XILINXIN 7-SARJAN FPGA-PIIRIEN MUISTIRESURSSIT JA NIIDEN KÄYTTÖ SUUNNITTELUSSA

Joona Salmela

Ohjaaja: Jukka Lahti

ELEKTRONIIKAN JA TIETOLIIKENNETEKNIIKAN TUTKINTO-OHJELMA

2019

Salmela J. (2019) Xilinxin 7-sarjan FPGA-piirien muistiresurssit ja niiden käyttö suunnittelussa. Oulun yliopisto, Elektroniikan ja tietoliikennetekniikan tutkinto-ohjelma. Kandidaatintyö, 26 s.

TIIVISTELMÄ

Työssä tutkittiin Xilinxin 7-sarjan FPGA-piirien muistiresursseja ja niiden käyttöä suunnittelussa. Työssä kuvattiin FPGA-piirien perusteet ja niiden käyttökohteet. Ennen Xilinxin muistitekniikoita käytiin läpi tavallisimpien muistien perusperiaatteet ja muistit, jotka olivat käytännöllisimpiä FPGA-piirin kannalta. Sitten tutkittiin kuinka Xilinxin hajautetut ja lohko-RAM-teknologiat toimivat ja kuinka niitä käytetään suunnittelussa Xilinxin toteutuskoodilla sekä RTL-koodilla. Lopuksi tutkittiin Xilinxin 7-sarjan FPGA-piirien muistiresursseja laitetasolla. Työ tehtiin kirjallisuuskatsauksena.

Salmela J. (2019) Xilinx 7-series FPGAs' memory resources and their use in design. University of Oulu, Degree Programme in Electronics and Communications Engineering. Bachelor's degree, 26 p.

ABSTRACT

The subject of this bachelor's thesis was to research Xilinx 7-series FPGAs' memory resources and their uses in design. FPGAs' basics and uses were described. The basics of most practical memories were reviewed followed by Xilinx's 7-series FPGAs' memory technologies distributed RAM and block RAM. Their basic principals and uses were researched and their uses in design in Xilinx instantiations and RTL code. Lastly 7-series FPGA's memory resources were researched. The work was done as a literature review.

LYHENTEET

FPGA	Field Programmable Gate Array, uudelleen ohjelmoitava digitaalinen mikropiiri
ASIC	Application-spesific Integrated Cuircut, sovelluskohtainen mikropiiri
SoC	System on Chip, Elektroninen järjestelmä mikropiirillä
ROM	Read-Only Memory
RAM	Random-Access Memory
LUT	Lookup Table
MUX	Multiplexer
RTL	Register-transfer level
TDP	True dual port
SDP	Simple dual port

SISÄLLYS

TIIVISTELMÄ 2
ABSTRACT 3
LYHENTEET 4
SISÄLLYS 5
1. JOHDANTO 6
2. FPGA-PIIRI 7
2.1. Perusteet 7
2.2. Käyttötarkoitukset 8
3. MUISTITEKNOLOGIAT 9
3.1 Perusteet 9
3.2 ROM 9
3.3 RAM 10
4. XILINXIN MUISTITEKNOLOGIAT 12
4.1 Distributed RAM 12
4.1.2 Distributed RAM ohjelmointi 13
4.2 Block RAM 14
4.2.2 Block RAM ohjelmointi 17
4.3 Read-only memory 18
4.4 Xilinx laitteiden muistiresurssit 18
5. YHTEENVETO 19
6. LÄHTEET 20
7. LIITTEET 21

1. JOHDANTO

Tutkimuksessa selvitetään, minkälaisia muistiresursseja on Xilinxin 7-sarjan FPGA-piireillä ja kuinka niitä käytetään suunnittelussa. Työssä tutkitaan FPGA-piirien perusperiaatteet ja mistä ne koostuvat. Tämän jälkeen tutkitaan perusperiaatteet muisteissa ja tutustutaan Xilinxin muistiteknologioihin. Työssä myös tutkitaan, kuinka näitä resursseja käytetään käytännössä ohjelmistokoodien avulla. Lisäksi tutkitaan piirien muistiresurssien määrää.

Työn tavoitteena on oppia FPGA-piirien perusasiat ja tutkia sen muistiresurssit. Kuitenkin työssä ei tutkita muistien ajoituksia tai virheenkorjaus- ja FIFO- (first in first out) sovelluksia. Työ toteutetaan kirjallisuuskatsauksena, joten tiedot etsitään artikkeleista ja Xilinxin nettisivuilta.

2. FPGA-PIIRI

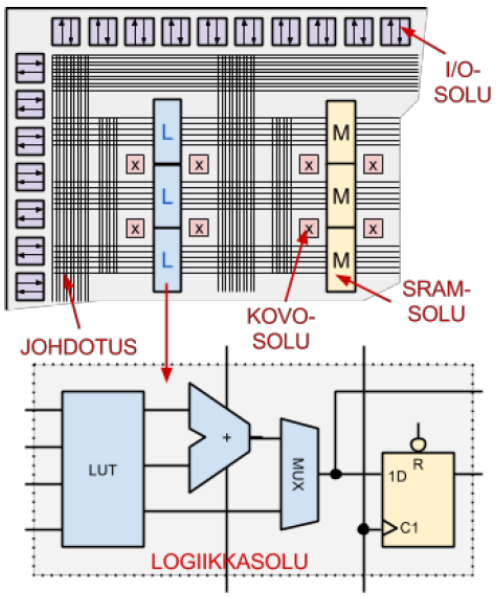
2.1. Perusteet

FPGA-piirit (Field programmable gate array) ovat digitaalisia integroituja piirejä. Ne koostuvat uudelleen ohjelmoitavista logiikkalohkoista, lohkomuisteista ja ohjelmoitavista johdotuksista lohkojen välillä kuten kuvassa 1. Näin piirillä pystytään tuottamaan teoriassa mitä tahansa digitaalisia piirejä. FPGA-piiri kehitettiin 1980-luvulla. Sen merkittävimpiä piirivalmistajia ovat Xilinx ja Intel.

FPGA-piirien logiikkalohkot koostuvat useista logiikkasoluista. Kuvassa 1 havainnollistetaan logiikkasolun koostumusta Logiikkasolut koostuvat LUT:eista (Lookup table), MUX:eista ja D-kiikuista. Nämä komponentit voidaan ohjelmoida käyttämällä lohkon sisäistä SRAM-muistia. Eli LUT voidaan ohjelmoida tekemään mikä tahansa logiikkafunktio sen tuloilla. Sitten haluttu LUT:n lähtö voidaan laittaa rekisteriin muistiin MUX:n ja D-kiikun avulla. FPGA-piirin kehittyessä, logiikkalohkoja on paranneltu lisäämällä lohkoihin omia RAM-muisteja ja lisäämällä LUT:n tulojen määrää, jolloin voidaan tuottaa monimutkaisempaa logiikkaa lohkon sisällä.

Digitaalisien piirien muodostamiseksi FPGA-piirissä logiikka lohkot kytketään toisiinsa ohjelmoitavilla johtimilla. Logiikkalohkot ovat asetettu matriisin muotoon toisiinsa nähden ja ne ovat kiinnitetty uudelleenkonfiguroitavilla kytkimillä johtimien kautta. Kytkimien ohjaussignaalit ovat tallennettu muisteihin, joita muuttamalla voidaan muodostaa kytkentöjä. Näin halutut lohkot voidaan yhdistää toisiinsa kytkimien avulla.

Ohjelmoitavan logiikan ohessa, FPGA-piirit voivat myös sisältää sulautettuja muisti lohkoja, sulautettuja prosessoreita, hyvin nopeita I/O väyliä ja IP-lohkoja. Tämä mahdollistaa FPGA-piirien käyttämisen melkein mihin tahansa, kuten kommunikaatiojärjestelmiin ja ohjelmistopohjaisiin radioihin, tutkiin, kuviin ja muihin digitaalisen signaalin prosessointi menetelmiin.[7],[3]



Kuva 1 FPGA-piirin rakenne ja muistisolun sisältö [5]

2.2. Käyttötarkoitukset

1980-luvulla huomattiin aukko digitaalisten IC-piirien tuotteissa. Toisessa päässä olivat ohjelmoitavat laitteet kuten SPLD (simple programmable logic device) ja CPLD (complex programmable logic device), jotka olivat hyvin muunneltavia, ja joilla oli nopeat suunnittelu ja muokkausajat. Kuitenkin ne eivät pysty tukemaan suuria tai monimutkaisia funktioita. Toisessa päässä asteikkoa ovat ASIC:t (application specific integrated cuircut) Ne pystyvät tukemaan monimutkaisia ja suuria funktioita, mutta ne ovat hyvin kalliita ja aikaa vieviä suunnitella. ASIC:ia ei voi myöskään muokata valmistuksen jälkeen, sillä se on integroitu piisiruun.

Tämän markkinaraon täytti Xilinx valmistama uusi IC luokka: FPGA. FPGA-piirit ovat uudelleen ohjelmoitavia kuten SPLD ja CPLD. Lisäksi FPGA-piireille pystytään toteuttamaan funktioita, jotka olivat ennen vain ASIC:lla mahdollista. Tämä mahdollistaa monimutkaisten piirien tekemisen, joita voidaan muokata mahdollisten ongelmien takia. Siksi FPGA-piireillä on hyvä tehdä digitaalisten piirien prototyyppejä ASIC-piireille tai sen osille.

FPGA-suunnittelu on myös paljon halvempaa kuin ASIC-suunnittelu. Lisäksi suunnitelmien muokkauksien toteuttaminen on paljon helpompaa FPGA-piireille. Siksi FPGA-piirit toimivat hyvin pienille yrityksille ja on alkanut löytämään valmiisiin tuotteisiin. [7]

3. MUISTITEKNOLOGIAT

3.1 Perusteet

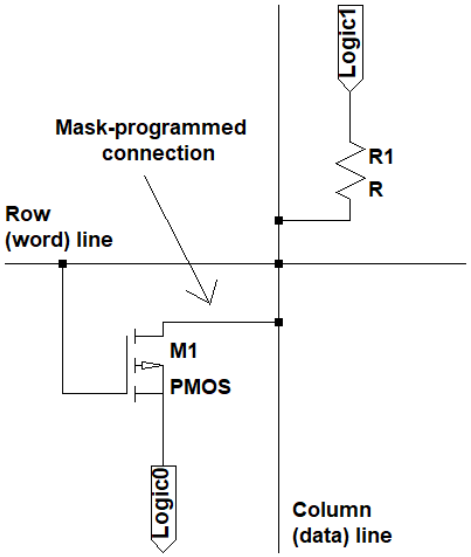
Muistia käytetään arvojen varastoimiseksi myöhempää käyttöä varten. Muisteista pitää pystyä lukemaan dataa tai kirjoittamaan ja lukemaan dataa. Muisteja on monenlaisia ja niiden toteutustavat ovat hyvin erilaisia. Seuraavassa kappaleessa tutustutaan tärkeimpiin muisteihin FPGA-piirien kannalta.

3.2 ROM

ROM-muisti (read-only memory) muistit ovat niin sanottuja ei-volatiilisia muisteja. Niiden data säilyy, vaikka systeemi suljetaan. Systeemin muut komponentit voivat lukea dataa ROM-laitteista, mutta eivät pysty kirjoittamaan uutta dataa niihin.

ROM:ia sanotaan myös maskiohjelmoiduiksi sen valmistusmenetelmän takia. ROM-muistin data koodataan sen valmistusvaiheessa valomaskin avulla, jolla luodaan transistorit ja metallikerrokset, yhdistäen ne toisiinsa piimikrosivulla.

ROM koostuu lukuisista vaakariveistä, joissa on osoite ja data pystyriveistä muodostaen matriisin. Jokaisella pystyrivillä on ylösvetovastus, joka pyrkii pitämään pystyrivit loogisessa 1. Jokaisessa pysty- ja vaakarivin risteyksessä on transistori ja mahdollisesti maski ohjelmoitu liitos.



Kuva 2 ROM-solu [7]

Kun vaakarivi laitetaan aktiiviseen asentoon, kaikki riviin liitetyt transistorit muuttuvat aktiiviseen tilaan. Näin kaikki transistorit, joissa on maski ohjelmoitu liitos yhdistävät pystyrivin transistorin kautta loogiseen 0 ja sen arvo muuttuu nollaksi. Jos transistorilla ei ole maski ohjelmoitua liitosta, transistori ei ole kytketty pystyriviin ja ylösvetovastus pitää sen arvon 1. Näin ROM pitää datansa samana valmistuksesta

alkaen. Näin ROM on hyvä muisti tietokoneohjelmien ja eri vakiodata arvojen tallentamiseen.

ROM on myös ohjelmoitavia versioita kuten PROM (programmable read-only memory), EPROM (erasable programmable memory) ja EEPROM (electrically erasable memory. Näissä versioissa pystytään muokkaamaan muistia valmistuksen jälkeen, joko muuttamalla maski ohjelmoitu liitos sulakeliitokseksi, joka voidaan rikkoa suurella jännite pulssilla tai muuttamalla vaakarivin transistori erilaiseksi, jota voidaan ohjata piirin tuloilla. Tämän avulla voitiin ROM-muisteja käyttää esim. LUT:na tai tilakoneena. [7]

3.3 RAM

RAM (random-access memory) on keskeisin muisti FPGA-piireissä. Random-access tulee siitä, että muistinpaikkojen dataan pääsee kiinni, missä järjestyksessä tahansa yhtä nopeasti. RAM on niin sanottu volatiili muisti. Kun laite suljetaan, RAM-muisti tyhjentyy kokonaan ja sen pitämä data menetetään. Se erottuu ROM-muistista myös siten, että RAM-muistiin voidaan kirjoittaa sekä lukea. Joten laitteen ollessa käynnissä RAM-soluun voidaan tallettaa minkä tahansa halutun arvon. RAM-muistin kaksi tärkeintä tyyppiä ovat DRAM (dynamic random-access memory) ja SRAM (static random-access memory).

DRAM-solu koostuu transistori -kondensaattori parista. Sana dynaaminen johtuu transistorin varauksen häviämisestä vähitellen, joten jokainen solu pitää säännöllisesti ladata, jotta data pysyy siinä. Tätä kutsutaan virkistykseksi. DRAM-solu on vie hyvin vähän tilaa, koska siihen tarvitsee vain transistori -kondensaattori parin. Lisäksi DRAM on hyvin kustannustehokas ratkaisu, kun bittejä on paljon ja kaikki solut voidaan virkistää yhtä aikaa. Kuitenkin DRAM-muistia ei käytetä paljoa ohjelmoitavassa logiikassa.

SRAM-muisti ei tarvitse virkistystä. Kun arvo ajetaan SRAM-soluun, se pysyy muuttumattomana, kunnes sitä tarkoituksella muutetaan tai virta suljetaan. Tästä sana staattinen tulee. SRAM-solu koostuu neljästä tai kuudesta transistorista, jotka muodostetaan salvaksi.

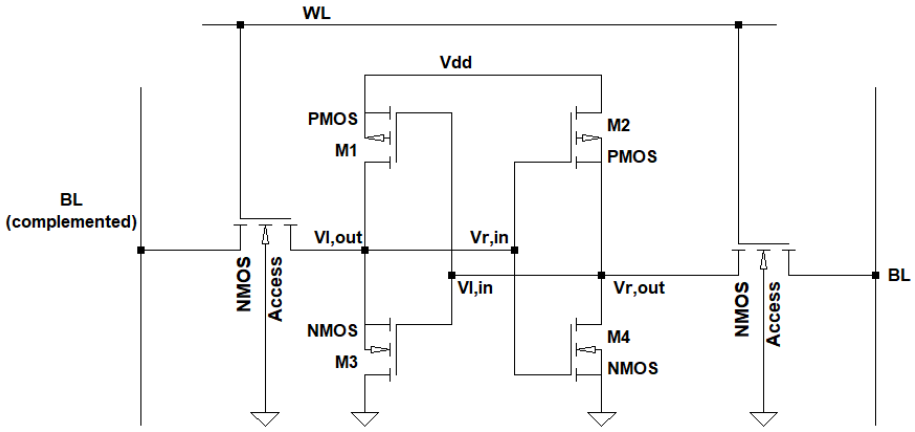
SRAM-solu koostuu kahdesta CMOS-inverteristä ja kahdesta access-transistorista. SRAM-solu pystyy pitämään datansa invertterien avulla, kunhan virtaa ei suljeta. Oletetaan, että kuvan 3 vasemman invertterin tulo on 1. Tällöin vasen invertteri kääntää sen lähdön 0. Koska vasemman puoleisen invertterin lähtö on kytketty oikean puoleisen invertteri tuloon, oikean puolimmaisen invertteri lähtö on 1. Eli data pysyy 1 vasemman puoleisessa invertterissä. Näin data säilyy SRAM-solussa ilman virkistystä. Invertterit ovat kytketty dataväylään (bit_line) ja osoiteväylään (word_line) access-transistorien kautta, jotka mahdollistavat kirjoittamisen SRAM-soluun ja lukemisen SRAM-solusta.

Kun dataa halutaan lukea tai kirjoittaa SRAM-solusta, osoiteväylä WL, pitää nostaa 1. Tämä käynnistää access-transistorit, jolloin kirjoittaminen SRAM-soluun tai lukeminen SRAM-solusta on mahdollista. Lukemisessa SRAM-solun arvo saadaan luettua dataväylän ja invertterien lähdön jännite-eron avulla. Oletetaan että vasemman

puoleisen invertterin lähtö on 1 ja oikean 0. Data-väylät ovat laitettu käyttöjännitteeseen Vdd. Nyt vasemmanpuoleisella access-transistorilla ei ole jännite-eroa ja se ei johda. Oikeanpuoleisella transistorilla on jännite-ero, jolloin oikea transistori alkaa johtamaan ja dataväylän jännite tippuu. Tämä huomataan dataväylään kiinnitetyssä sense-vahvistimessa, joka toimii piirissä komparaattorina. Kun dataväylän arvo on alhainen, komparaattorin lähtö on 1, josta voidaan päätellä oikean puoleisen invertterin lähtö on 0 ja vasemman 1. Jos lähtöjen arvot ovat toisinpäin komplementoidun dataväylän ja vasemman puoleisen access-transistorin ja invertterin lähdön välillä on jännite-ero ja se alkaa johtamaan. Kun komplementoidun dataväylän jännite laskee, komparaattorin lähdöksi saadaan 0. Joten saadaan luettua vasemman puoleisen invertterin lähdöksi 0 ja oikean 1.

Kirjoituksessa SRAM-solulle pyritään antamaan haluttu bitti arvo 0 tai 1. Kirjoituksessa toiseen dataväylään ajetaan käyttöjännite ja toinen ajetaan maihin. Jos vasemman puoleisen invertterin lähtö on 0 ja oikean 1 ja halutaan arvo 1 solulle ajetaan oikeanpuoleinen dataväylä maihin. Tämä luo jännite-eron oikean invertterin lähdön ja dataväylän kanssa. Nyt oikean puolen lähtö ajautuu maihin ja sen arvo muuttuu 0. Sitten vasemmanpuoleinen invertteri vaihtaa lähdön arvoksi 1. Jos halutaan arvo 0, niin laitetaan vasen dataväylä maihin. Tällöin vasemman puolen lähtö ajautuu maihin ja arvot vaihtuvat.

Yksi SRAM-solu tyyppisten laitteitten huonona puolena on, että jokainen solu vie hyvin paljon tilaa. Yhden bitin tallettamiseen tarvitaan neljä tai 6 transistoria, joka on paljon verrattuna DRAM-muistin transistori -kondensaattori pariin. Toinen ongelma on datan häviäminen laitteen sulkemisen jälkeen. Siksi systeemin käynnistyessä se pitää ohjelmoida aina uudestaan. Kuitenkin tällaiset laitteet ovat hyvin nopeasti ohjelmoitavissa ja ne voidaan aina ohjelmoida uudelleen. [7], [6]



Kuva 3 6 transistorinen SRAM-solu

4. XILINXIN MUISTITEKNOLOGIAT

4.1 Distributed RAM

Xilinxin logiikkalohkot koostuvat kahdesta slice:sta. Näitä slice:t eivät ole kytkettynä toisiinsa logiikkalohkon sisällä, mutta ne ovat kytketty toisiin logiikkalohkojen slice:ihin. Ne muodostavat kytkettynä monta erillistä vaakariviä, ja ovat kytkettynä toisiinsa pystyrivein. Slice on myös kahta erilaista laatua: SLICEL ja SLICEM. SLICEL voidaan käyttää vain logiikkaan. SLICEM tarkoittaa slice:a, jolla on useita eri toimintoja. Sitä voidaan käyttää multipleksereinä tai jaettuna muistina, toisin kuin SLICEL. SLICEL on noin kaksi kolmasosaa ja SLICEM on yksi kolmasosa.

Xilinxin SLICE:t koostuvat LUT:eista, MUX:eista ja D-kiikuista. Näistä LUT:it koostuvat useista SRAM-soluista. Joissakin tapauksissa näitä soluja voidaan käyttää erillisenä muistina. Tätä kutsutaan nimellä distributed RAM. Tämä nimi johtuu LUT:ien jakautumisesta ympäri mikropiirin pinnan, jolloin muisti on jakautunut ympäri mikropiirin.

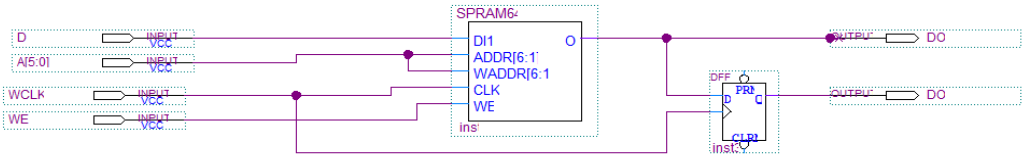
SLICEM:ien useammat LUT:it voidaan yhdistää eritavoilla muodostaen erikokoisia synkronisia muisteja. Kokoja on yksiporttisesta 32 x 1 -bittisestä muistista 4 porttiseen 64 x 1-bittiseen muistiin. Tarkastellaan jaetun muistin käyttöä kuvan 3 64 x 1 -bittisen yksiporttisen muistin avulla yksinkertaisuuden vuoksi.

Jotta jaetusta muistista voidaan lukea dataa tai kirjoittaa siihen dataa, tarvitaan muita avustavia signaaleja. Kuvassa 3 näkyy kuinka signaalit kytkeytyvät muistiin. Signaali D tuo uuden datan kirjoitettavaksi RAM-muistiin. Osoitetulo ADDR valitsee muistisolun mihin data kirjoitetaan tai mistä data luetaan signaalin A mukaan. Kirjoitus- ja lukuoperaatiota kuvaa taulukko 1. WE eli kirjoituksen sallimissignaali määrää milloin muistiin kirjoitetaan. Kun WE on 1, niin signaali D:n data ajetaan muistin sisälle. Kellosignaali CLK käytetään synkroniseen kirjoittamiseen. Täten kaikki kirjoitukset RAM-muistin soluun tapahtuvat kellosignaalin positiivisella reunalla. Kellopinnille on käytössä myös asetus, joka tekee kirjoittamisen kellosignaalin negatiivisella reunalla. Kun WE on 0, tai 1 muttei kellon nousevalla reunalla niin silloin tapahtuu lukuoperaatio. Muistin lähtöön tulee osoitetulon osoittama arvo. Tämä operaatio on asynkroninen ja ei tarvitse kellosignaalia. Useampi porttisessa RAM-moduulissa useampi 64- tai 32-bittinen RAM-moduuli on kytketty yhteen. Lukeminen ja kirjoittaminen onnistuu yhdestä portista. Muista porteista voidaan vain lukea dataa.

Taulukko 1 Jaetun muistin luku- ja kirjoitusoperaatiot

Inputs			Outputs
WE (mode)	CLK	D	O
0 (read)	X	X	Data
1 (read)	0	X	Data
1 (read)	1	X	Data
1 (write)	↑	D	D
1 (read)	↓	X	Data
Data = word addressed by bits A5:A0			

Jaettua muistia käytetään yleensä pienien tietomäärien tallentamiseen. Sitä käytetään yleensä 64-bittisien ja sitä pienempien muistien tallentamiseen, ellei SLICEM tai logiikka resursseista ole pulaa. Jaettu muisti on hyötysuhteeltaan parempi resurssien, suorituskyvyn ja tehon kannalta verrattuna sulautettuun muistiin. Jaetulla muistiolla on myös yleensä lyhyempi kellosta lähtöön aika ja vähemmän rajoituksia paikoille. Jos tarvitaan asynkronista lukemista, tarvitaan myös jaettua muistia, koska se ei ole mahdollista sulautetulla muistilla. [7], [4]



Kuva 3 Jaetun muistin SPRAM64 lohkokaavio

4.1.2 Distributed RAM ohjelmointi

Tarkastellaan jaetun muistin ohjelmointia alla olevan Xilinxin Vivado ympäristön Verilog RTL-esimerkkikoodia. Alkuun alustetaan tulot ja lähdöt muistille. Koska muisti on kaksiporttinen, se tarvitsee kaksi eri lähtödataväylää ja kaksi niitä ohjaavaa osoitetuloa. Muistin oma muuttuja muodostetaan reg muuttujalla, jolla saadaan aikaan 16-bittiä leveä ja 64-bittiä syvä muisti.

Kirjoitusoperaatio sijoitetaan always-komentoon, johon asetetaan kaikki arvojen asetukset kellon nousevalla reunalla. Kun kirjoituksensallinnan ohjaussignaali we on tosi, silloin osoitesignaalin a osoittamaan paikkaan talletetaan tulodataväylän arvo muistiin. Lukuoperaatiot asetetaan always-komennon ulkopuolelle, jotta siitä tulee asynkroninen. Ne muodostetaan assign-muuttujalla, jolloin osoitesignaalien osoittamat muistipaikkojen arvot asetetaan lähtödataväyliin. [8]

```
// Dual-Port RAM with Asynchronous Read (Distributed RAM)
// File: rams_dist.v
module rams_dist (clk, we, a, dpra, di, spo, dpo);

input clk;
input we;
input [5:0] a;
input [5:0] dpra;
input [15:0] di;
output [15:0] spo;
output [15:0] dpo;
reg [15:0] ram [63:0];
```

```
always @(posedge clk)
begin
  if (we)
    ram[a] <= di;
end
assign spo = ram[a];
assign dpo = ram[dpra];
endmodule
```

Verilog RTL-koodi jaetulle muistille [8]

4.2 Block RAM

Tiettyihin FPGA-piirien käyttötarkoituksiin tarvitaan paljon muistia, johon ei jaetun muistin pienet koko määrät riitä. Tämän takia FPGA-piirin pinnalle sijoitetaan suuria lohkoja sulautettua muistia, jota kutsutaan block RAM:ksi. Lohkot ovat sijoitettu piirin pinnalle eristettyinä muista tai organisoituna pystyriveittäin.

Nämä muistilohkot toimivat FPGA-piirin massamuistina. Riippuen laitteesta, nämä muistit voivat sisältää muutamasta tuhannesta yli kymmeniin tuhansiin bittejä dataa. Lisäksi näitä lohkoja voi olla laitteessa kymmenistä yli tuhanteen. Näin tallennuskapasiteettia laitteella olla jopa useita miljoonia bittejä. Lisäksi muistilohkot voidaan yhdistää toisiinsa luoden suurempia muistilohkoja.

Xilinxin 7-sarjan FPGA-piirit sisältävät 36 kilobitin lohkomuisteja. Lohkomuistit ovat järjestetty pystyrivein FPGA-piirin pinnalle. 7-sarjan FPGA-piirien muistimäärillä voi olla huomattavia eroja. Vähiten muistia on laitteessa XC7S6, jolla on vain viisi 36 kilobitin muistilohkoa. Eniten muistia löytyy laitteesta XC7VX1140T, jolla on jopa 1880 36 kilobitin muistilohkoa.

Lohkomuistit ovat joko yksi porttisia tai kaksiporttisia muisteja. Kuvassa 4 on 36 kilobitin RAMB36E1 kaksiporttinen lohkomuisti. Kaksiporttisessa muistissa, kummallekin portille on omat tulonsa ja ne toimivat itsenäisesti. Vaikka ne toimivat itsenäisesti, niillä on pääsy samaan 36 kilobitin muistiin. Data voidaan kirjoittaa joko toiseen tai kumpaankin porteista. Myös muistin dataa voidaan lukea joko toisesta portista tai kummastakin. Jokainen kirjoitus- ja lukuoperaatio on synkroninen, joten ne tarvitsevat kellon reunan toimiakseen.

Lohkomuisteihin voidaan kytkeä vapaavalintainen lähtörekisteri. Lähtörekisteri mahdollistaa muistin datalähdön arvon tallentamisen erilliseen rekisteriin, jos sitä tarvitaan myöhemmin. Lähtörekistereillä on myös toinen ominaisuus: se parantaa suunnitelman suorituskykyä poistamalla viiveen logiikkalohkon kiikuille.

Taulukko 2 Lohkomuistin tulot ja lähdöt

Portin toiminto	kuvaus
DI [A | B]	Tulodataväylä
DIP [A | B]	Tulodatan toinen väylä tarvittaessa
ADDR [A | B]	Osoiteväylä
WE [A | B]	Kirjoituksen sallimissignaali
EN [A | B]	Sallimissignaali
RSTREG [A | B]	Synkroninen lähtörekisterin resetointi
RSTRAM [A | B]	Synkroninen resetointi
CLK [A | B]	Kellosignaali
DO [A | B]	Lähtödataväylä
DOP [A | B]	Lähtödatan toinen väylä tarvittaessa
REGCE [A | B]	Lähtörekisterin sallimissignaali
CASCADEIN [A | B]	Sarjatulo 64K x 1 tilaa varten
CASCADEOUT [A | B]	Sarjalähtö 64K x 1 tilaa varten

Käydään läpi taulukon 2 lohkomuistin tulojen ja lähtöjen toiminnat. Kellosignaali CLK toimii muistin ajastimena. Kaikki muistin tulot toimivat kellosignaalin nousevalla reunalla. Muistin jokaiselle portille on omat kellosignaalinsa, minkä mukaan portit toimivat. Kellon polariteetti on myös muutettavissa, jos halutaan muistin toiminta toteuttaa kellon laskevalla reunalla.

Lohkomuistilla on kolme erillistä sallimissignaalia: EN (enable) ja WE (byte-write enable) REGCE (Lähtörekisterin sallimissignaali). EN signaali ohjaa sekä kirjoitus- että lukuoperaatiota. Kirjoittaessa muistiin pitää myös WE olla aktiivinen. Eli jos EN on aktiivinen mutta WE ei ole, lukuoperaatio tapahtuu ja osoiteväylän osoittamien muistisolujen datat menevät muistin lähtödata väylään. Kun kummatkin sallimissignaalit ovat aktiivisia, tapahtuu kirjoitusoperaatio, jolloin osoiteväylän osoittamiin muistisoluihin kirjoitetaan tulodataväylästä tuleva data. REGCE-signaali ohjaa valinnaista lähtörekisteriä, johon voidaan tarvittaessa tallentaa lähtödatan arvo. Tämä tapahtuu REGCE-signaalin ollessa aktiivinen.

Muistiin kirjoitettava data tulee DI-signaalista (tulodataväylä) ja DIP-signaalista (tulodatan toinen väylä). Muistista luettava data on DO-signaalista (lähtödataväylä) ja DOP (lähtödatan toinen väylä). Muistiin kirjoitettaessa lähtödataväylän data voi vastata joko kirjoitettavaa dataa tai dataa, mikä muistin lähdössä oli ennen kirjoitusta. Tämä riippuu muistin kirjoitustilasta. Kirjoitustiloja on kolme: WRITE_FIRST, READ_FIRST ja NO_CHANGE. WRITE_FIRST tilassa lähtödataväylä vastaa muistiin kirjoitettavaa muistia. READ_FIRST tilassa lähtödataväylään vaihdetaan osoitetulon valitsemien muistisolujen datan arvot ja sen jälkeen niiden päälle kirjoitetaan uudet arvot. NO_CHANGE tilassa muistisoluihin kirjoitetaan uudet arvot ja lähtödataväylä pysyy muuttumattomana.

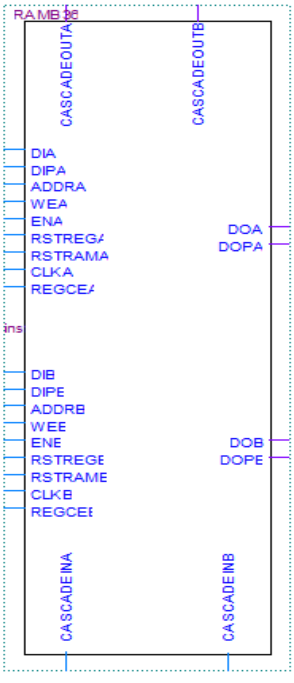
Osoiteväylä ADDR valitsee muistisolut, joihin kirjoitetaan dataa tai joista luetaan dataa. Muistilohkon kummallakin portilla on omat osoiteväylät ja tulodataväylät. Molemmilla porteilla on pääsy yhteiseen muistiin. Jos kumpaankin porttiin ajetaan yhtä aikaa sama signaali osoiteväylälle, kutsutaan törmäykseksi. Silloin kaksi eri dataa yritetään ajaa samoihin muistisoluihin samalla kellonjaksolla. Jos kummatkin portit lukevat dataa niin lukuoperaatiot onnistuvat. Jos kummatkin porteista kirjoittavat

dataa muistiin, niin muistipaikalle tulee ennalta arvaamatonta dataa. Jos taas toinen porteista lukee dataa ja toinen kirjoittaa, kirjoitus aina tapahtuu, mutta luettava data vaihtelee kirjoitustilojen mukaan.

RSTREG ja RSTRAM toimivat muistin ja lähtörekisterin resetointipinneinä. Kun RSTRAM pinni tulee todeksi, muistin datalähtö pakotetaan synkronisesti sisältämään arvon SRVAL. Tämä SRVAL arvo on itse määritettävissä oleva arvo, jolloin muistin lähtö tulee tähän arvoon resetointipinnin noustessa. RSTREG pakottaa lähtörekisterin sisältämään arvon SRVAL, kun DO_REG on tosi.

CASCADEIN ja CASCADEOUT portteja käytetään eri lohkomuistien yhdistämiseksi. Tämä mahdollistaa suurempien muistien luomisen, enemmän muistiresursseja vaativia sovelluksia varten. CASCADEIN pinnit ylemmässä lohkossa kiinnittyvät alemman muistin CASCADEOUT pinneihin. Tämä mahdollistaa lohkomuisteille 64 K x 1-bittisen tilan.

Lohkomuistilla on TDP (true dual port) tilan lisäksi toinenkin tila: SDP (simple dual port). Tällä tilalla lohkomuistin tulodataportin leveys muuttuu 18 kilobitin muisteissa 36-bittiä leveäksi ja 36 kilobitin muisteissa 72-bittiä leveäksi. SDP tilassa toinen portti toimii pelkästään lukuporttina ja toinen kirjoitusporttina. Tämän takia muistin tulot ovat erilaiset SDP tilassa kuin TDP tilassa. A-portin WE-tuloa ei ole ja sen EN-tulo muuttuu RDEN-tuloksi eli lukuoperaation sallimissignaaliksi. Lisäksi osoitesignaali muuttuu RDADDR eli lukuosoite ja kellotulo lukukelloksi. Samat muutokset tapahtuvat B-portille mutta ne muuttuvat kirjoitustuloiksi. [7],[3]



Kuva 4 Lohkomuisti RAMB36E1 lohkokaavio

4.2.2 Block RAM ohjelmointi

Lohkomuistia ohjelmoidakseen, aluksi pitää toteuttaa muistin lohkokaavio. Tämän jälkeen muistin sisäisiä asioita voidaan ohjelmoida, kuten mitä tapahtuu kirjoitusoperaation toteutuessa ja mitä tapahtuu lukuoperaation toteutuessa. Lohkokaaviota toteutettaessa alustetaan lohkokaavion portit, muistien sisältö ja muistin eri tilat. Tutkitaan toteutusta käyttämällä liitteen 1 synkronisen lohkomuistin RAMB36E1 Verilog ohjelmakoodimallia esimerkkinä.

Aluksi ohjelmakoodissa asetetaan eri tilat mahdollisten osoitetörmäyksien varalta. RDADDR_COLLISION_HWCONFIG(”Delayed_Write”) käskyllä laitetaan muistin törmäystila viiveelliseen kirjoitukseen, jolloin törmäyksiä ei tapahdu. Tähän on myös mahdollisuus laittaa tilaksi ”Performance”, jolloin kellosignaalin suorituskyky eli taajuus nousee. Tämä tila kuitenkin voi altistaa muistilohkon osoitetörmäyksille. Toisella käskyllä SIM_COLLISION_CHECK pystyään muokkaamaan simulaation käyttäytymistä, jos osoitetörmäys tapahtuu. ALL tuottaa varoituksen ja muistinlähtö tai muistipaikka menee tuntemattomaksi. WARNING_ONLY tuottaa vain varoituksen ja GENERATE_X_ONLY laittaa muistinlähdön tai muistipaikan tuntemattomaksi. NONE taas ei tuota varoitusta ja muisti tai lähtö pitää edellisen arvonsa.

Tämän jälkeen tulevat lähtörekisterin sallimissignaalin arvot DOA_REG ja DOB_REG. Kun nämä signaalit ovat tosia, lähtörekisterit tulevat käyttöön ja sallivat nopeamman kellosta lähtöön ajan yhden kellonjakson viiveen kustannuksella. Sitten tulevat EN_ECC_READ ja EN_ECC_WRITE komennot, joilla sallitaan ECC (Error code correction) virheenkorjaus mahdollisten bittivirheiden korjaamiseksi. Ne ovat oletuksena FALSE eli pois päältä, mutta ne voidaan kytkeä päälle laittamalla ne tilaan TRUE.

Seuraavaksi asetetaan alkuarvot muistipaikoille. Näitä paikkoja on INIT_00:sta INIT_7F:ään 256 bittisiä heksalukuina. Näitä muistipaikkoja on yhteensä 32 kB:n edestä. Myös INIP_00:sta INITP0F:ään saadaan 4 kB:n edestä parity dataa. Sitten asetetaan alkuarvot porttien lähtödataväyliin INIT_A ja INIT_B komennoilla.

RAM_MODE komennolla valitaan muistilohkon tilaksi joko SDP (simple dual port) tai TDP (true dual port). Kun tila on valittu, valitaan haluttu datan leveys lukemisessa ja kirjoituksessa sisältäen pariteetti bitit. Pariteettibiteillä voidaan tarkistaa, onko data oikein vai väärin. Jos porttia ei haluta käyttää voidaan tämä arvo laittaa nollaksi. RSTREG_PRIORITY muokataan resetointien prioriteetti. SIM_DEVICE(”7series”) koodilla saadaan oikeanlainen simulaatio kaikissa tilanteissa. Sitten määritellään muistin kirjoitustilat, jotka ovat WRITE_FIRST, READ_FIRST ja NO_CHANGE.

Kun nämä arvot ovat asetettu, tulee koodirivi RAMB35E1_inst. Tässä kohdassa asetetaan lohkokaavion tuloportit tulosignaaleihin ja lähtöportit lähtösignaaleihin. Tuloporttien koodeissa aluksi on lohkokaavion tuloportin nimi ja sulkuihin tuloporttiin tuleva signaali. Lähtösignaaleissa alkuun tulee lähtöportin nimi ja sulkuihin lähtösignaalien nimet.

Lohkomuisti voidaan myös toteuttaa liitteen 2 Verilog RTL-kuvaamisella. Lohkomuistia voidaan ohjelmoida käyttämään muistiansa yhden bitin leveästä ja 32 kilobittiä syvästä, 36 bittiä leveäksi ja tuhat bittiä syväksi. Sen takia tulodataväylän, lähtödataväylän ja osoitetulon leveydet pitää olla hyvin suuret, jotta muistin

varastoiminen olisi mahdollista. Lohkomuistin kirjoitus- sekä lukuoperaatio ovat synkronisia, joten kummatkin laitetaan always-komentoon, jossa kirjoitus ja lukeminen tapahtuu kellon nousevalla reunalla. Kun resetointisignaali rst muuttuu todeksi niin muistin lähtö nollaantuu. Silloin kun resetointi on alhaalla, muisti aina laittaa osoitesignaalin osoittaman arvon lähtöön.

Tämä RTL-kuvaus on kuitenkin hyvin yksinkertaistettu versio. Se kuvaa vain kirjoitus- ja lukuoperaation sekä resetoinnin toiminnan muistissa. Kuitenkin muistilla on myös monia muita tuloja, joita ei esimerkkikoodissa käytetä lainkaan. Lähtörekisteriä ja niiden resetointeja ei otettu huomioon ollenkaan. Lisäksi koodissa ei oteta huomioon cascadein ja cascadeout signaaleja, lohkomuistin eri kirjoitustiloja, muistin SDP tilaa ja virheenkorjauksia. Nämä olisivat tehneet koodista liian monimutkaisen tätä työtä varten, joten ne jätettiin pois. [2], [8]

4.3 Read-only memory

Xilinxin LUT:it pystyvät luomaan muutakin kuin normaalia jaettua muistia. Ne kykenevät luomaan myös 64 x 1-bittisiä ROM-muisteja. Lisäksi jokaisen muistisolun LUT:it kelpaavat, ei pelkästään SLICEM:n muistit. Myös 128 x 1-bittinen muisti voidaan kytkeä kahdella LUT:lla ja 256 x 1-bittinen muisti voidaan kytkeä neljällä LUT:lla. Lohkomuisti voidaan myös konfiguroida RAM-muistin sijasta ROM-muistiksi.[4], [3]

4.4 Xilinx laitteiden muistiresurssit

Xilinx on monia 7-sarjan FPGA-piirejä: spartan, artix, kintex ja virtex. Niillä on lukuisia ominaisuuksia, jotka vaikuttavat piirien suorituskykyyn. Näistä tehokkaimmat piirit ovat virtex-piirit ja heikoimmat spartan-piirit. Taulukossa 3 luetellaan heikoimman spartan-piirin muistiresurssit ja tehokkaimman virtex-piirin muistiresurssit vertailun vuoksi. Jaetun muistin määrä spartan-piirissä on vain 70 Kb kun virtex-piirissä sitä on 17700. Lohko muistia spartan-piirissä on 180 Kb ja virtex-piirillä 67680 Kb. Muistien määrässä piirien välillä on hyvin suuria eroja. Tämä on pitkälti siitä syystä, että spartan-piirejä käytetään yleensä harrastepohjalla ja tehokkaampia käytetään suunnittelu ja prototyyppi töissä. [1]

Taulukko 3 Spartan-7- ja Virtex-7-piirin muistiresurssit

	Spartan-7 (XC7S6)	Virtex-7 (XC7VH1140T)
Jaetun muistin kokonaismäärä (Kb)	70	17700
Lohkomuistien määrä (36 Kb jokainen)	5	1880
Lohkomuistin kokonaismäärä (Kb)	180	67680

5. YHTEENVETO

Työssä tutkittiin Xilinxin FPGA-piirien muistiresursseja ja tutkittiin, miten niitä käytetään suunnittelussa. FPGA-piireissä pääasiassa käytetään RAM-muisteja, mutta tarvittaessa muistit voidaan myös konfiguroida ROM-muisteiksi. Xilinx käyttää FPGA-piireissään kahdenlaista muistityyppiä: jaettua muistia ja lohkomuistia.

Jaettua muistia pyritään käyttämään alle 64-bittisten arvojen tallentamiseen. Se on hyötysuhteeltaan parempi resurssien, suorituskyvyn ja tehon kannalta kuin lohkomuisti. Suurempien muistien tallentamiseen käytetään lohkomuistia. Lohkomuistia voidaan konfiguroida tarkemmin asettamalla muistille omat tilat joko TDP tilaan tai SDP tilaan. Lohkomuistille voidaan ohjelmoida eri kirjoitustilat ja sille voidaan tehdä virheenkorjausta. Kuitenkin jaettua muistia voidaan käyttää myös suuremmille muisteille, jos tarvitaan asynkronista lukemista tai suorituskykyisempää muistia.

Muistien käyttämistä suunnittelussa tutkittiin RTL-esimerkkikoodien avulla sekä Xilinxin toteutuskoodin avulla. Tässä osiossa olisi ollut parannettavaa. RTL-koodit olivat hyvin yksinkertaisia, jotka kuvasivat ainoastaan kirjoitus- ja lukuoperaation ja resetoinnin tapahtumisen. Lohkokaavion RTL-koodissa ei ollut sen ominaisuuksia tai lähtörekisterejä ja niitten resetointeja ollenkaan. Nämä olisi voitu lisätä esimerkkikoodia muokkaamalla.

6. LÄHTEET

[1] All programmable 7 series product selection guide

. https://www.xilinx.com/support/documentation/selection-guides/7-series-product-selection-guide.pdf.

[2] Vivado design suite 7 series FPGA libraries guide.

https://www.xilinx.com/support/documentation/sw_manuals/xilinx2012_2/ug953-vivado-7series-libraries.pdf.

[3] 7 series FPGAs memory resources.

https://www.xilinx.com/support/documentation/user_guides/ug473_7Series_Memory_Resources.pdf.

[4] Xilinx 7 series FPGAs configurable logic block user guide

https://www.xilinx.com/support/documentation/user_guides/ug474_7Series_CLB.pdf.

[5] Jukka Lahti. Digitaalitekniikka 3

https://moodle.oulu.fi/pluginfile.php/3471/mod_label/intro/DT3_2017.pdf.

[6] S.B. Lokesh, K. Mecha Chandana, V. Niharika, A. Prathvusha, G.Rohitha. Design of read and write operation for 6t SRAM cell. 2018;8(1):43-46.

http://www.iosrjournals.org/iosr-jvlsi/papers/vol8-issue1/Version-1/E0801014346.pdf.

[7] Maxfield C. The design warrior's guide to FPGAs: Devices tools and flows. Boston: Newnes; 2004: 542 sivua.

https://oula.finna.fi/Record/oula.1326969.

[8] Vivado design suite user guide

https://www.xilinx.com/support/documentation/sw_manuals/xilinx2016_4/ug901-vivado-synthesis.pdf

7. LIITTEET

```
// RAMB36E1: 36K-bit Configurable Synchronous Block RAM
 // 7 Series
// Xilinx HDL Libraries Guide, version 2012.2
RAMB36E1 #(
// Address Collision Mode: "PERFORMANCE" or "DELAYED_WRITE"
.RDADDR_COLLISION_HWCONFIG("DELAYED_WRITE"),
 // Collision check: Values ("ALL", "WARNING_ONLY", "GENERATE_X_ONLY" or "NONE")
.SIM_COLLISION_CHECK("ALL"),
// DOA_REG, DOB_REG: Optional output register (0 or 1)
.DOA_REG(0),
.DOB_REG(0),
.EN_ECC_READ("FALSE"), // Enable ECC decoder,
// FALSE, TRUE
.EN_ECC_WRITE("FALSE"), // Enable ECC encoder,
// FALSE, TRUE
// INITP_00 to INITP_0F: Initial contents of the parity memory array
.INITP_00(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INITP_01(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INITP_02(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INITP_03(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INITP_04(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INITP_05(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INITP_06(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INITP_07(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INITP_08(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INITP_09(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INITP_0A(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INITP_0B(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INITP_0C(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INITP_0D(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INITP_0E(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INITP_0F(256'h0000000000000000000000000000000000000000000000000000000000000000),
// INIT_00 to INIT_7F: Initial contents of the data memory array
.INIT_00(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INIT_01(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INIT_02(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INIT_03(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INIT_04(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INIT_05(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INIT_06(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INIT_07(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INIT_08(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INIT_09(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INIT_0A(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INIT_0B(256'h0000000000000000000000000000000000000000000000000000000000000000),
```

```
.INIT_0C(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INIT_0D(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INIT_0E(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INIT_0F(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INIT_10(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INIT_11(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INIT_12(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INIT_13(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INIT_14(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INIT_15(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INIT_16(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INIT_17(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INIT_18(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INIT_19(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INIT_1A(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INIT_1B(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INIT_1C(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INIT_1D(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INIT_1E(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INIT_1F(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INIT_20(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INIT_21(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INIT_22(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INIT_23(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INIT_24(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INIT_25(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INIT_26(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INIT_27(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INIT_28(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INIT_29(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INIT_2A(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INIT_2B(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INIT_2C(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INIT_2D(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INIT_2E(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INIT_2F(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INIT_30(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INIT_31(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INIT_32(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INIT_33(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INIT_34(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INIT_35(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INIT_36(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INIT_37(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INIT_38(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INIT_39(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INIT_3A(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INIT_3B(256'h0000000000000000000000000000000000000000000000000000000000000000),
```

```
.INIT_3C(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INIT_3D(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INIT_3E(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INIT_3F(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INIT_40(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INIT_41(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INIT_42(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INIT_43(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INIT_44(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INIT_45(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INIT_46(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INIT_47(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INIT_48(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INIT_49(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INIT_4A(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INIT_4B(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INIT_4C(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INIT_4D(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INIT_4E(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INIT_4F(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INIT_50(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INIT_51(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INIT_52(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INIT_53(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INIT_54(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INIT_55(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INIT_56(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INIT_57(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INIT_58(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INIT_59(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INIT_5A(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INIT_5B(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INIT_5C(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INIT_5D(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INIT_5E(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INIT_5F(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INIT_60(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INIT_61(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INIT_62(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INIT_63(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INIT_64(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INIT_65(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INIT_66(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INIT_67(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INIT_68(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INIT_69(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INIT_6A(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INIT_6B(256'h0000000000000000000000000000000000000000000000000000000000000000),
```

```
.INIT_6C(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INIT_6D(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INIT_6E(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INIT_6F(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INIT_70(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INIT_71(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INIT_72(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INIT_73(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INIT_74(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INIT_75(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INIT_76(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INIT_77(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INIT_78(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INIT_79(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INIT_7A(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INIT_7B(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INIT_7C(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INIT_7D(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INIT_7E(256'h0000000000000000000000000000000000000000000000000000000000000000),
.INIT_7F(256'h0000000000000000000000000000000000000000000000000000000000000000),
// INIT_A, INIT_B: Initial values on output ports
.INIT_A(36'h000000000),
.INIT_B(36'h000000000),
// Initialization File: RAM initialization file
.INIT_FILE("NONE"),
 // RAM Mode: "SDP" or "TDP"
.RAM_MODE("TDP"),
// RAM_EXTENSION_A, RAM_EXTENSION_B: Selects cascade mode ("UPPER", "LOWER", or "NONE")
.RAM_EXTENSION_A("NONE"),
.RAM_EXTENSION_B("NONE"),
// READ_WIDTH_A/B, WRITE_WIDTH_A/B: Read/write width per port
.READ_WIDTH_A(0), // 0-72
.READ_WIDTH_B(0), // 0-36
.WRITE_WIDTH_A(0), // 0-36
.WRITE_WIDTH_B(0), // 0-72
// RSTREG_PRIORITY_A, RSTREG_PRIORITY_B: Reset or enable priority ("RSTREG" or "REGCE")
.RSTREG_PRIORITY_A("RSTREG"),
.RSTREG_PRIORITY_B("RSTREG"),
// SRVAL_A, SRVAL_B: Set/reset value for output
.SRVAL_A(36'h000000000),
.SRVAL_B(36'h000000000),
// Simulation Device: Must be set to "7SERIES" for simulation behavior
.SIM_DEVICE("7SERIES"),
// WriteMode: Value on output upon a write ("WRITE_FIRST", "READ_FIRST", or "NO_CHANGE")
.WRITE_MODE_A("WRITE_FIRST"),
.WRITE_MODE_B("WRITE_FIRST") )
RAMB36E1_inst (
// Cascade Signals: 1-bit (each) output: BRAM cascade ports (to create 64kx1)
```

```
.CASCADEOUTA(CASCADEOUTA),
// 1-bit output: A port cascade
.CASCADEOUTB(CASCADEOUTB),
// 1-bit output: B port cascade
// ECC Signals: 1-bit (each) output: Error Correction Circuitry ports
.DBITERR(DBITERR), // 1-bit output: Double bit error status
.ECCPARITY(ECCPARITY),          // 8-bit output: Generated error correction parity
.RDADDRECC(RDADDRECC),          // 9-bit output: ECC read address
.SBITERR(SBITERR),              // 1-bit output: Single bit error status
// Port A Data: 32-bit (each) output: Port A data
.DOADO(DOADO), // 32-bit output: A port data/LSB data
.DOPADOP(DOPADOP), // 4-bit output: A port parity/LSB parity
// Port B Data: 32-bit (each) output: Port B data
.DOBDO(DOBDO), // 32-bit output: B port data/MSB data
.DOPBDOP(DOPBDOP), // 4-bit output: B port parity/MSB parity
// Cascade Signals: 1-bit (each) input: BRAM cascade ports (to create 64kx1)
.CASCADEINA(CASCADEINA), // 1-bit input: A port cascade
.CASCADEINB(CASCADEINB), // 1-bit input: B port cascade
// ECC Signals: 1-bit (each) input: Error Correction Circuitry ports
.INJECTDBITERR(INJECTDBITERR), // 1-bit input: Inject a double bit error
.INJECTSBITERR(INJECTSBITERR), // 1-bit input: Inject a single bit error
// Port A Address/Control Signals: 16-bit (each) input: Port A address and control signals (read port
// when RAM_MODE="SDP")
.ADDRARDADDR(ADDRARDADDR), // 16-bit input: A port address/Read address
.CLKARDCLK(CLKARDCLK), // 1-bit input: A port clock/Read clock
.ENARDEN(ENARDEN), // 1-bit input: A port enable/Read enable
.REGCEAREGCE(REGCEAREGCE), // 1-bit input: A port register enable/Register enable .RSTRAMARSTRAM(RSTRAMARSTRAM), //
1-bit input: A port set/reset .RSTREGARSTREG(RSTREGARSTREG), // 1-bit input: A port register set/reset
.WEA(WEA), // 4-bit input: A port write enable
// Port A Data: 32-bit (each) input: Port A data
.DIADI(DIADI), // 32-bit input: A port data/LSB data
.DIPADIP(DIPADIP), // 4-bit input: A port parity/LSB parity
// Port B Address/Control Signals: 16-bit (each) input: Port B address and control signals (write port
// when RAM_MODE="SDP")
.ADDRBWRADDR(ADDRBWRADDR), // 16-bit input: B port address/Write address
.CLKBWRCLK(CLKBWRCLK), // 1-bit input: B port clock/Write clock
.ENBWREN(ENBWREN), // 1-bit input: B port enable/Write enable
.REGCEB(REGCEB), // 1-bit input: B port register enable
.RSTRAMB(RSTRAMB), // 1-bit input: B port set/reset
.RSTREGB(RSTREGB), // 1-bit input: B port register set/reset
.WEBWE(WEBWE), // 8-bit input: B port write enable/Write enable
// Port B Data: 32-bit (each) input: Port B data
.DIBDI(DIBDI), // 32-bit input: B port data/MSB data
.DIPBDIP(DIPBDIP) // 4-bit input: B port parity/MSB parity
);
```

Liite 1: Lohkomuistin Verilog koodi [2]

```
// End of RAMB36E1_inst instantiation

// Block RAM with Resettable Data Output
// File: rams_sp_rf_rst.v

module rams_sp_rf_rst (clk, en, we, rst, addr, di, dout);
input clk;
input en;
input we;
input rst;
input [9:0] addr;
input [15:0] di;
output [15:0] dout;

reg [15:0] ram [1023:0];
reg [15:0] dout;

always @(posedge clk)
begin
  if (en) //optional enable
    begin
      if (we) //write enable
        ram[addr] <= di;
      if (rst) //optional reset
        dout <= 0;
      else
        dout <= ram[addr];
    end
end
endmodule
```

Liite 2: Lohkomuistin RTL-kuvaus [8]