekvensseri taas rakentuu logiikkaporteista, joten idea alkoi rakentumaan. Kuitenkin tietokoneen toiminnasta puhuttaessa mainitaan aina Boolen algebra, sekä miljoonia transistoreja pienellä pinta-alalla.

Boolen algebra on 1800-luvulla George Boolen kehittämä ajatusmalli, jossa on kolme erilaista operaatiota kahdelle eri arvolle. Arvot ovat joko 0 tai 1, tai kuten hilateoriassa, ylempi ja alempi. Operaatioita, eli laskutoimituksia, on kolme erilaista. Ja-operaatio esimerkiksi joukko-opissa on helppo ymmärtää; meillä on kaksi joukkoa, jolloin näiden ja-operaatio on kummatkin joukot. Tai-operaatio tarkoittaa jompaa kumpaa joukoista. Kolmas operaatio ei käsittele kahta joukkoa, kuten aikaisemmat. Kahta joukkoa tai muuttujaa käsitteleviä operaatioita kutsutaan binäärisiksi operaatioiksi. Kolmas operaatio, eli ei-operaatio on kaikki paitsi joukko. Tämä on niin sanottu unäärinen operaatio.

Boole määritteli, tai pikemminkin huomasi muutaman lain toimivan kyseisille operaatioille ja arvoille. Liitäntälaki pätee, eli operaatio toimii kuten kertolasku välittämättä siitä mitkä viereiset luvut kerrotaan yhteen. Vaihdantalaki on toinen laki, jonka vastaava kertolaskuvertaus olisi kuinka neljä kertaa viisi on sama kuin viisi kertaa neljä. Nämä kaksi lakia ovat runsaassa käytössä, kun fysiikan yhtälöitä ratkotaan yhden muuttujan suhteen.

Absorptiolaki on kolmas laki, joka ei ole enää niin helposti lähestyttävä, joten tähän ei kannata liian pitkäksi aikaa jumiutua. Jos meillä on kaksi ihmisryhmää, ensimmäinen, muttei ensimmäinen ja toinen, se tarkoittaa että meillä on vain ensimmäinen ryhmä.

Sama pätee kun meillä on ensimmäinen ja ensimmäinen muttei toinen ryhmä, jolloin jäljellä on vain ensimmäinen ryhmä kahdesti, ja siten ensimmäinen ryhmä on valittu.

Osittelulait jakaantuvat unääri- ja binääri-operaatioihin. Ei-operaatiolla ryhmä ja ei ryhmä on aina 1, sillä kuuluit ryhmään tai et, olet aina olemassa. Ryhmään kuuluminen ja kuulumattomuus samanaikaisesti taas ei voi koskaan pitää paikkaansa, jolloin tulos on 0.

Toinen osittelulaki vaatii kolme joukkoa, mutta operaatiot ovat silti kahden joukon kesken, eli binääri-operaatioita. Ensimmäinen ryhmä muttei toinen ja kolmas ryhmä tarkoittaa samaa kuin ensimmäinen ryhmä muttei toinen ryhmä ja ensimmäinen muttei kolmas. Sama pätee kun vaihdetaan ja- ja tai-operaatioiden paikkoja. Kuten sanottua, kahta jälkimmäistä lakia ei ole oleellista ymmärtää tätä opinnäytetyötä varten, joten ne on vain mainittu kokonaisuutta täydentämään.

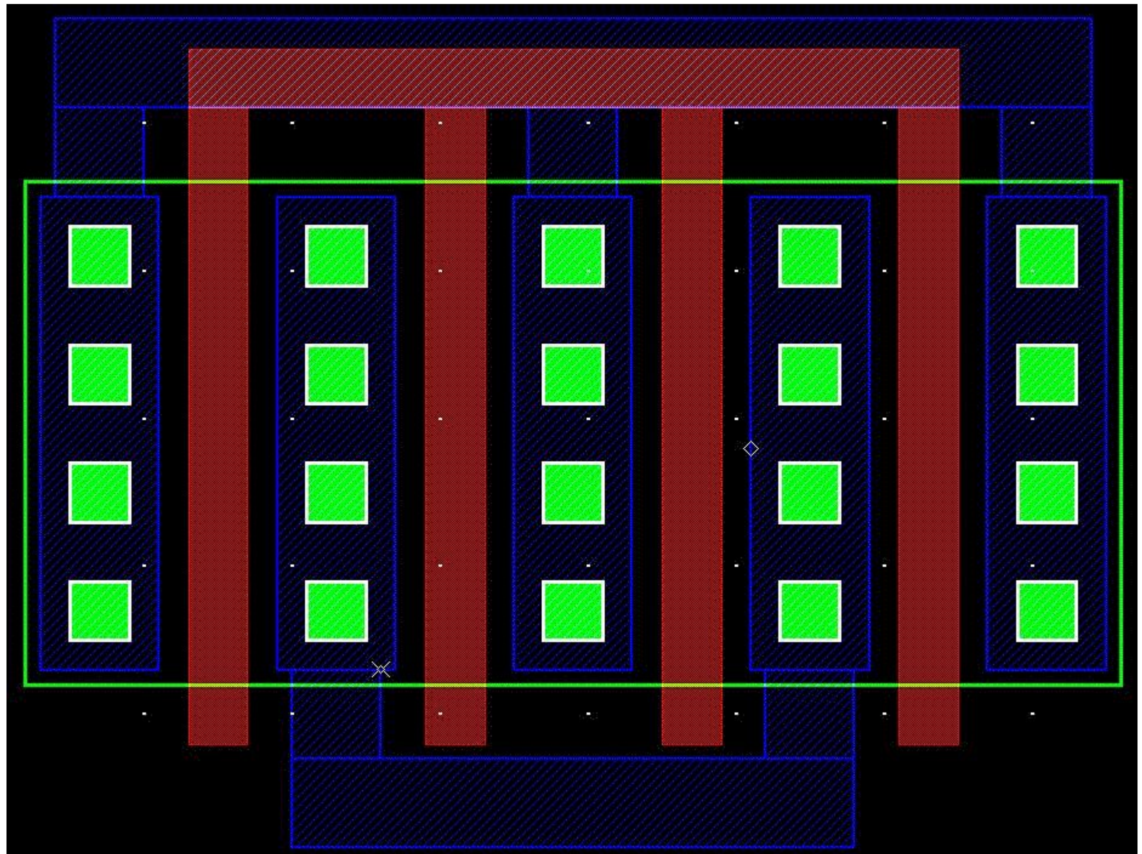
Boolen algebran hyötykäyttäminen tai käytännönläheisyys voi ihmetyttää. Meillä on kolme laskutoimitusta ja kaksi arvoa, joita kaksi arvoa voi kuvata esimerkiksi onko olohuoneen valo päällä. Käytännössä näillä kahdella arvolla ohjataan aina vain seuraavaa operaatiota.

FYYSINEN TOTEUTUS

Kuten johdannossa mainittiin, kytkin toimii hyvin operaationa. Sillähän on tilat päällä tai pois. Pelkillä kytkimillä ohjattava tietokone ei olisi käyttäjäystävällinen, eikä se tekisi itsestään mitään. Tarvitaan sähköisesti ohjattavia kytkimiä, kuten releet, putket, käämit tai halvin, helpoin ja vähävirtaisin vaihtoehto, transistorit. Nämä voivat vähillä oheiskomponenteilla jopa kytkeä itseään päälle tai pois, jolloin ne voivat vaikka ajoittaa koko kytkennän toimintaa tasaisilla pulsseilla.

Transistorin esimerkkirakenne löytyy esimerkiksi alla olevasta kuvasta 1. Kuva on Cadence-nimisestä ohjelmasta, joka on tarkoitettu IC-piirien suunnittelua varten.

Piisirulla on monta eri kerrosta. Punainen ja sininen kuvaavat kahta eri kerrosta ja vihreät neliöt yhdistävät nämä kerrokset.

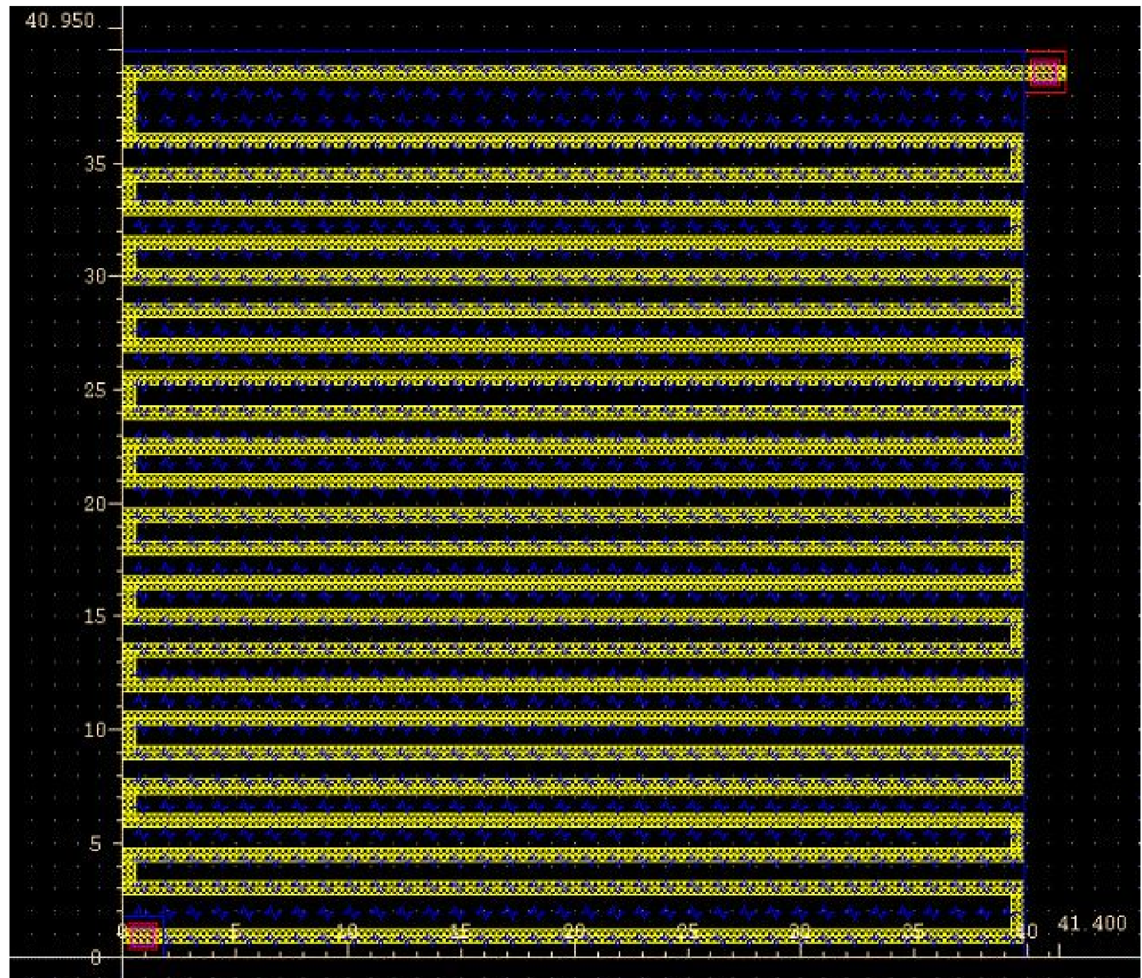


KUVA 1. MOSFetin rakenne (pages.cs.wisc.edu, 2017)

Kuvasta 1 voidaan hieman havaita transistorin toimintaa. Oleellista on, että transistori on viritetty tilaan, jossa virta ei vielä kulje. Kun tässä tilanteessa hilalle ohjataan pieni jännite, saadaan vähän elektroneja hyppäämään transistorin yli ja suurella ohjausjännitteellä niin ikään enemmän elektroneja. Käytännössä tämä tapahtuu siten, että piisirun eri kerroksilla on pii-atomeja, jotka ovat niin sanotusti doupattuja, eli niihin on sekoitettu epäpuhtauksia, eli eri alkuaineita. Tällöin jokin kerros on esimerkiksi positiivisesti varautunut, koska sillä kerroksella on puutos elektroneista. Siis, elektronit ovat jo reunalla ja ohjausjännite antaa pienen tuuppauksen jotta elektronit uskaltavat mennä sinne minne vetovoimat niitä ohjaavat.

Pelkän transistorin käyttäminen kytkimenä tarkoittaisi suurta virrankulutusta. Virran määrää, siis liikkuvien elektronien lukumäärää, voidaan rajata antamalla sille

epäsuotuisat olot kulkea. Hyvä esimerkki on kuvassa 2.



KUVA 2. Vastus (webpages.eng.wayne.edu 2017)

Kuvassa 2 virran kulkua on vaikeutettu ohjaamalla se pitkän ja ohuen johteen kautta. Tämä toimii käytännössä kuten vastus, kunhan taajuudet eivät nouse liian korkeiksi. Siis vastus rajoittaa virran kulkua, jolloin transistori voidaan ohjata toimimaan pienellä virralla.

SIMULOINNIN POHDINTA

Kun on käytössä valmis tietokone, voidaan tehdä simulaatiota, eli tarkastaa logiikan toiminta. Logiikan testaukseen voidaan käyttää esimerkiksi SPICE-direktiiviin pohjautuvia simulaattoreita, mutta näiden tekeminen vaatii paljon suunnittelua ja

editointi, saati simulaatio on kaikkea muuta kuin reaaliaikaista. Toinen, mahdollisesti miellyttävämpi vaihtoehto on käyttää raudan, eli komponenttitason kuvaamiseen erikoistuneita kieliä (HDL, hardware description language). Näiden ongelmaksi huomattiin vaikeaselkoisuus, sekä niiden kaupallisuus. Käytännössä helpoin tapa käyttää HDL-kieliä on ostaa kehitysalusta, joka ohjelmoidaan kyseisellä kielellä, mutta tämä ei tunnu taloudelliselta, saati joustavalta ratkaisulta. Suurin ongelma on suoraan ratkaisun hinta ja käyttöjärjestelmärajoitukset, sillä toimivuus Linux-järjestelmissä on olematon. Vaihtoehdoiksi jää valmiin avoimen lähdekoodin ratkaisun koodin lukeminen, ymmärtäminen ja muokkaaminen. Tarvetta ei kuitenkaan ole monille toiminnoille, joten nähtiin järkeväksi tehdä itse vastaava raudan kuvaamiseen tarkoitettu ohjelmointikieli, HDL.

HDL:n toteutukseen ensimmäinen askel on päättää ohjelmointikieli projektin rakentamiseen. Ohjelmointia vaaditaan joka tapauksessa, joten C ja C++ olivat järjestelmäkielinä vahvoilla. Näiden rajoitukset tulivat vastaan ohjelman vianhaussa ja näiden kielien kehittäminen oli mahdotonta puhelimella, jolla HDL lopulta suurilta osin rakentui. Käytännössä lopputulokseen päädyttiin käyttäen repullinen muistilappuja ja puhelimen selaimen Javascript-tulkkia. Vaihtoehdoiksi jäi siis puhelimessa toimivat tulkattavat ohjelmointikielet, Ruby, Python ja Javascript. Javascript tarjoaisi etuutenaan mahdollisuuden integroida ohjelma nettisivuun ja näin saada helposti käytettävä ohjelma. Kun logiikka on tehty jollain kielellä, on se mahdollista siirtää toiselle ohjelmointikielelle. Jos ohjelma ei toimi tarpeellisella nopeudella Javascriptillä kirjoitettuna, voidaan ohjelmointikieltä vaihtaa jälkikäteen. Teoria ja toteutustapa oli näin ollen valittu.

SIMULAATTORIN TOTEUTUS

"Kaikki Boolen algebran operaatiot voidaan toteuttaa NAND portilla" (luentomuistiinpano Digitaalitekniikkakurssilta, Mauri Inha). NAND rakentuu not ja and porteista, eli ei-operaation ja ja-operaation sisältämistä porteista. Portti tarkoittaa tässä operaatiota tai operaatiojoukkoa. Meillä on tiedossa transistorin ja vastusten

rakenne, joten yritetään pärjätä niillä. NAND-portin tulee toteuttaa seuraava totuustaulu taulukossa 1, jossa a ja b ovat tulevia signaaleja ja q on lähtösignaali.

TAULUKKO 1. NAND-portin totuustaulu

a	**b**	**q**
0	0	1
0	1	1
1	0	1
1	1	0

Aloitetaan suunnittelu ohjaamalla käyttöjännite suoraan lähtöön, sillä lähtösignaali on 1, eli käyttöjännitteet, suurimman osan ajasta. Ainoa tilanne jossa tämä ehto ei toteudu on kun kummatkin tulosignaalit ovat käyttöjännitteissä, eli loogisessa 1-tilassa. 1-tilan ohjaaminen transistorille sai virran kulkemaan transistorin lävitse, joten ohjataan tulosignaalit omiin transistoreihinsa ja asetetaan ne sarjaan ohjaamaan lähtösignaali maihin. Tällöin tulosignaalien ollessa 0 ja 0 lähtösignaali on 1. Kun jompikumpi transistoreista on johtavassa tilassa, eli saa tulosignaalin 1, ei lähtösignaali laske 1-tilasta. Kun kummatkin ovat päällä, laskee lähtösignaali, joten kytkentä vaikuttaa järkevältä. Koska Javascript-simulaattorin ei tarvitse kuin loogisesti toimia samalla tasolla, luodaan NAND-portti alla olevaksi funktioksi. Funktio, eli aliohjelma, sopii toteutukseksi vallan mainiosti, sillä se voi ottaa arvoja sisäänsä ja palauttaa niitä.

Koodi on suoraan kopioitu lopullisesta simulaattorista ja käyttää Javascriptin valmiita ei- ja ja-operaatioita. Nimeämiskäytäntönä käytettiin Boolen algebran loogisista operaatioista ei-, ja- ja tai-operaatioita, luentomuistiinpanoissa esiintyvistä logiikkaporteista NOT-, AND- ja OR-portti nimityksiä ja javascriptissä not-, and- ja or-funktioita. Loogisesti kaikki toimivat samoin, mutta ovat olemassa eri asiayhteyksissä. Simulaattori on toteutettu englannin kielellä, jotta sen jälkimuokkaaminen ja muihin projekteihin yhdistäminen olisi helppoa myös heille, jotka eivät suomen kieltä ymmärrä.

```
function nand(a, b){
   // nand as it says, not-and is
   // logically like this
   // Not ( a and b )
   return ! ( a & b ) ;
}
```

Kun NAND-portti on rakennettu, palataan takaisin Boolen logiikan alkeisiin. Jotta voisimme toteuttaa Boolen logiikan, tarvitsemme ei-, ja- ja tai-portit. Ei-portti päädyttiin toteuttamaan seuraavanlaisesti. Kun NAND-portin molemmat tulonastat saavat saman arvon, absorptiolain nojalla lopputulos on negaatio, eli toimii, kuten ei-operaatio.

```
function not( a ){
   // So we only have NAND. What if we put         IN   OUT
   // a to both inputs? Yep, not is just that.     0   1
   // In real life we could use just 1 transistor. 1   0
   return nand( a, a );
}
```

Kuten huomataan, not-funktion sisällä käytetään vain nand-funktiota, eli kahta transistoria ja vastusta. Seuraavaksi ja-operaatio. Negaation negaatio on luku itse, joten päädytään seuraavan sivun ratkaisuun.

```
function and( a, b ){
   //   A   B   OUT
   //   0   0     0
   //   0   1     0
   //   1   0     0
   //   1   1     1
   return ( not ( nand ( a , b ) ) );
}
```

And-funktio käyttää not- ja nand-funktioita, joista not sisältää vain nand-funktion, joten koko toteutus rakentuu vieläkin NAND-porttiin. Tai-operaatio tuottaa suurinta ongelmaa ja vaati hieman pohdintaa kynän ja paperin kanssa, mutta tärkeintä oli kokeilu ja simulointi jo muodostetuilla funktioilla.

```
function or( a, b ){
   //   A   B   OUT
   //   0   0     0
   //   0   1     1
   //   1   0     1
   //   1   1     1

   //
   //       1     2     3
   // A B   A B   OUT
   //
   // 0 0   1 1   0
   // 0 1   1 0   1
   // 1 0   0 1   1
   // 1 1   0 0   1
   //
   return nand( not( a ) , not( b ) );
}
```

Kun perusoperaatiot saatiin suunniteltua, osin mielenkiinnon ja osin jatkosuunnittelun helpottamiseksi toteutettiin muitakin loogisia portteja. NOR-portti rakentui lyhyesti not- ja or-porteista.

```
function nor(a,b){
   return not(or(a,b));
}
```

XOR-portti olisi hyvän ohjelmointikäytännön mukaisesti toteutettu muuttujilla, mutta meillä ei ole kuin nand, joten tulosta sievennettiin kahteen otteeseen ja muuttujien tarpeesta päästiin eroon. Samalla simulaattori menetti osan luettavuuttaan, mutta oleellisempaa oli pitäytyä vain nand-funktion käytössä ainakin yksinkertaisissa porteissa.

```
function xor(a,b){

   // var tmp_1 = nand( a, b );
   // var tmp_2 = nand( a, tmp_1 );
   // var tmp_3 = nand( b, tmp_1 );
   // return nand( tmp2 , tmp3);

   // we really don't have any memory yet
   // to store these, so let's just make the monster.

   // return nand( tmp2 , tmp3);
   // return nand( nand( a, tmp_1 ) , nand( b, tmp_1 ));
   return nand( nand( a, nand( a, b ) ) ,
              nand( b, nand( a, b )));

}
```

HALF-ADDER JA LIPPUREKISTERI

Kun perusportit oli suunniteltu, alkoi prosessorin rakenne hieman hahmottua. Se tarvitsee joka tapauksessa laskutoimituksia ja niiden perusta on summa. Summa muodostetaan FULL-ADDER -kytkennällä, joka pohjautuu HALF-ADDER -kytkentään. Sanallisena huomiona mainittakoot, että käytettäessä useampaa perusporttia kutsutaan niiden yhdistelmää kytkennäksi.

Nyt huomataan tarve lippurekisterille, sillä HALF-ADDER -kytkentä vaatii toimiakseen carry-lipun. Carry-lippu on vain tapa merkitä ylivuotobitti muistiin seuraaville porteille tulosignaaliksi. Suorin suomennos carry-lipulle voisi olla muistilukulippu.
HALF-ADDER -kytkentä asettaa carry-lipun olemaan tulosignaalien and ja
HALF-ADDER -portin lähtö on xor tulosignaaleista.

```
function half_adder(a,b){
   set_flag('c' , and(a,b) );
   return xor( a , b );
}
```

FULL-ADDER

Kuten xor-funktion suunnittelussa huomattiin, ohjelmasta tulee kamalan sekava ilman muuttujien käyttöä. Xor-funktion tapauksessa sekavuus oli vielä harmaalla alueella, mutta FULL-ADDER -kytkennän simuloinnissa muuttujien poistaminen olisi tehnyt koodista mahdotonta ylläpitää.

```
function full_adder(a,b){

   // we need to ad a+b+carry, but then carry would change.
   // let's store the incoming carry
   var tmp_carry_in = get_flag('c');

   // Now we can add a and b
   var tmp_first_add = half_adder(a,b);

   // but hey, we need this carry too
   //var tmp_carry_2 = global_flag_carry;
   var tmp_carry = get_flag('c');

   // first carry and a+b is final output
   var tmp_out = half_adder( tmp_first_add, tmp_carry_in);

   // but before returning, we need to edit carry
   //global_flag_carry = or( tmp_carry_2 , global_flag_carry);
   set_flag('c' , or( tmp_carry , get_flag('c')) );

   // Now that was ugly.
   return tmp_out;

}
```

Full-adder -funktio vaikuttaa toimivan, yhteenlaskut bittien kesken toimivat ja carry-lippu asettuu oikein. Myös seuraavaa bittiä laskettaessa carry-lippu otetaan huomioon oikein. Tämä tarkoittaa, että kaikki yhden bitin operaatiot ovat nyt valmiit.

PROSESSORIN VALINTA

Prosessorin hyödyllisyyden määrittää se, mitä se osaa tehdä. Suurin rajaava tekijä on bittisyys ja käskykanta. Tämän prosessorin kooksi vakiintui 4 bittiä, sillä se on tarpeeksi laaja toiminnan testaamiseen ja sen laajentaminen nykyaikaisen 64-bittisen prosessorin kokoiseksi on mahdollista. Nyt luettavuuden lisäämiseksi nelibittisyys sopii mainiosti ja on tarpeeksi kattava testaamaan toimintaa.

Suurin rajaava tekijä on muisti. Näin pienellä muistilla voidaan ohjata vain 16 eri muistipaikkaa, joten päädyttiin rakenteeseen, jossa muistia osoitetaan 8 bitillä, siis sivuutetaan muisti 16:een sivuun, joista jokainen on 16-paikkainen.

Itse muisti oli alunperin tarkoitus rakentaa kiikuista, mutta pitkän suunnittelun ja ajatusvirheiden seurauksena päädyttiin ottamaan muuttujajono kuvastamaan muistia. Tätä verrattakoot ulkoiseen muistipiiriin, kuten vanhemmissa prosessoreissa.

Käskykannassa on otettu paljon vaikutteita Commodore 64:sen vastaavista komennoista, kuten Jim Butterfield:in kirjassa Machine Language ne on esitetty. Koko käskykanta komeudessaan ei vaadi paljoa tilaa, joten esitettäkööt se tässä välissä.

TAULUKKO 2. käskykanta

```
nop                   No OPeration, Don't do anything, for timings etc.
jmp 0001 0000         JuMP, Program_Counter/next executed command = 0001 0000
get 0000 0101         GET value of to r0, r0 = value of 0000 0101  ( r0 =  I/O 0-3 )
put 0000 0010         PUT value of r0 to, r0's value is copied to 0000 0010 (DDR 0-3)

ldr 0010 0110         LoaD to Register, r2 = 6
crr 0110 0011         Copy Register value to another Register, r6 = r3
and 0001 0010         logical AND, r0 = r1 and r2
lor 0100 0110         Logical OR, r0 = r4 or r6

not 0101 0011         logical NOT, r5 = not r3
xor 0010 0011         eXclusive OR, r0 = r2 xor r3
lsh 0001 0001         Left SHift,  r1 = r1 << 1
rsh 0110 0101         Right SHift, r6 = r6 >> 5

add 0100 0010         full ADDer, r0 = r4 + r2
neg 0001 0100         NEGate, r1 = one's complement of r4
beq 0001 0000         Branch if EQual to zero, If zero flag =1 , jmp 0001 0000
bne 0010 1111         Branch if Not Equal, if zero flag =0 , jmp 0010 1111
```

Ensimmäinen komento, nop, voi toimia esimerkiksi ajoituksien kanssa. Hidas muistipiiri voi tarvita montakin nop-komentoa lukukäskyn ja varsinaisen lukemisen väliin. Tämän voisi toteuttaa toki muullakin komennolla, mutta toistaiseksi se on jäänyt. Ainut mitä nop tekee, on kasvattaa ohjelmalaskuria yhdellä. Ohjelmalaskuri on vain osoitin, joka kertoo prosessorille, mikä käsky suoritetaan seuraavaksi.

Komento jmp muuttaa ohjelmalaskurin (eng. Program counter) arvoa ja siis mahdollistaa alkeellisella tasolla aliohjelmien tekemisen. Esimerkiksi komento

```
jmp 1010 1001
```

muuttaa muistisivurekisteriin (siis vaihtaa muistisivua) arvon 10 ja muistisivun sisällä käskyyn 9. Seuraavaksi otetaan muut hyppykäskyt, beq ja bne. Beq toimii kuten jmp, mutta jos ja vain jos nollalippu on asetettu. Liput käsitellään myöhemmin, joten jääkööt se tässä vain maininnaksi. Tämä mahdollistaa ehtolausekkeet, eli prosessori ei toista samaa käskylitaniaa aina, vaan ohjelman suoritusta voidaan muuttaa tilanteen mukaan.

Esimerkiksi alla on komento, joka voi esimerkiksi aloittaa ohjelman suorituksen alusta, jos nollalippu on asetettu.

```
beg 0001 0000
```

Periaatteessa käsky bne on nop:in tavoin turha, mutta se on kehitetty ohjelmointia helpottamaan. Tämä käsky muuttaa ohjelmalaskurin arvoa jos ja vain jos nollalippu ei ole asetettu.

Muistinkäsittelyyn on otettu get- ja put-komennot. Tässä ne eivät ole pinokomentoja, kuten useissa muissa prosessoreissa, vaan niillä kirjoitetaan ensimmäisen rekisterin arvo muistipaikkaan tai luetaan muistipaikan arvo ensimmäiseen rekisteriin. Rekisterit käydään läpi myöhemmin, mutta niitä voi ajatella prosessorin sisäisinä muuttujina. Ldr-komento yksinkertaisesti lataa rekisteriin minkä tahansa nelibittisen arvon. Crr kopioi arvon rekisteristä rekisteriin. Ohjelmaesimerkki joka käyttää näitä on alempana.

```
get 0110 1111   , haetaan muistiosoitteen 6f sisältö rekisteriin r0
crr 0001 0000   , kopioidaan rekisterin r0 arvo rekisteriin r1
ldr 0010 0001   , Ladataan rekisteriin r0 arvo 1
add 0001 0010   , lasketaan rekisterien r1 ja r2 arvot yhteen
                 , ja talletetaan se rekisteriin r0
put 0110 1111   , tallennetaan summa takaisin muistipaikkaan,
                 , josta edellinen arvo otettiin.
```

Lyhyesti selitettynä kyseinen ohjelma siis lisää tietyssä muistipaikassa olevaa arvoa yhdellä. Loogiset operaatiot and, l(ogical)or, not ja xor toimivat kuin yksibittiset versionsa. Nimeämiskäytäntönä käskykannan tekijällä oli mielenkiintoinen idea lor:in suhteen, yleisesti vastaavaa komentoa kutsutaan or-komennoksi, mutta tässä se on lor. Shiftaus-komennot, eli kertominen ja jakaminen kahdella tarvitaan myös. Add on full-adder, eli yhteenlaskutoimitus, jossa kahden rekisterin yhteenlasku talletetaan ensimmäiseen rekisteriin. Sivuhuomiona mainittakoot, että komento add 0000 0000 siis vastaa komentoa nop. Siinä luetaan ensimmäisen rekisterin arvo, lisätään siihen nolla ja tallennetaan takaisin. Viimeinen käsiteltävä komento, neg, kääntää rekisterin bitit johonkin toiseen rekisteriin. Yksi etu nop-komennolla on, sillä muisti alkaa tilanteesta, jossa kaikki arvot ovat nollattuina, siis kohdat, joissa ei ole ohjelmaa, voi jättää olemaan

ja niiden yli vain hypätään. Tämä tekee myös konekielikoodista hieman luettavampaa. Huomionarvoista lienee myös se, että nop vie vain yhden muistipaikan, kun add 0000 0000 vie kolme.

REKISTERIT

Rekistereillä ohjataan prosessorin toimintaa. Tärkeimmät rekisterit ovat ohjelmalaskuri ja lippurekisterit, sillä näillä määrätään mitä komentoa prosessori suorittaa. Ohjelmalaskuri selitettiin kattavasti aiemmin, joten jatketaan lippurekisterillä.

Neljään bittiin mahtuu 4 lippua. Carry-lippu eli ylivuotolippu asettuu, mikäli esimerkiksi lukuun 15 lisätään yksi. Tapahtui ylivuoto, sillä neljään bittiin ei mahdu enempää kuin kuusitoista lukua, nollasta viiteentoista. Aritmethic ja Interrupt, eli A- ja I-liput eivät ole prosessorin tässä versiossa käytössä, vaikka ideana oli mahdollistaa keskeytykset I-lipulla. A-lippu omasi myös idean, joka on varmaankin jo hukkuneessa paperissa selitettynä. Zero eli Z-lippu, eli nollalippu kertoo mikäli laskutoimituksesta seurasi nolla. Komennot bne ja beq tarkistavat tätä lippua hyppykäskyissään. Esimerkkikäyttötapa on hypätä muutama käsky taaksepäin, jossa laskurin arvoa vähennetään yhdellä ja kun vähennyksen tulosena syntyy 0, nollalippu nousee ja ohjelmarakenteessa jatketaan eteenpäin.

Prosessori ilman I/O-rakenteita olisi liian teoreettinen. I/O-rakenne tarkoittaa liitäntäpinnejä, joilla jokin ulkoinen asia voi ohjata prosessorin toimintaa ja jota kautta prosessori itse ohjaa ulkomaailmaan signaaleja. Nollasivulla oli vielä muistipaikkoja vapaana, joten näille uhrattiin 6 muistipaikkaa. DDR, eli datankulkusuuntarekisteri kertoo luetaanko vai kirjoitetaanko kyseisestä pinnistä. Jos prosessorilta halutaan asettaa I/O-pinni 1, tarvitsee kyseinen pinni asettaa kirjoitusasentoon, jonka jälkeen pinni asetetaan. Syntaksiksi DDR-rekisteriin otettiin PIC-prosessorien syntaksi, I/O näyttää samalta, kuin 1/0, joka taas vastaa englanninkielisiä termejä In/Out, eli luku/kirjoitus. Siispä 1 on luku ja 0 kirjoitus.

```
ldr 0000 1101 , bitti 1 lähtöbitiksi, eli out-asentoon
put 0000 0011 , talletetaan r0:n sisältö muistipaikkaan 3, jossa
on DDR-rekisterit
put 0000 0110 , ja sama myös I/O rekisteriin
```

Vastaavasti jos asetetaan bitit 4-11 lukemaan tuloja, käy se esimerkiksi näin.

```
ldr 0000 1111 , kaikki in-asentoon
put 0000 0100 , ja arvo tallennetaan DDR 4-7 muistipaikkaan
put 0000 0101 , kuten myös DDR 8-11 paikkaan.
```

Arvoja voidaan lukea get-komennolla seuraavasti

```
get 0000 0111 , luetaan I/O 4-7
crr 0000 0001 , tallennetaan luettu arvo rekisteriin r1
get 0000 1000 , luetaan I/O 8-11
```

Varsinaiset viisi toimintarekisteriä ovat nimetty AVR-prosessorin mukaan r0:sta r4:ään. Rekisteri r0 käyttäytyy kuten accumulator-rekisteri Commodore 64:ssa, johon tallentuu mm. get-komennon lopputulos ja add-komennon summat, mutta jota voi käyttää kuten muitakin rekistereitä. Nämä ovat siis kuin prosessorin sisäiset muuttujat.

Viimeiset kaksi tavua on varattu keskeytysten hallintaan, mutta koska näitä ei ole toteutettu, ne ovat käyttämättömiä ominaisuuksia. Näihin pääsee käsiksi get- ja put-komennoilla, joten eivät ole aivan hukkaan heitettyä muistia.

MUISTIN RAKENNE JA KÄSKYKANTA

Rekisterit sijaitsevat muistissa, tarkemmin sanottuna muistin ensimmäisellä sivulla. Tätä sivua merkataan ensimmäisellä luonnollisella kokonaisluvulla, nollalla, joten sitä kutsutaan nollasivuksi. Muisti on tässä Von Neumann:in nimellä kulkevaa, eli

ohjelmamuisti ja datamuisti asuvat sulassa sovussa keskenään. Tämä mahdollistaa ajon aikana muokattavan ohjelman, mutta käytännössä tätä ei kannata harrastaa ilman, että on valmis debuggaamaan tuntikaupalla muutamaa komentoa. Seuraavalta sivulta, jota prosessori käsittelee sivuna 1, alkaen ohjelma on täynnä nop-komentoja, joiden päälle oma ohjelma ohjelmoidaan seuraavan viidentoista sivun ajan. Hyvä idea on hypätä viimeisen sivun lopussa nollasivun yli, ettei prosessori ala suorittamaan ohjelmalaskuria, lippuja ja rekistereiden sisältöä komentoina.

Alkeistason logiikkaportit ovat valmiina, sekä prosessorin käskykanta on lukittu. Filosofinen toiminta tulisi yhdistää nyt alkeistason logiikkaportteihin.

Nop nosti ohjelmalaskuria yhdellä. Ohjelmalaskuri majaili nollasivun kahdessa ensimmäisessä muistipaikassa ja arvon nostaminen onnistuu Full-adder -portilla. Ohjelmalaskurin nostaminen vaihdettiin simulaattorin tehtäväksi, koska se tapahtuisi jokaisessa komennossa joka tapauksessa ainakin kerran.

Simulaattorissa on siis funktio nimeltä "simulate_one_tick()", joka nimensä mukaisesti suorittaa ohjelmaa yhden komennon verran. Tämän funktion alkuvaiheessa kutsutaan toista funktiota nimeltä "increment_program_counter()", joka nimensä mukaisesti nostaa ohjelmalaskurin arvoa yhdellä. Lopullinen ohjelmalaskurin nosto tapahtuu siis jälkimmäisessä funktiossa, jonka sisältö on alla katkelmana.

```
function increment_program_counter(){
   var mem_h = get_memory( 0);
   var mem_l = get_memory( 1);

   mem_l = four_bit_full_adder(mem_l, [0,0,0,1]);

   set_flag('a',true);
   mem_h = four_bit_full_adder(mem_h, [0,0,0,0]);

   set_memory(0,mem_h);
   set_memory(1,mem_l);
}
```

Tässä luetaan alempi ja ylempi osa ohjelmalaskuria vastaaviin muuttujiin, suoritetaan nelibittinen full-adder kummallekkin ja talletetaan arvot takaisin.

Komento jmp toteutuu melko lailla samoin tavoin kuin edellinen, mutta muistista ei kysytä alkuperäisiä arvoja, vaan päälle sijoitetaan seuraavat arvot. Funktio on lyhykäisyydessään tällainen.

```
function is_jmp(addr1, addr2){
   set_memory( 0 , addr1);
   set_memory( 1 , addr2);
}
```

Ohjelmaa tehdessä nähtiin oleelliseksi lisätä instruction_set.js tiedoston funktioiden nimiin "is_" alkuun erottamaan ne mahdollisesti joistain muista funktioista, joita ei koskaan tullutkaan. Tästä olisi voinut tulla ongelmia, joten varatoimi jäi voimaan.

Get ja put toimivat edellisen tavoin, mutta muistipaikka ei ole vain ohjelmamuistin osoite, vaan mikä tahansa osoite. Get-komennon tapauksessa tulee huomioida, että mikäli r0 saa arvon nolla, nollalipun pitää nousta ylös.

```
function is_get(addr1,addr2){
   set_memory( r0_addr  , get_memory( memory_address(addr1,addr2)) );
   if( _to_dec(get_memory(r0_addr)) == 0 )
        set_flag('z', true);
   else
        set_flag('z', false);
}
```

Huomion arvoista on myös, että put tallentaa vain yhteen muistipaikkaan, joten set_memory -funktiota kutsutaan vain kerran.

```
function is_put(addr1,addr2){
   set_memory(memory_address(addr1,addr2) , get_memory(r0_addr))
}
```

Hyvin samaan linjaan edellisten kanssa asettuvat viimeiset kaksi komentoa, mutta ne tarkastelevat nollalipun asemaa ennen varsinaista hyppyä. Ratkaisuissa päädyttiin käyttämään jo valmista hyppykäskyä mikäli nollalippu on tai ei ole asetettu, komennosta riippuen.

```
function is_beq(addr1, addr2){
   if ( get_flag('z') != 0){  // equal to zero => !Z = 1
      is_jmp(addr1, addr2);
   }
function is_bne(addr1, addr2){
   if ( get_flag('z') == 0){  // not equal to zero
      is_jmp(addr1, addr2);
   }
}
```

Ldr-komento muuttaa jonkun rekisterin muistipaikkaa tietyksi arvoksi. Javascriptin heikkous ilmeni erittäin hyvin tätä funktiota suunnitellessa; Kun luot muuttujan, jonka arvo on 0, on se Javascriptin mielestä merkkijono, jossa on alkiona merkki 0. Tämä saatiin kierrettyä asettamalla plusmerkki kummankin yhteenlaskettavan eteen ja niiden väliin, kuten alla olevan katkelman toisella rivillä näkyy. Kun rekisteriin ladataan arvo, tarvitsee nollalippu asettaa, mikäli ladattava arvo oli nolla.

```
function is_ldr(reg, value){
   set_memory( +r0_addr + +_to_dec(reg)  ,value  );
   if ( _to_dec(value) == 0 )
      set_flag('z' , true);
   else
      set_flag('z', false);
}
```

Crr toimii samalla tapaa kuin edellinen, mutta valmiin arvon sijaan funktiolle annetaan arvona rekisterin numero, josta arvo kopioidaan. Alla olevan katkelman toinen rivi alkaa vaikuttaa jo painajaismaiselta, mutta seuraa edellisen logiikkaa niin hyvin, ettei riviä kirjoittajan mielestä tarvinnut pätkiä pienemmiksi.

```
function is_crr(to,from){
   set_memory( +r0_addr + +_to_dec(to)  ,
               get_memory( +r0_addr + +_to_dec(from)  )
              );
   if( _to_dec(get_memory(+r0_addr + +_to_dec(to))) == 0 )
      set_flag('z', true);
   else
      set_flag('z', false);
}
```

And palaa takaisin alkeistasolle käyttäen and-porttia neljästi. Logiikkakomentojen lopputulos päädyttiin palauttamaan rekisteriin r0. Tämä funktio päätettiin pätkiä muuttujiin koodin ulkoasun yksinkertaistamiseksi ja luettavuuden parantamiseksi. Rekisterin arvot siis luetaan muuttujiin, muuttujien alkioille tehdään andit yksi kerrallaan ja lopputulos palautetaan muistiin rekisteriin r0. Tämän jälkeen vielä muistetaan tarkistaa, tarvitseeko nollalippua muuttaa.

```
function is_and(reg1, reg2){
   var tmp1 = get_memory( r0_addr + _to_dec(reg1));
   var tmp2 = get_memory( r0_addr + _to_dec(reg2));

   var out = [ and( tmp1[0] , tmp2[0]) ,
               and( tmp1[1] , tmp2[1]) ,
               and( tmp1[2] , tmp2[2]) ,
               and( tmp1[3] , tmp2[3]) ]

   set_memory( r0_addr  , out );
   if( _to_dec(get_memory(r0_addr)) == 0 )
      set_flag('z', true);
   else
      set_flag('z', false);
}
```

Lor, nimi tulee termistä "logical or". Tai-operaatioita on kaksi erilaista. Ensimmäinen on luonnollinen or, eli xor. Esimerkiksi jos kassahenkilö vaatii pankki- tai luottokortin, ei hän oleta saavansa kahta korttia, kuten loogisen tain tapauksessa. Toinen syy nimen käyttöön oli saada kaikki komennot kolmimerkkisiksi, jolloin ohjelmakoodi näyttää tasaisemmalta, ilman että or erottuu joukosta ja jolloin keskittymiskyky pysyy oleellisessa.

TAULUKKO 3. Looginen or (lor) ja luonnolinen or (xor)

a	b	lor	xor
0	0	0	0
0	1	1	1
1	0	1	1
1	1	1	0

Toteutus lor- ja xor-operaatioille on kuten and, ainoastaan lopputuloksen tekemiseen käytetään eri operaatiota.

```
function is_lor(reg1, reg2){
   var tmp1 = get_memory( r0_addr + _to_dec(reg1));
   var tmp2 = get_memory( r0_addr + _to_dec(reg2));
   var out = [ or( tmp1[0] , tmp2[0]) ,
            or( tmp1[1] , tmp2[1]) ,
            or( tmp1[2] , tmp2[2]) ,
            or( tmp1[3] , tmp2[3]) ]
   set_memory( r0_addr  , out );

   if( _to_dec(get_memory(r0_addr)) == 0 )
      set_flag('z', true);
   else
      set_flag('z', false);
}



function is_xor(reg1, reg2){
   var tmp1 = get_memory( r0_addr + _to_dec(reg1));
   var tmp2 = get_memory( r0_addr + _to_dec(reg2));

   var out = [ xor( tmp1[0] , tmp2[0]) ,
            xor( tmp1[1] , tmp2[1]) ,
            xor( tmp1[2] , tmp2[2]) ,
            xor( tmp1[3] , tmp2[3]) ]



   set_memory( r0_addr  , out );
   if( _to_dec(get_memory(r0_addr)) == 0 )
      set_flag('z', true);
   else
      set_flag('z', false);
}
```

Not on hieman poikkeava tällä prosessorilla. Not on unäärioperaatio, eli yhden arvon ottava toiminto. Jotta not saataisiin muiden kanssa samanpituiseksi, päätettiin ottaa not-funktion parametreiksi muistiosoite. Tämä tarkoittaa, että not-operaatiolla voidaan korvata osittain get-operaatio, mikäli jälkikäteen vielä muistetaan tehdä uusi

not-operaatio rekisterin r0 muistipaikalle, sillä kaksi not-operaatiota kumoavat toisensa ja lopputulos tallentui tällä prosessorilla rekisteriin r0.

```
function is_not(addr1, addr2){
   var tmp = get_memory( memory_address( addr1, addr2));
   tmp = [not(tmp[0]) ,
      not(tmp[1]) ,
      not(tmp[2]) ,
      not(tmp[3]) ];

   set_memory( r0_addr  , tmp  );
}
```

"Shiftaus"-operaatioissa rekisterin sisältö luetaan kokonaisluvuksi, shiftataan suuntaansa ja palautetaan takaisin paikallensa. Muistetaan taas tarkistaa nollalippu. Ylivuotoa ei ole huomioitu, joten tämä tulee huomioida ohjelmallisesti.

```
function is_lsh(addr, amount){
   var tmp = get_memory(r0_addr + _to_dec(addr));
   tmp = _to_bin(_to_dec(tmp) << _to_dec(amount));

   if( _to_dec(get_memory(r0_addr + _to_dec(addr))) == 0 )
        set_flag('z', true);
   else
        set_flag('z', false);


   set_memory(r0_addr + _to_dec(addr), tmp);
}

function is_rsh(addr, amount){
   var tmp = get_memory(r0_addr + _to_dec(addr));
   tmp = _to_bin(_to_dec(tmp) >> _to_dec(amount));

   if( _to_dec(get_memory(r0_addr + _to_dec(addr))) == 0 )
        set_flag('z', true);
   else
        set_flag('z', false);

   set_memory(r0_addr + _to_dec(addr), tmp);
}
```

Add-funktio ottaa kaksi rekisteriä, tekee niille nelibittisen full-adder-operaation ja palauttaa lopputuloksen rekisteriin r0.

```
function is_add(addr1, addr2){
//   set_memory( r0_addr , four_bit_full_adder( a,b) );
   var tmp1 = get_memory(+r0_addr + +_to_dec(addr1));
   var tmp2 = get_memory(+r0_addr + +_to_dec(addr2));

   var out = four_bit_full_adder( tmp1, tmp2);

   if ( _to_dec(out) == 0 )
        set_flag('z' , true);
   else
        set_flag('z', false);
   set_memory( r0_addr , out);
}
```

Jäljellä viimeinen operaatio, luvun negation tekeminen.

```
function is_neg(from, to){
   var tmp_from = get_memory(+r0_addr + +_to_dec(from));
   var out = [ full_adder(tmp_from[3] ,1) ,
            full_adder(tmp_from[2] ,0) ,
            full_adder(tmp_from[1] ,0) ,
            full_adder(tmp_from[0] ,0) ].reverse();
   set_memory(+r0_addr + +_to_dec(to) , out);
}
```

SIMULAATTORI

Jokaisen käskykannan jäsenen ollessa nyt olemassa funktiona, rakennetaan simulaattori, jossa näitä kutsutaan. Ulkoasu simulaattorilla on varsin kankea, kuten kuvasta 3. näkyy.

```
0100
0000
1111
```

Program

Simulate one Cycle

Simulate Cycles

Update values

PC	34
Used memory	47
Memory size	256

PC high	false,false,true,false
PC low	false,false,true,false
Flags (caiz)	false,false,false,true
DDR 0-3	1,1,1,1
DDR 4-7	0,0,0,0
DDR 8-11	false,false,false,false
I/O 0-3	false,false,false,false
I/O 4-7	false,false,false,false
I/O 8-11	false,false,false,false
R0	false,false,false,false
R1	false,false,false,false
R2	false,false,false,false
R3	false,false,false,false
R4	false,false,false,false
Interrupt high	false,false,false,false
Interrupt low	false,false,false,false

```
16:  0,1,0,0
17:  0,0,0,0
18:  1,1,1,1
19:  0,0,1,1
20:  0,0,0,0
21:  0,0,1,1
22:  0,1,0,0
23:  0,0,0,0
24:  0,0,0,0
25:  0,0,1,1
26:  0,0,0,0
27:  0,1,0,0
28:  0,0,1,0
29:  0,0,0,0
30:  0,1,1,0
31:  1,1,1,1
32:  0,0,1,0
33:  1,0,0,1
34:  0,0,0,0   <--- PC
35:  0,0,0,0
36:  0,0,0,0
37:  0,0,0,0
38:  0,0,0,1
39:  0,0,0,1
40:  1,1,0,0
41:  0,0,1,1
42:  0,0,0,0
43:  0,1,1,1
44:  0,0,0,1
45:  0,0,0,1
46:  1,1,0,0
```

Kuva 3. Simulaattorin ulkoasu

Simulaattori on siis rakennettu muutamasta osasta. Aluksi on ohjelman syöttökenttä vasemmalla yläkulmassa, johon assemblerilla muodostettu binäärikoodi syötetään. Nappi "Program" ohjelmoi prosessorin, eli muuttaa muistin arvot ohjelman mukaisiksi. Seuraavaksi ohjelma tekee joko yhden komennon simuloinnin napilla "Simulate one cycle" tai niin monta komentoa, kuin napin "Simulate Cycles" edessä olevaan numerokenttään on syötetty numerona. Update values on ohjelmassa mukana, mikäli oikealla näkyvässä Firefox-selaimen javascript-tulkissa muutetaan arvoja. Tämä oli alunperin vain suunnittelutyökaluna, mutta se nähtiin veikeäksi lisäksi ja jäi lopulliseen tuotokseen mukaan.

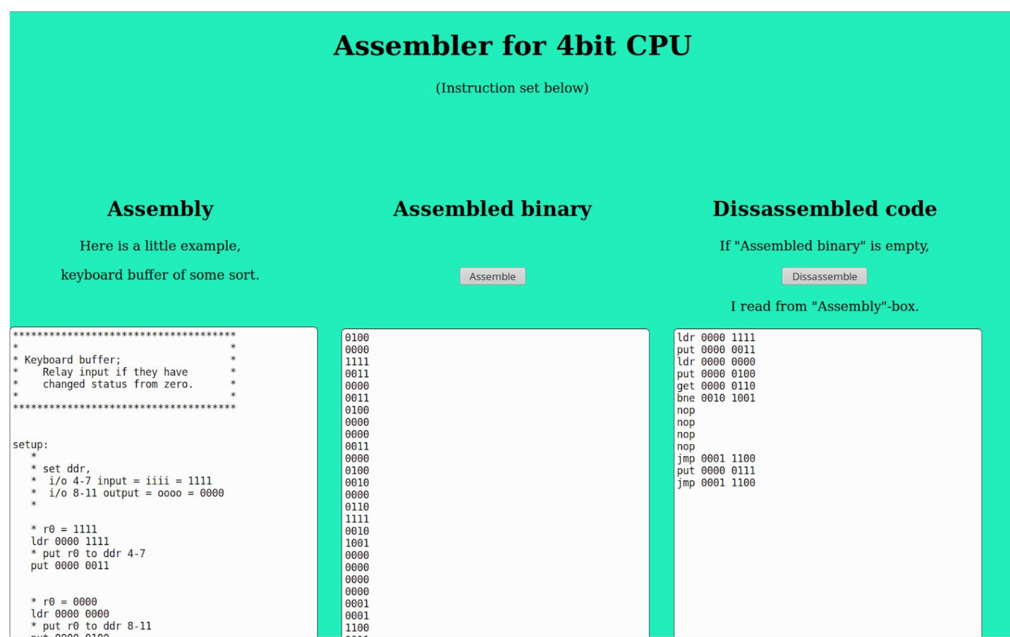
Tämän jälkeen vasemmalla alhaalla on ohjelmalaskuri, muistin käyttö ja nollasivun arvot. Keskellä sivua on muistialue, jota suoritetaan ja sen perässä näkyy nuolena ohjelmalaskurin osoittama muistiosoite. Javascript-konsoliin tulostetaan mikä komento

oli ja mitä sille annettiin arvoina.

ASSEMBLER

Simulaattori siis vaati toimiakseen binäärikoodin, joten tarvitaan assembler, joka muuttaa korkean tason käskyt nop:ista bne:n numeroarvoksi. Myöhemmin lisättiin myös tuki hyppykohdille. Tämä mahdollistaa aliohjelmat ilman, että ohjelmoijan tarvitsee laskea aliohjelman pituutta ja siten hyppykohtien osoitteita, vaan assembler hoitaa sen.

Koska simulaattori oli testaajien mielestä kovin väritön ja riisuttu, päätettiin assembleriin lisätä väriä. Mikä väri kuvastaisi assemblerin toimintaa? Punainen ei sovi pääväriksi, sillä tarkoitus on saada binäärilukujen vastapainoksi luonnollisempaa väriä. Päädyttiin ottaamaan pääväriksi metsästä tuttu vihreä, lisätä siihen jonkun verran sinistä järvistä ja taivaasta sekä ripaus punaista tuomaan tasapainoa. Lopullinen tuotos näkyy kuvassa 4, ja selityksen jälkeen testihenkilöt vaikuttivat tyytyväisiltä värin määrään.



KUVA 4. assembler

Käyttöliittymässä otettiin huomioon käyttöliittymäteorian alkeita, joita opittiin Jenifer Tidwell:in kirjasta "Designing Interfaces". Yksi pääteema on yksinkertaisuus. Yksinkertaisuus on taattu antamalla esimerkki assembly-nimiseen tekstisyöttökenttään.

Seuraavaksi käyttäjän huomio kiinnittyy vasemmalta oikealle isoimpiin teksteihin, eli "Assembled Binary" lauseeseen, jolloin käyttäjä huomaa "Assemble"-nimisen napin. Hän siis seuraa alitajuisesti ohjelman kulkua ja painaa nappia huomatakseen, että tämä assembleri antoi korkean tason assemblystä matalan tason binäärikoodin. Seuraavaksi hän suunnittelun mukaan huomaisi vielä "Disassembled Code" otsikon, mutta napin ympärillä on maininta, ettei mitään järkevää tapahdu, ellei assemblyä ole suoritettu ja siten konekielikoodia tehty. Nyt napin "Disassemble" painaminen saa aikaan binäärikoodin muuttumisen takaisin korkeamman tason koodiksi, josta on kommentit ja hyppymerkit kadonneet käännöksen seurauksena.

Käyttäjä nyt suunnitelman mukaan palaa isoimpaan otsikkoon etsimään mitä tekee tällä tiedolla ja havaitsee otsikon alla olevan "Instruction set below"-lauseen suluissa. Sulut ovat tärkeitä, sillä niiden takia käyttäjä hyppäsi tämän lauseen yli aikaisemmin. Nyt käyttäjä huomaa että sivu jatkuu alaspäin ja täällä on käyttöliittymäteorian sisällönpiilottamissäännön mukaan ohjeita. Sisällönpiillottaminen on helpoin ymmärtää katsomalla esimerkiksi kuvan 5 television kaukosäädintä.

KUVA 5 . Kaukosäädin, jossa lisätoiminnot ovat luukun alla. (Alzproducts, 2017)

Kun käyttäjä ottaa kaukosäätimen käteensä, hän näkee kanavien ja äänentason säätönapit, sekä virran kytkemiseen tarvittavan kytkimen. Hetken katsomisen jälkeen havaitaan taustavärillä piilotettu "Mute"-näppäin, jolla televisio saatetaan äänettömään tilaan. Kun tarvetta on esimerkiksi vaihtaa kanavasta 2 kanavaan 14, voi käyttäjä halutessaan painaa kaksitoista kertaa ylöspäin-nuolta nähden television vaihtavan

kanavaa jokaisella napinpainalluksella, eli käyttäjä saa jatkuvaa palautetta toiminnastaan. Toinen vaihtoehto on tehokäyttäjille oikotie, eli avataan luukku ja painetaan numeroita 1 ja 4, sekä odotetaan hetki ja television kanava vaihtuu. Samalla teorialla nyt tehokäyttäjä päätyy lukemaan käskykantaa tarkemmin.

ASSEMBLERIN TOIMINTA

Tekstikenttään siis syötetään assembly-kielinen ohjelmakoodi, joka tottelee tämän prosessorin käskykantaa. Syntaksi on hyvin tyypillinen lukuunottamatta kommenttiriviä, joka merkitään epätyypillisesti tähdellä. Hyppykohtien nimet loppuvat kaksoispisteeseen ja muut rivit tulkitaan käännettäviksi komennoiksi.

Assemblerissa tämä on tarkoitus saattaa binäärikoodiksi, jonka voi kopioida simulaattoriin. Luetaan siis tekstikentän sisältö ja luodaan jokaisesta rivistä oma muuttuja muuttujajonoon, jotta rivejä voidaan käsitellä yksittäin. Muokataan lopputulos assemblereille tyypillisesti olemaan välittämättä kirjainkoosta ja poistetaan ylimääräiset rivinvaihdot, jolloin yhdellä rivillä on yksi komento.

Seuraavaksi luodaan dynaamisille hyppykohdille muuttujajonot rivinumeroille ja niitä vastaaville nimille. Hyppykohtien nimet tunnistetaan sillä, että rivin viimeinen merkki on kaksoispiste. Kun rivi, joka päättyy kaksoispisteeseen, on löydetty, talletetaan sen rivinumero ja sitä vastaava nimi ilman kaksoispistettä luotuihin muuttujajonoihin.

Seuraavaksi erotellaan yhden muistipaikan kuluttava nop ja kolmen muistipaikan kuluttavat komennot niin, että rivinumerot pätevät. Huomataan muistipaikan ja muistisivun väliseksi yhteydeksi seuraava kaava.

$$\frac{(arvo - (arvo \bmod 16))}{16}$$

Kaava 1. Muistiosoitteen märittely

Kun hyppykohdat on määritelty, assembleri kirjoittaa ne ylös ja poistaa ne koodista. Niitä kun ei prosessori voi suorittaa vaan ne ovat ohjelmoijan apuna. Tässä käytetään vanhan ohjelman funktiota "onko_jonossa_arvoa()", joka nimensä mukaisesti palauttaa totuusarvon sen mukaan, oliko halutussa jonossa arvo.

Nyt jäljellä on pelkät komennot ilman hyppykohtia, joten binäärikoodin generointi on vain taulukosta arvon katsominen ja sen korvaaminen vastaavalla. Esimerkiksi kun rivillä on sana nop, korvataan se binäärillä 0000 ja niin edelleen käskykannan mukaisesti. Seuraavaksi kirjoitetaan valmis binäärikäännös seuraavaan tekstikenttään, josta sen voi käyttäjä kopioida.

Binäärikoodin takaisinkääntäjä, disassembler, ottaa binäärikoodin ja muuttaa sen vastaavaksi assemblykoodiksi. Tämä on helpompi toteuttaa, kun hyppykohtia ei ole, mutta antaa ohjelmalle suuren lisäarvon.

Ohjelman rakenne noudattaa samaa kaavaa, luetaan binääri, josta otetaan tulostumattomat merkit, kuten välilyönnit ja rivivaihdot pois. Koska syöte oli binääriluku, siis jono merkkejä yksi ja nolla, ei mitään muuta saa löytyä. Mikäli löytyy jokin muu symboli, varoitetaan käyttäjää virheestä.

Komennoilla on kolme osaa, komento ja kaksi parametriä. Poikkeuksena taas nop-komenot, joka käsitellään ensin. Sitten palataan taas lukemaan binääriä vastaava komento ja talletetaan se toiseen merkkijonoon. Lopuksi koko koodi laitetaan näkyville viimeiseen tekstikenttään.

Sivuhuomautuksena mainittakoot C-kielen syntaksiin perustuva kääntäjä, joka tuomittiin melko alkuvaiheessa yliampuvaksi. Kääntäjä osaa erotella eri komennot, tunnistaa puolipilkun joka lopettaa koodirivin ja osaa asettaa muuttujille arvoja, muttei tuota varsinaista koodia lainkaan. Idea tämän toteutukseen löytyy muistilapusta, jossa on if-lauseen toteutus and-portilla seuraavasti

$(ehto \wedge komento) \vee (\neg ehto \wedge komento)$ Kaava 2. If-lause and porttina

Mikäli ehto on epätosi, ensimmäinen komento päätyy nollaksi, kun se ja-operaatiolla nollan kanssa nollautuu. Vastaavasti tällöin toinen komento ja-operoidaan yhdellä, eli nyt suoritetaan jälkimmäinen operaatio. Ehdon ollessa tosi, vastaavasti ensimmäinen komento toteutuu.

POHDINTA

Lähtötilanteessa ymmärrys monimutkaisiin prosessorirakenteisiin oli olematon, eikä assemblerin rakentaminen vaikuttanut lainkaan mahdolliselta. Assembly ja sen ohjelmoiminen oli sentään tuttua, joten päämäärä, se rajapinta joka pitää rakentaa, oli selvä. Kun loogiset operaatiot rakentuivat funktioiden ympärille, simulaattori alkoi vaikuttamaan mahdolliselta, sillä kun nappia painetaan, suoritetaan vain lukemattomia tällaisia alkeislogiikkaporttien funktiokutsuja.

Tyhjän päälle rakentui toimiva kokonaisuus ja simulaattoria testatessa ylpeys saavutuksesta oli sanoinkuvaamaton. Kuitenkin huomio kiinnittyi jatkuvasti suunnittelun monimutkaisuuteen, kun tarkoituksena oli tehdä myös muille ihmisille toimiva työkalu. Ohjelmoinnin tulisi olla mielekästä ja ohjelmistotuotannon terminologialla debuggaus, eli virheiden korjaaminen tulisi olla mahdollisimman helppoa.

Täydellisyyttä ei kuitenkaan tavoiteltu. Esimerkiksi assembler ei osaa kertoa kirjoitusvirheistä, vaan merkitsee ne hyppykohdiksi. Kun hyppykohtaan ei viitata koodin sisällä, hävitetään se käsiteltävästä koodista ennen biteiksi kääntämistä niin, että ohjelmoijalle voi tulla mietittävää pidemmäksikin aikaa. Onneksi assemblerin nettisivulla Javascript-konsoli kertoo paljon tietoja, joihin lukeutuvat myös kyseiset hyppykohdat. Näiden katsominen tosin vaatii tehokäyttäjyyttä, sillä nämä nappulat ovat luukun takana, mikäli jo mainittuun kaukosäätimeen verrataan.

Jatkokehitysideoita voisi listata lukemattomia. C-tyyppisen kääntäjän tai BASIC-tulkin rakentaminen prosessorin ympärille, prosessorin kääntäminen FPGA-prosessorille, esimerkkiohjelmia tyypillisten ohjelmointiongelmien ratkaisuiksi tai vaikka simulaattorin muuttaminen niin, että prosessoreja on useampi ja rinnakkain. Mihin kuriositeetti sitten kannatteleekin.

LÄHTEET

Projektin lähdekoodi, https://github.com/ikosa/4-Bit-CPU, luettu 9.3.2017

Luentomuistiinpanot Boolen Algebraa & Hilateoriaa, Tampereen Yliopiston matematiikan luentoja vuosina 2010-2011

Kelat logiikkaportteina, http://hackaday.com/2013/11/20/making-logic-with-inductors/, Luettu 3.1.2017

MOSFET kuva Cadence-ohjelmalla suunniteltuna, http://pages.cs.wisc.edu/~butts/icons/ex2.gif, luettu 4.3.2017

Vastuksen kuva Cadence-ohjelmalla suunniteltuna, http://webpages.eng.wayne.edu/cadence/ECE6570/res/Layout_of_Resistor.htm, luettu 4.3.2017

Luentomuistiinpanoja transistorin rakenteesta Esa Kunnarin luennoilta vuodelta 2015

Luentomuistiinpanoja Mauri Inhan Digitaalitekniikan peruskurssilta vuodelta 2014.

Jim Butterfield, Machine Language - for the Commodore 64 and other commodore computers, 1984

Designing Interfaces, Jenifer Tidwell, 2010

Yksinkertainen Kaukosäädin, https://www.alzproducts.co.uk/flipper-big-button-remote-control.html, luettu 8.3.2017

LIITTEET

Liite 1. Tiedosto hdl.js

```
//
//   So VHDL is expensive and Javascript is cheap.
//   Could JS offer same functionality?
//
//   All these are what I have understanded, please contact and
//   correct me if I made an error or if I say something that
//   is not understandable.
//
//
//   In the beginning there is NAND-gate, and it was good.
//
//                        VCC
//                         |
//                        | |   // some resistor, try 1k or something
//                        | |
//                         |
//                         *---- OUT
//                         |
//                         N
//          IN 1 ---------P
//                         N
//                         |
//                         |
//                         N
//          IN 2 ---------P
//                         N
//                         |
//                        GND
//
//
//   Only when both of the inputs are 1, the both transistors are
//   closed and circuit looks like this
//
//
//     VCC
```

```
//      |
//      |                                VCC
//     | |                                |
//     | |                               | |     and    OUT ----- GND
//      |               Is equal to      | |
//       *----- OUT                        |       ( so out is GND, 0
)
//      |                                GND
//      |
//     GND                             (This is why
//                                      resistor)
//
//
//   Otherwise It's like this
//
//       VCC                                 VCC
//        |                                   |
//        |                                   |
//       | |           And is like           | |
//        | |                                 | |   ( So out is VCC, 1
)
//        |                                   |
//        *----- OUT                          ------ OUT
//        |
//        x
//
//        x   ( One or both transistor open )
//        |    ( so no connection )
//       GND
//
//
//     Let's make a table out of the logic of this.
//
//     IN 1    IN 2   OUT
//       0      0      1     // Both closed, out = VCC
//       0      1      1     // One of them closed, out = VCC
//       1      0      1     // One of them closed, out = VCC
//       1      1      0     // Both open, out = GND
//
```

```
function nand(a, b){
   // nand as it says, not-and is
   // logically like this
   // Not ( a and b )
   return ! ( a & b ) ;
}



// So there is this mystical AND operation?
// But we only have NAND? Well, javascript doesn't have
// transistors, so I use mystical & and ! symbols
// to fix this error. We cannot be using them from now on!

// So first we need NOT
function not( a ){
   // So we only have NAND. What if we put                IN   OUT
   // a to both inputs? Yep, not is just that.             0    1
   // In real life we could use just 1 transistor.         1    0
   return nand( a, a );
}

// So, let's put inputs trough nand and then not => nnand = and
function and( a, b ){
   //   A   B   OUT
   //   0   0    0
   //   0   1    0
   //   1   0    0
   //   1   1    1
   return ( not ( nand ( a , b ) ) );
}

// For diversity we could need OR-gate,
function or( a, b ){
   //   A   B   OUT
   //   0   0    0
   //   0   1    1
   //   1   0    1
   //   1   1    1
```

```
   // This is tricky. Let's check Boole's algebra.
   // NWM, wikipedia says next, let's test this one
   //
   //  A----NAND
   //            -----NAND---Out
   //  B----NAND
   //
   //  So let's simplify this and make table out of this
   //
   //   1          2         3
   //   A ----NOT--
   //               >--NAND---OUT
   //   B-----NOT--
   //
   //      1      2     3
   //     A B    A B   OUT
   //
   //     0 0    1 1    0
   //     0 1    1 0    1
   //     1 0    0 1    1
   //     1 1    0 0    1   Goddamit, it works!
   //
   return nand( not( a ) , not( b ) );
}

// Nor is needed at memory/RS flipflop
function nor(a,b){
   return not(or(a,b));
}

// Oh, it seems full-adder needs some mystical star-ball-function
// called XOR.
//        ___
//      /  |  \   So is that like 4-bit circle?
//      |--|--|   I just felt like doing some art, sorry.
//      \ _|_ /
//
//  Alright, se we have outputs 0111, 0001 and now we need 0110, so
//  it's like or, but when A=1, B=1 there should be OUT=0.
```

```
//
//
//   A   B   OUT
//   0   0    0
//   0   1    1     XOR
//   1   0    1
//   1   1    0
//
//   Goddamit Wikipedia, this looks awful!
//   Well, I feel like art-y still.
//
//   A---*---------NAND  (2)
//       |       /      \            Looks little like
//        > NAND  (1)   NAND--OUT   bridge rectifier
//       |       \      /
//   B---*---------NAND  (3)
//
//   (3) = NAND B, (1)
//   (2) = NAND A, (1)
//   OUT = NAND (2),(3)
//
//   So  A  B   (1)  (2) (3)   OUT
//       0  0    1    1   1     0
//       0  1    1    1   0     1
//       1  0    1    0   1     1
//       1  1    0    1   1     0   Oh, it works. Awesome
//
function xor(a,b){

   // var tmp_1 = nand( a, b );
   // var tmp_2 = nand( a, tmp_1 );
   // var tmp_3 = nand( b, tmp_1 );
   // return nand( tmp2 , tmp3);

   // we really don't have any memory yet
   // to store these, so let's just make the monster.

   // return nand( tmp2 , tmp3);
   // return nand( nand( a, tmp_1 ) , nand( b, tmp_1 ));
```

```
   return nand( nand( a, nand( a, b ) ) , nand( b, nand( a, b ) ));
}

//
// Finally we can have half adder.
// Well, we needed it anyway, because
// addition doesn't work on javascript.
//
//    0     0      1     1
//  + 0   + 1    + 0   + 1
// -----  ----  -----  ----
//    0     1      1    10
//
//  Okay, if 1+1=10 is made with
//  one bit, it's like 1+1=0, so that
//  is just the XOR-operation, nice!
//
//  But that overflow number is needed
//  if we want precicion. It is
//  historically called Carry, so let's
//  make our first global flag.
//
// var global_flag_carry = 0; This is now in registers.js
//
//  This lives in something called
//  register, we'll come to that
//  someday.
//
//  But how do we do carry? Well,
//  it is needed only when both are 1,
//  otherwise it stays 0.. So and?
function half_adder(a,b){
   //global_flag_carry = and( a , b);
   set_flag('c' , and(a,b) );
   return xor( a , b );
}

//
// That's just half adder.
```

```
// So when there's like 8 bits or
// even more(!), and we add them,
// we need to take count the carry
// of the last bit. When we don't
// it's like job half done, so half adder.
//
// So like this.
//    c     c      c     c
//    0     0      1     1
//  + 0   + 1    + 0   + 1
// -----  ----  -----  ----
//    x     x      x      x
//
// We have 2 scenarios, c=1 and c=0
//    1     1      1     1
//    0     0      1     1
//  + 0   + 1    + 0   + 1
// -----  ----  -----  ----
//   01    10     10    11
//
//    0     0      0     0
//    0     0      1     1
//  + 0   + 1    + 0   + 1
// -----  ----  -----  ----
//    0     1      1    10
//
//  So only when carry was set, it's not
//  just full adder.
//  when carry is set, it's set on 0111, so we need or
//
//  we can use AND to select, let's calculate both
//  and AND the output with carry like this
//  if carry is not set
//     and ( not (carry ) , half_adder(a,b))
//  if carry is set
//     and ( not (carry ) , other_thing)
//  This is gonna be awful, I need to format this..
//
//   or (
```

```
//        and (
//              not (carry ) , half_adder(a,b)
//            ) ,
//        and (
//              not (carry ) , other_thing
//            )
//      )                              OMG, I made lisp =)
//
//   Now, what could the other_thing be?
//   I don't know, I'm going to watch Doctor Who.
//
//   Goddamit, the man under the skull mask was Master!
//   Ok, so I maby tested all the functions so far while
//   that episode. How? Look ./tests file.
//
//   BTW the other thing might be another half adder. Let's start
//   from scratch, lost my focus.
//
//    A and B goes to half adder. The output is... added with carry,
right?
//   Yeah, a+b+carry is out ok.
//
//   The new carry is like 0111, so just or? Let's test.
//
//  So this idea looks like...
//
//   Carry---------------[half out]------OUT
//   A ----[half out]----[ add  c ]...
//   B-----[ add  c ]-----------------OR--Carry
//

function full_adder(a,b){

   // So because these can't happen same time on javascript,
   // I need to do this section by section.
   // If there is easily understandable
   // way to do this without tmp's, please contact me and tell me how!

   // we need to ad a+b+carry, but then carry would change.
```

```
    // let's store the incoming carry
    var tmp_carry_in = get_flag('c');

    // Now we can add a and b
    var tmp_first_add = half_adder(a,b);

    // but hey, we need this carry too
    //var tmp_carry_2 = global_flag_carry;
    var tmp_carry = get_flag('c');

    // first carry and a+b is final output
    var tmp_out = half_adder( tmp_first_add, tmp_carry_in);

    // but before returning, we need to edit carry
    //global_flag_carry = or( tmp_carry_2 , global_flag_carry);
    set_flag('c' , or( tmp_carry , get_flag('c')) );

    // Now that was ugly.
    return tmp_out;

}
```