Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Martin Artnik

Slikovni filtri v vezjih FPGA

diplomsko delo

visokošolski strokovni študijski program prve stopnje računalništvo in informatika

Mentor: prof. dr. Patricio Bulić

Ljubljana 2013

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Univerza *v Ljubljani*
Fakulteta *za računalništvo in informatiko*

Tržaška 25
1001 Ljubljana, Slovenija
telefon: 01 476 84 11
01 476 83 87
faks: 01 426 46 47
01 476 87 11
www.fri.uni-lj.si
e-mail: dekanat@fri.uni-lj.si

Št. naloge: 00318/2012
Datum: 03.09.2012

Univerza v Ljubljani, Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Kandidat: **MARTIN ARTNIK**

Naslov: **SLIKOVNI FILTRI V VEZJIH FPGA**
IMAGE FILTERING IN FPGAS

Vrsta naloge: Diplomsko delo visokošolskega strokovnega študija prve stopnje

Tematika naloge:

V programirljivem vezju Xilinx Virtex FPGA implementirajte sistem, ki omogoča konvolucijsko filtriranje slik. Sistem naj omogoča uporabo poljubnih konvolucijskih mask velikosti 3x3 piksle. Preučite dve rešitvi: z obrobljenim kvadratom, ki uporablja prilagojen zapis slik v pomnilniku ter cevovodno implementacijo s pomikalnimi registri za piksle. Implementirajte tisto, ki bo optimalna glede na oceno porabe logičnih vrat in hitrosti delovanja. V vezju FPGA slike hranite v bločnem pomnilniku RAM. V ta namen v vezju FPGA implementirajte tudi krmilnik za tak pomnilnik.

Mentor:
izr. prof. dr.Patricio Bulić

Fakulteta za računalništvo in informatiko • UNIVERZA V LJUBLJANI • 2

Dekan:
prof. dr. Nikolaj Zimic

Izjava o avtorstvu diplomskega dela

Spodaj podpisani Martin Artnik, z vpisno številko **63070024**, sem avtor diplomskega dela z naslovom:

Slikovni filtri v vezjih FPGA

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom prof. dr. Patricia Bulića,
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela
- soglašam z javno objavo elektronske oblike diplomskega dela v zbirki "Dela FRI".

V Ljubljani, dne 13. marca 2013 Podpis avtorja:

Zahvaljujem se svoji družini, prijateljem, mentorju in Tini za podporo in spodbudo.

Vsem prijateljem.

Kazalo

Seznam uporabljenih kratic in simbolov

VHDL - VHSIC Hardware Description Language
VHSIC - Very-High-Speed Integrated Circuits
FPGA - Field Programmable Gate Array
RAM - Random Access Memory
DDR - Double Data Rate
PGM - Portable Graymap
UART - Universal Asynchronous Receiver/Transmitter

Povzetek

Konvolucijska jedra se pogosto uporabljajo za obdelavo in analizo slik. Z njimi je mogoče na slike aplicirati različne filtre. Ti filtri so lahko le estetskega pomena (npr. izostritev ali zameglitev slike), lahko pa so pomembno orodje pri analizi slik, pogost primer je prepoznavanje robov. Implementacij konvolucije slik s konvolucijskimi jedri je veliko, lahko so npr. programske in tečejo na osebnem računalniku, lahko pa so tudi strojne. Strojne implementacije so primerne takrat, ko potrebujemo hitrost in obdelujemo veliko podatkov. Še posebej se take implementacije uporablja v raznih vgrajenih sistemih. Cilj naše diplomske naloge je bila implementacija konvolucje slik s konvolucijskimi jedri v jeziku VHDL, za Xilinxovo platformo FPGA Virtex-6. Implementacija omogoča uporabo poljubnih konvolucijskih jeder velikosti 3×3 na slikah velikost 128×128 pikslov, pri čemer je piksel velik 8 bitov. Glavno enoto za obdelavo slike smo za demonstracijo delovanja povezali z bločnim pomnilnikom RAM ter serijskim vmesnikom UART, preko katerega prejemamo ter pošiljamo podatke. Implementirali smo tudi enostaven krmilnik za bločni RAM, kar nam omogoča enostavnejšo prilagoditev za delovanje z različnimi krmilniki za pomnilnik. Pravilno delovanje implementacije smo najprej testirali na simulatorju, nato pa tudi na dejanski razvojni plošči Xilinx Virtex-6. Implementacija je pripravljena tako, da je enostavna za razširitev z različnimi pomnilniškimi krmilniki, prav tako pa je enostavna prilagoditev za delovanje z različnimi velikostmi slik.

Ključne besede:

FPGA, VHDL, Xilinx Virtex-6, konvolucija, konvolucijsko jedro, analiza slik

Abstract

Convolution kernels are frequently used for image processing and analysis. They can be used to apply different filters on images. These filters can serve either an esthetic purpose (for example, to sharpen or blur an image), or they can be a useful tool for image analysis, a common example of which is edge detection. There are many implementations of convolution of an image with a convolution kernel. They can be software implementations, which runs on a desktop computer, or they can be implemented in hardware. Hardware implementations are useful in cases, when we need speed and are working with large sets of data. These kinds of implementations are common in various embeded systems. The goal of our thesis was to implement convolving an image with a convolution kernel in the VHDL language, for use on the Xilinx FPGA platform Virtex-6. The implementation enables us to use a 3×3 size convolution kernel of our choice on images the size of 128×128 8-bit pixels. For demonstrating the use of our main image processing unit, we have linked it to block RAM and a UART serial interface, which we use to send and recieve data. We also implemented a simple block RAM controller, which enables easier modification of our logic for use with various memory controllers. We first tested our implementation on a simulator and later also on the Xilinx Virtex-6 development board itself. The implementation is ready for modification for use with various memory controllers and also enables simple modification for use with different image sizes.

Keywords:

FPGA, VHDL, Xilinx Virtex-6, convolution, convolution kernel, image analysis

Poglavje 1

Uvod

Konvolucija slik s konvolucijskimi jedri je pomembno orodje pri analizi slik, kot je naprimer prepoznavanje robov. Na področjih, kjer se ta postopek uporablja (npr. računalniški vid), velikokrat delamo z večjimi količinami podatkov in potrebujemo hitro procesiranje. V takih primerih je primerna strojna implementacija enot za procesiranje s pomočjo vezji FPGA. Namen diplomske naloge je implementacija enote za procesiranje slik z uporabo poljubnih konvolucijskih jeder velikosti 3×3 in demonstracija njenega delovanja. Za pomoč pri demonstraciji delovanja na razvojni platformi Xilinx Virtex-6 ML605 bomo uporabili enoto UART za pošiljanje ter prejemanje podatkov in bločni pomnilnik RAM, za katerega bomo implementirali tudi krmilnik, ki bo služil lažji prilagoditvi enote za uporabo različnih pomnilniških krmilnikov. V drugem poglavju bomo na kratko opisali konvolucijo in uporabo konvolucijskih jeder. V tretjem poglavju bomo predstavili nekaj rešitev problema učinkovitega branja podatkov iz pomnilnika in obdelave, ter osnove rešitve, ki smo jo uporabili pri končni implementaciji. Četrto poglavje je namenjeno predstavitvi naše implementacije na razvojni platformi Xilinx Virtex-6 ML605, krmilnika bločnega pomnilnika RAM, modula za konvolucijo, modula za obdelavo slike ter vrhovnega modula. V petem poglavju bomo opisali testiranje enote ter predstavili rezultate.

Poglavje 2

Obdelava slik s konvolucijskimi jedri

2.1 Konvolucija

Konvolucija je matematična operacija nad dvema funkcijama, definirana kot (enačba (2.1)[1]):

$$(f * g)(t) = \int_{-\infty}^{\infty} f(u)g(t-u)du \tag{2.1}$$

Iz dveh funkcij s pomočjo konvolucije dobimo tretjo funkcijo, ki predstavlja ploščino preseka vhodnih funkcij kot funkcijo količine premika ene od funkcij po abscisni osi. Za potrebe digitalnega procesiranja uporabljamo diskretno konvolucijo, definirano v enačbi (2.2)[2].

$$(f * g)[n] = \sum_{m=-\infty}^{\infty} f[m]g[n-m] = \sum_{m=-\infty}^{\infty} f[n-m]g[m] \tag{2.2}$$

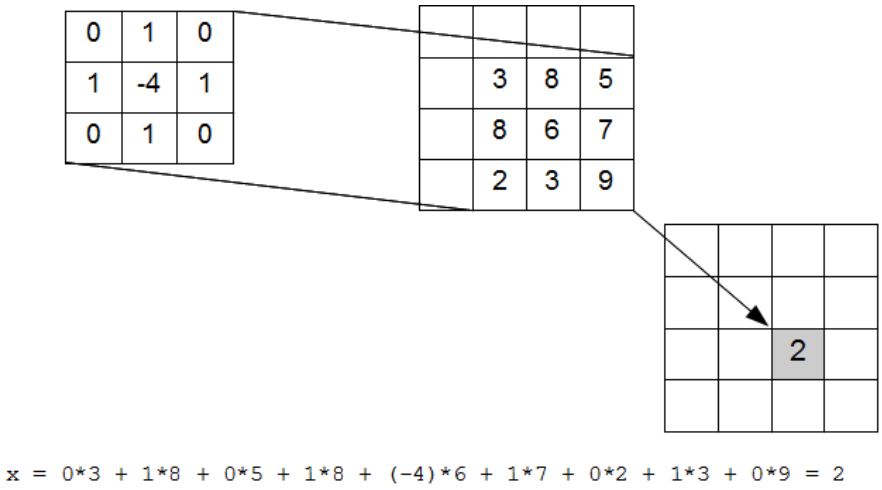
2.2 Konvolucijsko jedro

Konvolucijsko jedro je matrika, ki predstavlja diskretno funkcijo $g[i, j]$. Sliko si namreč lahko predstavljamo kot funkcijo $f[i, j] = x$, kjer je x vrednost piksla na lokaciji *(i, j)*. V našem primeru delamo s konvolucijskimi jedri

```
for i in 1 ... sirina_slike-1
  for j in 1 ... visina_slike-1
  nova_slika[i][j] =
      jedro[0][0] * slika[i-1][j-1] +
      jedro[1][0] * slika[i][j-1] +
      jedro[2][0] * slika[i+1][j-1] +
      jedro[0][1] * slika[i-1][j] +
      jedro[1][1] * slika[i][j] +
      jedro[2][1] * slika[i+1][j] +
      jedro[0][2] * slika[i-1][j+1] +
      jedro[1][2] * slika[i][j+1] +
      jedro[2][2] * slika[i+1][j+1]
  end for
end for
```

Slika 2.1: Psevdokoda algoritma za konvolucijo slike z jedrom velikosti 3 × 3

velikosti 3 × 3. Ko izvajamo konvolucijo nad sliko, pomikamo konvolucijsko jedro po celotni sliki in izračunavamo vrednosti pikslov tako, da izvajamo konvolucijo jedra ter pikslov, ki se trenutno nahajajo pod jedrom (slika 2.2). Algoritem za konvolucijo z jedrom velikosti 3 × 3 lahko opišemo s psevdokodo na sliki 2.1.



Slika 2.2: Primer konvolucije, ki jo izvedemo s konvolucijskim jedrom nad sliko.

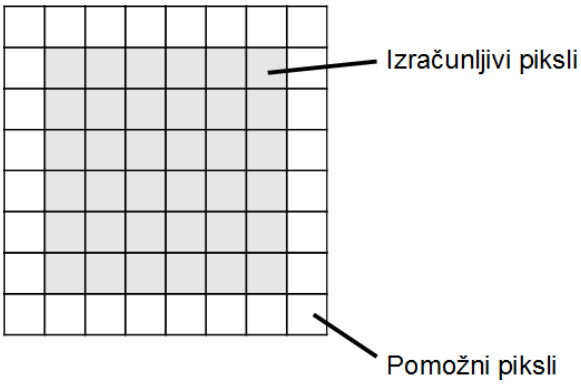
Poglavje 3

Idejna zasnova

Pri uporabi pomnilnika RAM pri obdelavi slik z uporabo konvolucijskih matrik, se pojavi problem, kako učinkovito hraniti, oziroma brati podatke iz pomnilnika. Težavno je namreč, da potrebujemo za obdelavo enega piksla slike tudi njemu sosednjih osem pikslov. To v praksi pomeni, da bomo morali podatke iz pomnilnika brati večkrat, zagotoviti dovolj velik predpomnilnik, v katerem bomo hranili tri vrstice naenkrat, ali pa drugače organizirati podatke v pomnilniku. Vsak od pristopov ima seveda svoje prednosti ter slabosti. Delno smo bili omejeni tudi z velikostjo prebranega bloka iz pomnilnika (256 ali 512 bitov), saj bili na začetku omejeni s specifikacijami že izdelanega krmilnika RAM[5], ki smo ga kasneje nadomestili z lastnim krmilnikom.

3.1 Rešitev z obrobljenimi kvadrati

Prva rešitev, o kateri smo razmišljali, je imela osnovo v reorganizaciji zapisa slike v pomnilnik. Če privzamemo, da iz pomnilnika lahko preberemo 512 bitov v enem branju, kar je bila specifikacija krmilnika RAM, s katerim smo sprva delali, bi bila slika v pomnilniku lahko shranjena kot zaporedje kvadratov velikosti 8×8 pikslov, pri čemer bi zunanji piksli kvadrata predstavljali dodatne podatke (slika 3.1) in bi se zato v pomnilniku ponavljali. To je tudi glavna slabost takega pristopa, saj to v našem primeru pomeni 77 %



Slika 3.1: Kvadrat velikosti 8 × 8 s pomožnimi robnimi piksli.

povečanje porabe prostora, kar vsekakor ni malo. Za vsakih 36 izračunljivih pikslov namreč potrebujemo 64 dejanskih pikslov v pomnilniku - 36 + 28 robnih pikslov. Branje iz pomnilnika se sicer povsem trivializira, potrebno je namreč le branje zaporednih blokov podatkov, vendar je tak pristop precej nefleksibilen in potraten. To možnost smo predstavili zgolj kot teoretično uporabno, morda v kakšnem zelo specifičnem primeru. Za naše potrebe pa se je že po krajšem premisleku izkazala za neprimerno.

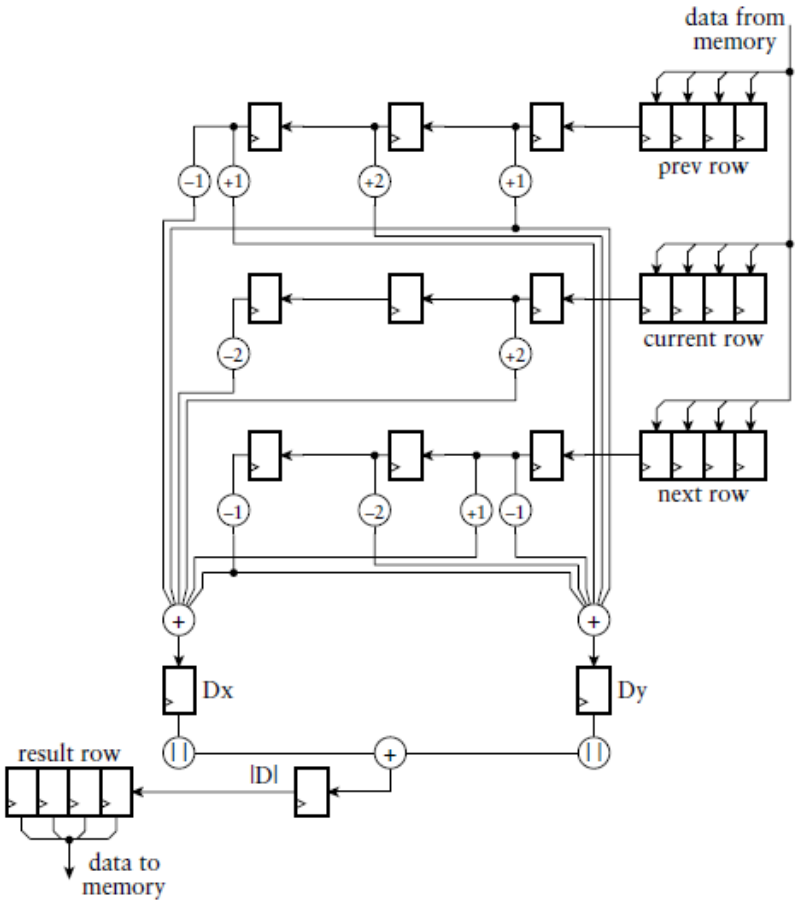
3.2 Rešitev s pomikalnimi registri za piksle

Primerno rešitev smo našli v knjigi Petra J. Ashendena, *Digital Design* [3], kjer je opisan primer zgradbe pospeševalnika za prepoznavanje robov z uporabo Sobelovega operatorja. V pomikalne registre beremo bloke podatkov iz prejšnje, trenutne ter naslednje vrstice, nato pa piksle enega za drugim pomikamo v cevovod, kjer so nato obdelani s Sobelovim operatorjem (slika 3.2). Ko izpraznemo registre, naložimo naslednje tri bloke podatkov. Osnovno idejo te rešitve smo morali razširiti za naš primer, kjer ne operiramo le s Sobelovim operatorjem, temveč s poljubnim konvolucijskim jedrom velikosti 3 × 3. Velika prednost takega pristopa je, da lahko delamo z različnimi velikostmi slik. Za primer, ko iz pomnilnika dobivamo bloke podatkov velikosti

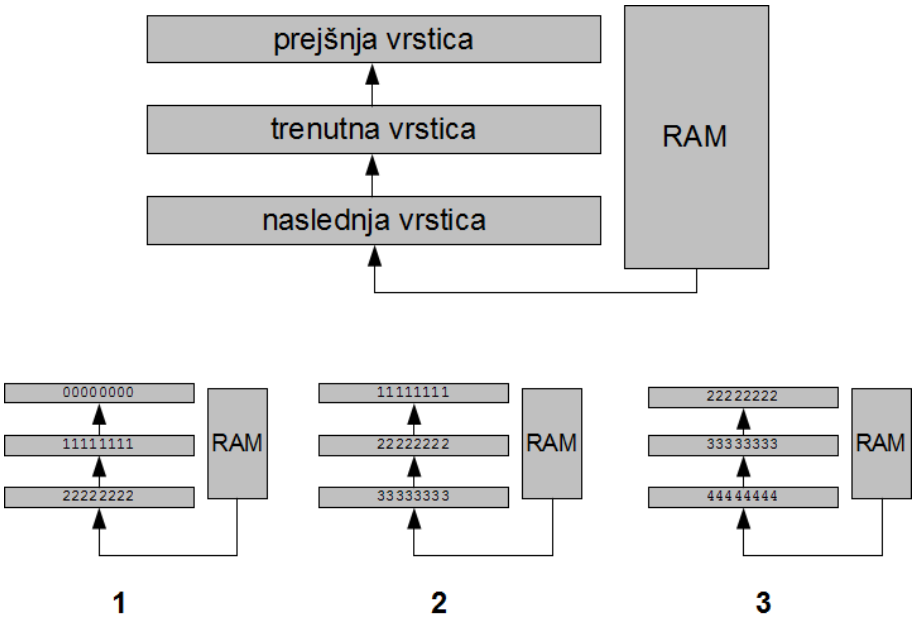
256 bitov, morajo biti dimenzije slike le deljive z 32 (32×8 bit $= 256$).

3.3 Rešitev s hranjenjem treh vrstic slike

Variacija pristopa s pomikalnimi registri je, da v vsakem od treh registrov hranimo celotno vrstico. Ko končamo procesiranje vseh pikslov v vrstici, samo pomaknemo vsebino trenutne vrstice v register za prejšnjo vrstico, vsebino naslednje vrstice v trenutno vrstico, v register za naslednjo vrstico pa naložimo nove podatke iz pomnilnika. Velika prednost takega pristopa je, da vse podatke iz pomnilnika beremo le enkrat, kar pomeni manj dostopov do pomnilnika ter hitrejše delovanje. Na žalost pa je taka implementacija zelo prostorsko potratna za vezje FPGA in zelo slabo skalabilna. Prav tako se je ne da enostavno razširiti za delovanje z različnimi velikostmi slik, zato je taka rešitev nefleksibilna.



Slika 3.2: Shema pospeševalnika za prepoznavanje robov. Vir [3].



Slika 3.3: Primer nalaganja podatkov v primeru hranjenja celotne vrstice v vsakem od treh registrov.

Poglavje 4

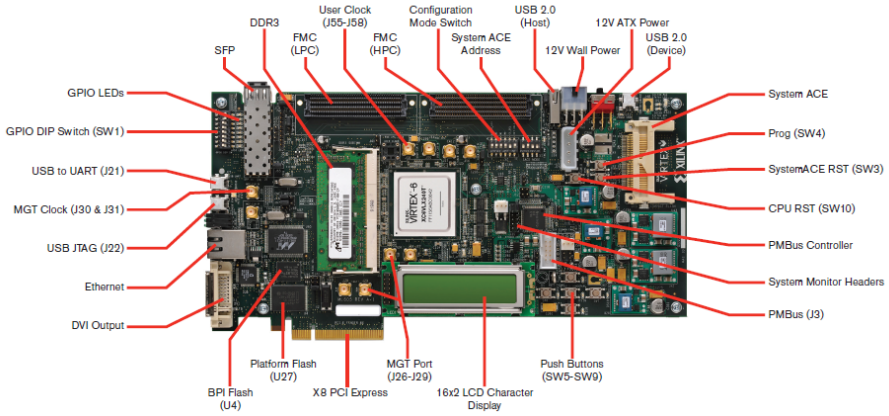
Implementacija na platformi Xilinx Virtex-6

Za implementacijo in testiranje enote smo uporabili Xilinxovo razvojno platformo Xilinx Virtex-6 ter jezik VHDL.

Realizirali smo štiri module. Prvi modul je krmilnik za bločni RAM. Uporabili smo 32 KB bločnega pomnilnika s širino besede 8 bitov. Drugi modul je modul za konvolucijo dveh 3×3 matrik. Modul za obdelavo slike sprejema, obdeluje ter oddaja podatke z uporabo modula za konvolucijo. Vse module povezuje vrhovni modul, ki posreduje podatke med krmilnikom RAM ter modulom za obdelavo slike.

4.1 Platforma Xilinx Virtex-6 ML605

Virtex-6 ML605 je razvojna plošča, ki vsebuje Xilinxov Virtex-6 XC6VLX240T-1FFG1156 čip FPGA [4]. Poleg čipa, so na plošči tudi razne naprave, s katerimi se pogosto srečujemo pri razvijanju vgrajenih sistemov. Na voljo so nam naprimer DDR3 RAM, gigabitni Ethernet, PCI Express ® vmesnik, UART in drugo [4]. Ploščo in njene naprave si lahko ogledamo na sliki 4.1.



Slika 4.1: Razvojna plošča Xilinx Virtex-6 ML605. Vir [4]

4.2 Krmilnik za bločni RAM

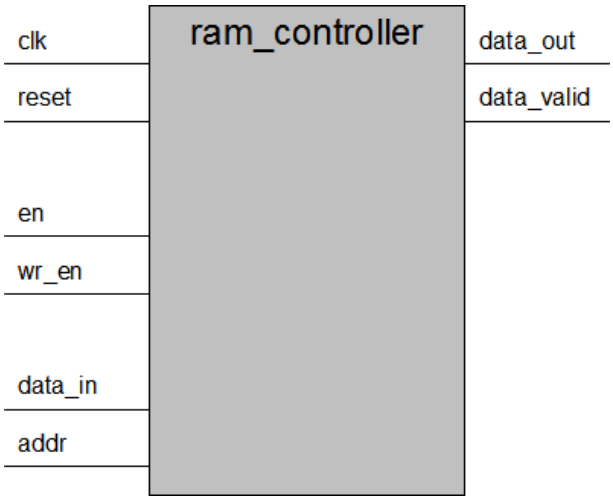
Za nadzorovanje branja in pisanja v bločni RAM smo implementirali preprost krmilnik prikazan na sliki 4.2, ki pri vsakem branju iz pomnilnika prebre 32 8-bitnih besed, prav toliko pa jih zapiše pri pisanju. S tem smo zagotovili lažjo morebitno prilagoditev enote za uporabo različnih krmilnikov za različne pomnilnike (npr. DDR), ki delujejo na podoben način.

Vhodni signali krmilnika so naslednji:

- **clk** - ura
- **reset** - sinhrona ponastavitev
- **en** - omogočimo interakcijo z RAM
- **wr_en** - omogočimo pisanje v RAM
- **data_in** - vhodni podatki (256-bit)
- **addr** - pomnilniški naslov za branje ali pisanje

Izhoda sta dva:

- **data_out** - izhodni podatki (256-bit)



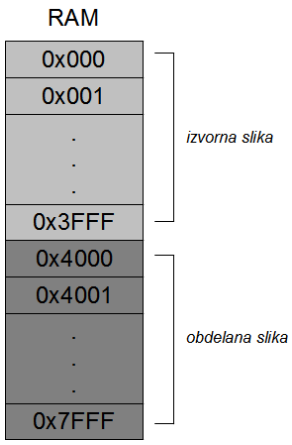
Slika 4.2: Shema vhodnih ter izhodnih signalov krmilnika za RAM.

- **data_valid** - podatki so pripravljeni za branje, ali pa zapisani v pomnilnik

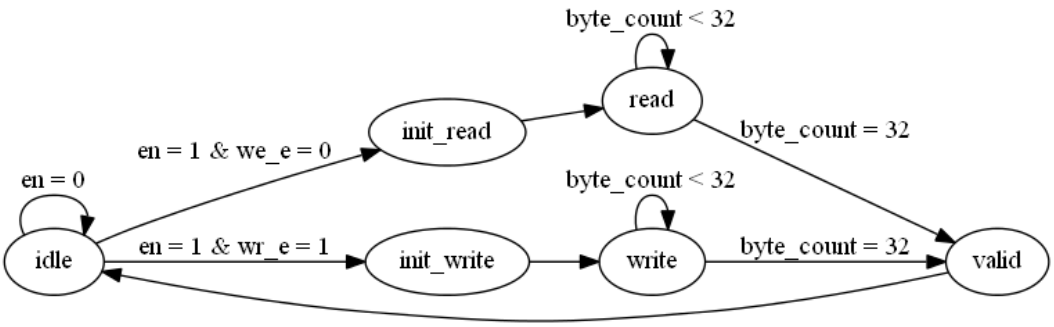
Za hranjenje vhodne slike in izhodne slike smo uporabili 32KB bločnega pomnilnika. Vhodno sliko hranimo v prvih 16KB, rezultat pa zapisujemo v zadnjih 16KB, organizacija pomnilnika je prikazana na sliki 4.3.

Širina pomnilniške besede je 8 bitov, kar ustreza velikosti piksla na naši sliki. Krmilnik nadzoruje branje iz RAM-a ter pisanje podatkov v RAM, realiziran pa je kot končni avtomat, prikazan na sliki 4.4, vrednosti signalov v različnih stanjih pa so prikazane v tabeli 4.1.

Začetno stanje je stanje *idle*, kjer čakamo na ukaz. V odvisnosti od vrednosti vhodnega signala **we_e**, preidemo v stanje *init_read* ali *init_write*, kjer naložimo vhodni naslov v register za hranjenje naslova, zatem pa preidemo v stanje *read* ali *write*. V stanju *read* podatke prenašamo iz RAM-a, po eno 8-bitno besedo na urino periodo, v 256-bitni pomikalni register (slika 4.5). Najprej podatke pomaknemo za 8 bitov v levo, nato pa v spodnjih 8 bitov registra naložimo novo besedo. V stanju *write* gre za obraten postopek, kjer zgornjih 8 bitov zapišemo v RAM, nato pa pomaknemo vsebino registra za 8 bitov v desno. Ko zapišemo ali preberemo 32 zaporednih besed,



Slika 4.3: Organizacija pomnilnika.



Slika 4.4: Diagram prehajanja stanj za končni avtomat krmilnika za RAM.

signal / stanje	idle	init_read	read	init_write	write	valid
data_fifo_push	0	0	1	0	1	0
data_fifo_fill	0	0	0	1	0	0
ram_en	0	0	1	0	1	0
ram_wr_en	0	0	0	0	1	0
data_valid_sig	0	0	0	0	0	1
byte_count_up	0	0	1	0	1	0
byte_count_reset	0	0	0	0	0	1
ram_addr_load	0	1	0	1	0	0
ram_addr_count_up	0	0	1	0	1	0

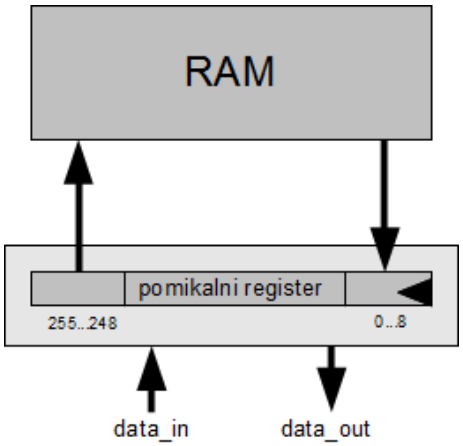
Tabela 4.1: Tabela vrednosti signalov v različnih stanjih avtomata krmilnika za RAM.

se premaknemo v stanje *valid*, kjer postavimo izhodni signal *data_valid* in tako sporočimo, da so podatki bodisi zapisani bodisi prebrani in na voljo na izhodu **data_out**.

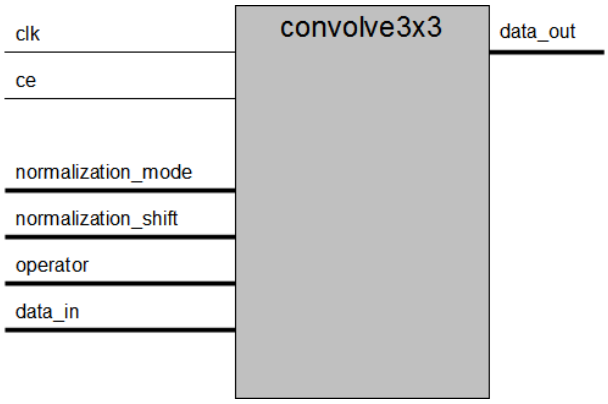
4.3 Modul za konvolucijo

Modul za konvolucijo (slika 4.6) množi istoležne piksle dveh 3×3 matrik, v našem primeru konvolucijskega jedra ter izseka slike, obenem pa poskrbi tudi za normalizacijo. Modul ima sledeče vhode:

- **clk** - ura
- **ce** - omogoči uro
- **normalization_mode** - način normalizacije, 2 bita
- **normalization_shift** - normalizacijski pomik, integer
- **operator** - konvolucijski operator (72-bit, 9×8 bitov)
- **data_in** - vhodni podatki (72-bit, 9×8 bitov)



Slika 4.5: Pomikalni register za hranjenje podatkov, ki jih zapisujemo v RAM ali jih beremo iz njega.



Slika 4.6: Modul za konvolucijo.

Izhod je en, **data_out**, širok je 8 bitov, kar ustreza enemu pikslu.

Množenje se izvaja s pomočjo devetih v Xilinx Virtex-6 vgrajenih množilnikov. Vsak zmnoži dva istoležna piksla, rezultati pa se potem predznačeno seštejejo, nato sledi normalizacija. Ker ima piksel na naši sliki lahko le vrednosti od 0 do 255, je treba definirat, kaj se zgodi, ko je rezultat konvolucije manjši od 0 ali večji od 255. Take primere lahko v modulu obravnavamo na tri načine, ki jih izberemo z vhodom **normalization_mode**:

- **normalization_mode = 00** - „cutoff“

  ```
  x = 0 when x < 0
  x = 255 when x > 255
  ```

- **normalization_mode = 01** - „cutoff_absolute“

  ```
  x = |x|
  x = 255 when x > 255
  ```

- **normalization_mode = 10** - „absolute“

  ```
  x = |x|
  ```

Pri slednjem načinu („absolute“) moramo uporabiti tudi vhod **normalization_shift**, s katerim povemo, za koliko mest v desno bomo pomaknili bite rezultata. To pomeni, da bomo vsak rezultat delili z $2^{normalization\_shift}$, s čimer ga bomo spravili pod vrednost 255. Stopnja pomika mora biti izbrana za vsako konvolucijsko jedro posebej, tako da izračunamo, s katero potenco števila 2 moramo deliti najvišjo možno vrednost, da bo padla pod 255.

4.4 Modul za obdelavo slike

Modul sprejema odseke slike iz treh vrstic (prejšnje, trenutne, naslednje), ter izvaja konvolucijo z določenim operatorjem z uporabo modula za konvolucijo. Vsebuje naslednje vhode ter izhode (slika 4.7):

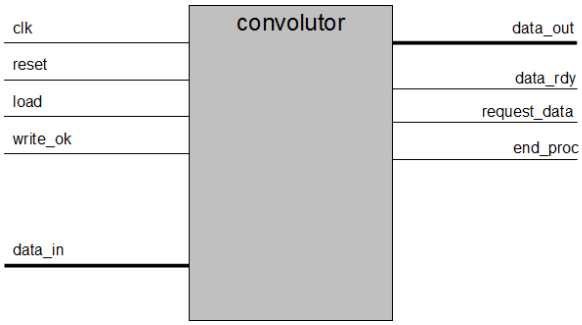
Vhodi:

- **clk** - ura
- **reset** - sinhrona ponastavitev
- **load** - omogoči zapis podatkov na vhodi **data_in** v enega izmed treh registrov za vrstice
- **write_ok** - izhodni podatki so se uspešno zapisali
- **data_in** - vhodni podatki, 256-bit

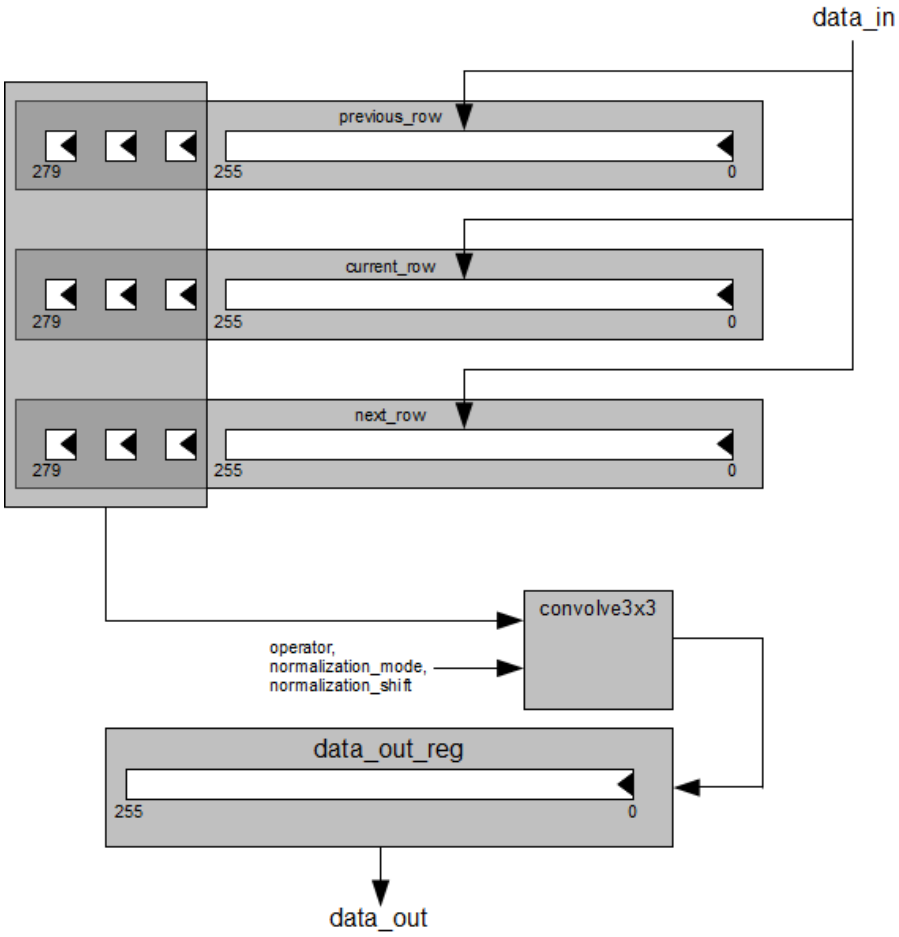
Izhodi:

- **data_out** - izhodni podatki, 256-bit
- **data_rdy** - sporoča veljavnost izhodnih podatkov
- **request_data** - signal za zahtevo novih podatkov
- **end_proc** - signal, ki sporoča, da je procesiranje končano

Deli slike se shranjujejo v tri pomikalne registre velikosti 256 + 24 bitov (slika 4.8). Matrika za konvolucijo je tako sestavljena iz zadnjih 24 bitov vsakega izmed treh registrov. Podatki se vedno naložijo v spodnjih 256 bitov, nato pa se ob vsaki urini periodi premaknejo za 8 bitov (en piksel) v levo. Na ta način v modul za konvolucijo pošiljamo vsako periodo nov piksel in njegove sosede. Vsakič ko nam v pomikalnem registru ostanejo trije piksli



Slika 4.7: Modul za obdelavo slike.



Slika 4.8: Shema modula za obdelavo slike.

(spodnjih 256 bitov je praznih), zahtevamo nove podatke, tako da enota postavi izhod **request_data** na 1. Iz modula za konvolucijo rezultat nato potuje v 256-bitni pomikalni izhodni register. Ko je le-ta poln (napolnili smo vseh 32 pikslov), enota postavi izhod **data_rdy** na 1.

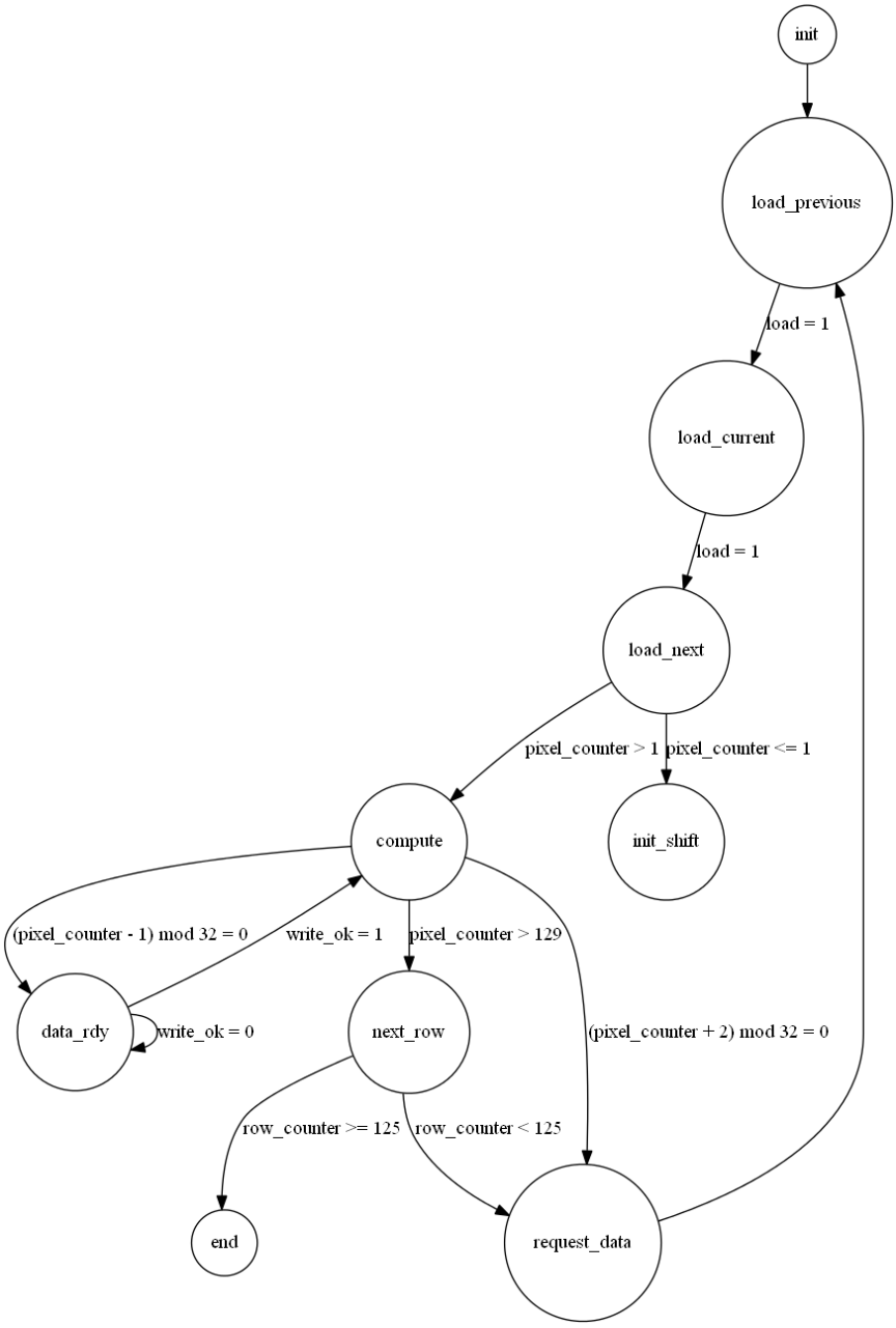
Delovanje modula je realizirano s končnim avtomatom (slika 4.9 ter tabela 4.2). Uporabili smo dva števca, enega za piksle (*pixel_counter*) ter enega za vrstice (*row_counter*). Začnemo v stanju *init*, kjer ponastavimo vrednosti števcev, vsebine podatkovnih registrov pa zapolnimo z ničlami. V stanjih *load_previous*, *load_current* ter *load_next* naložimo izseke slike iz prejšnje,

trenutne ter naslednje vrstice. Če se nahajamo na začetku vrstice, ko je vrednost števca pikslov 0, v stanju *init_shift* vsebino vseh treh registrov za vrstice pomaknemo za 16 bitov (dva piksla) v levo, tako da dobimo prvo matriko vrstice, z ničlami na levem robu. Če se ne nahajamo na začetku vrstice, se premaknemo direktno v stanje *compute*, kjer pričnemo vsako periodo pomikati vsebino podatkovnih registrov za en piksel (8 bitov) v levo, oziroma desno v primeru podatkovnega registra za rezultat. Ko izpraznemo spodnjih 256 bitov treh podatkovnih registrov (*pixel_counter + 2 mod 32 == 0*), se pomaknemo v stanje *request_data*, kjer postavimo izhodni signal **request_data** na 1, nato pa pričnemo z nalaganjem novih podatkov. Ko je izhodni podatkovni register poln (*pixel_counter - 1 mod 32 == 0*), upoštevali smo zamik ene periode zaradi množenja), se pomaknemo v stanje *data_rdy*, kjer postavimo izhodni signal **data_rdy** na 1, ter tako sporočimo, da so podatki pripravljeni. Tukaj čakamo na vhodni signal **write_ok**, ki nam pove, da so bili podatki uspešno zapisani, nato pa se pomaknemo nazaj v stanje *compute*. Ko pride vrednost števca *pixel_counter* do 130 (128 + 1 perioda zamika + 1 perioda za izpraznjenje cevovoda), se premaknemo v stanje *next_row*, kjer ponastavimo števec pikslov in s premikom v stanje *request_data* zahtevamo nove podatke. Če je števec vrstic (*row_counter*) enak 125 (štejemo od 0 do 127, zgornje in spodnje vrstice pa zaradi konvolucije ne računamo, saj gre za robna primera), se pomaknemo v stanje *end*, končali smo z računanjem.

4.5 Vrhovni modul

Vrhovni modul kontrolira prejemanje podatkov po UART, zapisovanje podatkov v RAM, branje podatkov iz RAM ter pošiljanje rezultata po UART. Povezuje krmilnik za RAM, enoto UART ter modul za procesiranje slike. Realiziran je kot končni avtomat (slika 4.10 ter tabela 4.3). Podoben je avtomatu modula za obdelavo slike, le da za prehode med stanji upošteva signale modula za obdelavo slike (npr. **request_data** ali **data_ready**).

Ena glavnih nalog vrhovnega modula je, da poskrbi za ustrezne po-



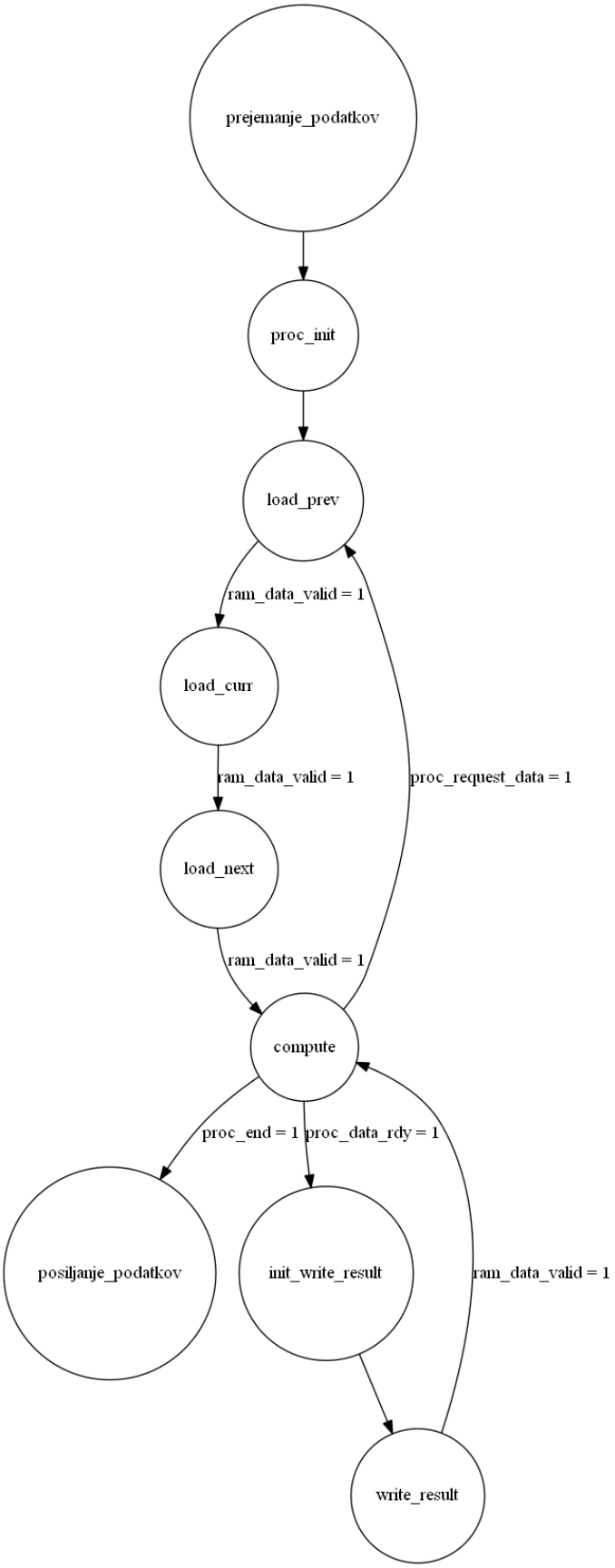
Slika 4.9: Diagram prehajanja stanj končnega avtomata modula za konvolucijo.

stanje	aktivni signali
init	pixel_count_rese, row_count_reset, request_data
load_previous	load_previous_row
load_current	load_current_row
load_next	load_next_row
init_shift	init_shift_row
compute	convolve3x3_clock_enable, pixel_count_up, shift_row
data_ready	data_rdy
request_data	request_data
next_row	pixel_count_reset, init_shift_row, data_out_reg_reset, row_count_up
end	end_proc

Tabela 4.2: Tabela aktivnih signalov v različnih stanjih avtomata modula za obdelavo slike.

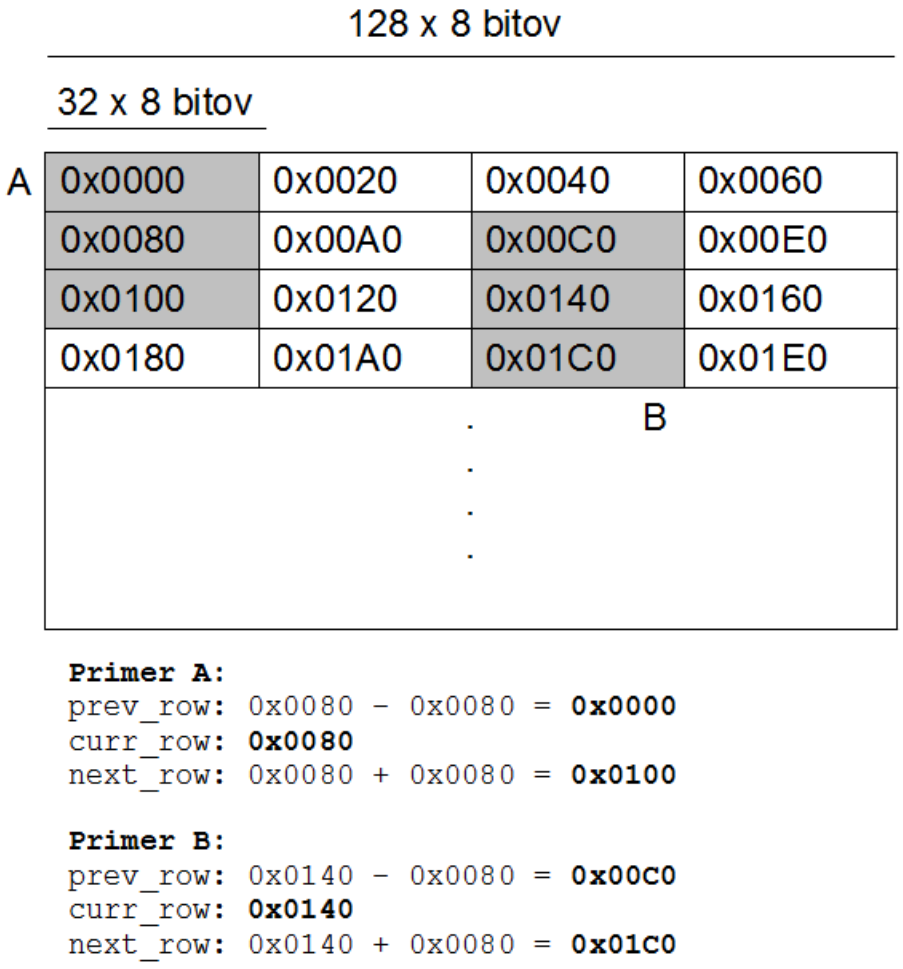
stanje	aktivni signali
proc_init	ram_addr_reset, ram_addr_read_proc_reset, batch_count_reset, proc_reset
load_prev	ram_en, (proc_load, če je load enak 1)
load_curr	ram_en, (proc_load, če je load enak 1)
load_next	ram_en, (proc_load, če je load enak 1)
compute	ram_addr_select
write_result_init	ram_addr_select, batch_count_up
write_result	ram_addr_select, ram_en, ram_wr_en, (proc_write_ok, ram_addr_write_count_up, če je ram_data_valid enak 0)

Tabela 4.3: Tabela aktivnih signalov v različnih stanjih avtomata vrhovnega modula.



Slika 4.10: Končni avtomat vrhovnega modula.

mnilniške naslove. Modul vsebuje dva registra za pomnilniški naslov, enega za branje ter enega za pisanje. Obema registroma ob zaključku branja ali pisanja prištevamo 32, saj to ustreza velikosti izhodnega registra krmilnika RAM, ki hrani po 32 8-bitnih besed (256 bitov). Pri branju podatkov iz pomnilnika moramo biti pozorni na branje prejšnje, trenutne in naslednje vrstice. V našem primeru, kjer je vrstica slike široka 128 pikslov, to pomeni, da moramo pri branju prejšnje vrstice, v stanju *load_prev*, od trenutnega naslova odšteti 128, pri branju naslednje, v stanju *load_next*, pa prišteti 128 (slika 4.11).



Slika 4.11: Dva primera naslovov podatkov, ki jih nalagamo v tri podatkovne registre enote za procesiranje slike (prejšnja, trenutna in naslednja vrstica).

Poglavje 5

Testiranje in rezultati

5.1 Ocena delovanja ter rezultati sinteze

5.1.1 Ocena rešitve z obrobljenimi kvadrati

Glavna slabost rešitve je velika prostorska poraba. Kot smo že zapisali, se zaradi dodatnih pikslov pri vsakem kvadratu slike potreben prostor za hranjenje poveča za najmanj 77%. Zaradi te slabosti se za implementacijo te rešitve nismo odločili, lahko pa podamo grobo oceno praktične izvedljivosti ter učinkovitost delovanja pri uporabi implementiranega krmilnika za bločni pomnilnik RAM. Za primer slike velikosti 128 $\times$ 128, pri kateri je velikost piksla 8-bit, bi potrebovali 484 kvadratov velikosti 8 $\times$ 8. Pojavi se namreč tudi problem deljivosti dimenzij slike. Vsaka stranica slike mora biti namreč deljiva s 6 (kar za naš primer ne drži), v nasprotnem primeru moramo število kvadratov zaokrožiti navzgor in zapisati ne povsem zapolnjene kvadrate, kar povzroči odvečno porabo prostora v pomnilniku. Tako se izkaže, da bomo za opisano sliko v pomnilniku namesto dejanskih 16384 bajtov za hrambo slike potrebovali kar 30976 bajtov (484 kvadratov $\times$ 64 pikslov), kar pomeni 89% povečanje porabe prostora. Iz pomnilnika beremo bloke velikosti 256 bitov, vsako branje pa zahteva 33 urinih period, ena perioda za postavitev ustreznih signalov, ter 32 period za branje 32 8-bitnih besed. Za 30976 bajtov bomo

morali izvesti 968 branj, za kar bomo porabili 31944 urinih period. Ker bi obdelano sliko nazaj v pomnilnik zapisali v običajni obliki, bi za zapisovanje porabili 16896 urinih period. Ozko grlo je branje ter pisanje v pomnilnik, tako da bi lahko sliko obdelali v 48840 urinih periodah. Hitrost bi bila sicer odvisna tudi od tega, kako bi rešili težavo določanja, kateri bloki in kateri deli blokov so prazni. Zaradi tega lahko vezje postane bolj kompleksno, ali pa bomo tudi za zapis sprocesirane slike porabili več časa in prostora, kot je potrebno. To je tudi eden od razlogov, da se za implementacijo te metode nismo odločili, saj je bila na prvi pogled prednost te zasnove ravno enostavnost.

5.1.2 Rezultati implementacije rešitve s pomikalnimi registri za piksle

Za implementacijo smo zaradi fleksibilnosti in relativne enostavnosti izvedbe izbrali rešitev s pomikalnimi registri. Delni rezultati sinteze so vidni na sliki 5.1. Vsako vrstico razen prve, druge, predzadnje in zadnje je treba prebrati trikrat, saj vedno potrebujemo trenutno, prejšnjo in naslednjo vrstico. Prvo vrstico preberemo enkrat, drugo pa dvakrat. Enako je z zadnjo ter predzadnjo vrstico. Beremo bloke velikosti 32 bajtov, v sliki velikosti 128 $\times$ 128 je takih blokov 512. Vsaka vrstica slike vsebuje 4 bloke. Potrebujemo torej 1512 branj iz pomnilnika, vsako pa traja 33 urinih period. Za branje bomo torej porabili 49896 period. Procesiranje ene vrstice traja 129 period, vsak piksel se sprocesira v eni periodi, eno periodo pa traja prehod na naslednjo vrstico. Za procesiranje torej porabimo 16254, saj sprocesiramo 126 vrstic, ker robnih ne procesiramo. Pri zapisovanju ni nobenih posebnosti, tako da to traja 16896 urinih period. Za procesiranje slike velikosti 128 $\times$ 128 tako porabimo 83046 urinih period. V poročilu sinteze je makismalna frekvenca ocenjena na 168,591MHz, kar nam da čas procesiranja okrog 500ms. Dejansko smo uporabili frekvenco 200MHz, kar pomeni, da se slika sprocesira v približno 410ms.

simple_top Project Status (03/07/2013 - 11:53:35)			
Project File:	simple_design.xise	**Parser Errors:**	No Errors
Module Name:	simple_top	**Implementation State:**	Placed and Routed
Target Device:	xc6vlx240t-1ff1156	**• Errors:**	No Errors
Product Version:	ISE 13.4	**• Warnings:**	16 Warnings (0 new)
Design Goal:	Balanced	**• Routing Results:**	All Signals Completely Routed
Design Strategy:	Xilinx Default (unlocked)	**• Timing Constraints:**	All Constraints Met
Environment:	System Settings	**• Final Timing Score:**	0 (Timing Report)

Device Utilization Summary				[-]
Slice Logic Utilization	**Used**	**Available**	**Utilization**	**Note(s)**
Number of Slice Registers	1,838	301,440	1%	
Number used as Flip Flops	1,799			
Number used as Latches	0			
Number used as Latch-thrus	0			
Number used as AND/OR logics	39			
Number of Slice LUTs	1,588	150,720	1%	
Number used as logic	1,557	150,720	1%	
Number using O6 output only	1,239			
Number using O5 output only	69			
Number using O5 and O6	249			
Number used as ROM	0			
Number used as Memory	0	58,400	0%	
Number used exclusively as route-thrus	31			
Number with same-slice register load	27			
Number with same-slice carry load	4			
Number with other load	0			
Number of occupied Slices	624	37,680	1%	
Number of LUT Flip Flop pairs used	1,816			
Number with an unused Flip Flop	145	1,816	7%	
Number with an unused LUT	228	1,816	12%	
Number of fully used LUT-FF pairs	1,443	1,816	79%	
Number of unique control sets	27			
Number of slice register sites lost to control set restrictions	65	301,440	1%	
Number of bonded IOBs	10	600	1%	
Number of LOCed IOBs	10	10	100%	

Slika 5.1: Del poročila o rezultatih sinteze.

Vzorčna slika:

Izsek iz tekstovne datoteke:

```
28351e1d110d0c0c0c0b0c0d0b0b0e0f0c0d26200b0c0d1512130f14110d0c09
1a3432434b41351b120c0c0a0c0d0d0e111212426100b0d151c0d14250d100c
182a37373d576a51402c25251c161915141 61f18213116 0d0d1a180f282d2328
080714391b1c25434953536412d160f1427367c8e6b586362746f596084727e
110f133730224067687 05b4657453f1b2354639598887c7246596b58495c5b91
2d1d223639394e707e7a625b606c54264f8e8ba2a08d8c6a37446757302f5c89
845a585620193943473d3d391b213c1406000601000102130b0c0c0d0e101012
8c6b6d50330f1b1f2f2a38261e3e44350e050c0303000c130608090b1015100e
8f8a754b28090e112623422e3c57605731220d0004041b0e0708080c0e100c0d
1317151113151315131414151517171615151211111212121416161718181918
141e161112121414121212121315171616151211111111121313161618191818
182514121213111213121012121215151514121111111111112141415161818 8
1a3432434b41351b120c0c0a0c0d0d0e111212426100b0d151c0d14250d100c
46312d2f3138464e54594f565a403b3f321f1e1f19283b1f0c14151e1e444745
46312d2f3138464e54594f565a403b3f321f1e1f19283b1f0c14151e1e444745
110f133730224067687 05b4657453f1b2354639598887c7246596b58495c5b91
4d3733413d47636c817d635c57734e477b9b9cafa77b714d323e6852331f597e
4d3733413d47636c817d635c57734e477b9b9cafa77b714d323e6852331f597e
8c6b6d50330f1b1f2f2a38261e3e44350e050c0303000c130608090b1015100e
989e855b1e1013292b3844586d637e705737190c0111261b18130b08080a0b0a
989e855b1e1013292b3844586d637e705737190c0111261b18130b08080a0b0a
141e161112121414121212121315171616151211111111121313161618191818
1b2013111112111211110f1112111215161211101011111312141516171919 18
1b2013111112111211110f1112111215161211101011111312141516171919 18
.
.
.
```

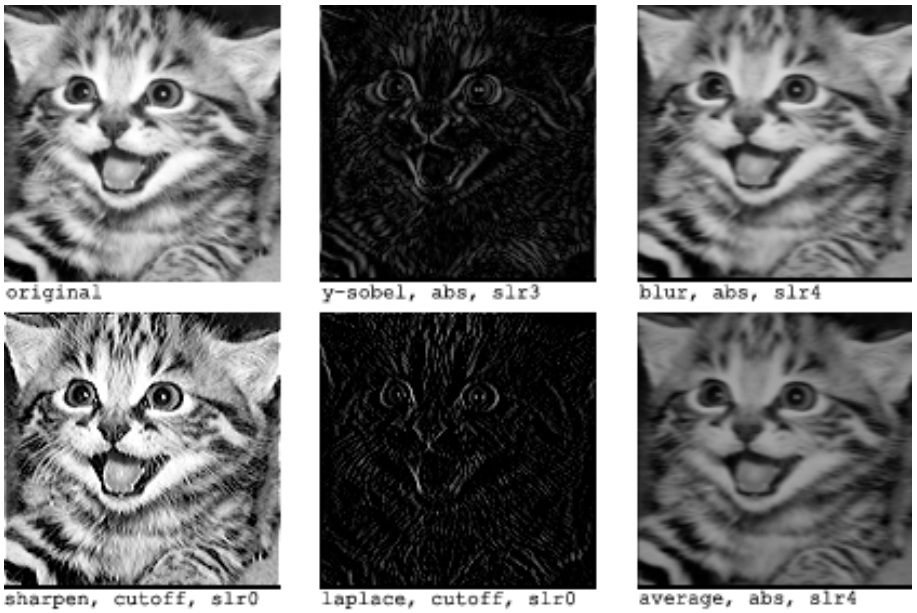
Slika 5.2: Vzorčna slika ter izsek tekstovne datoteke izpeljana iz nje.

5.1.3 Poskus implementacije s hranjenjem treh vrstic slike

Poskusili smo tudi z implementacijo nadgradje zgornje rešitve, vendar se je izkazalo, da za naše pogoje ni izvedljiva. Potrebujemo namreč tri pomikalne registre velikosti 1024 bitov. Pomikalni registri so za vezja FPGA prostorsko zahtevni, še posebej tisti večjih dimenzij. Pri poskusu implementacije smo tako ugotovili, da bi ta porabila več kot 100% sredstev, ki so nam bila na voljo, tako da je žal ni bila izvedljiva, bi pa lahko prišla v poštev pri zelo majhnih slikah.

5.2 Testiranje na simulatorju

Zaradi enostavnejšega in hitrejšega testiranja, smo enoto za procesiranje slik najprej testirali z uporabo Xilinxovega simulatorja ISim. Sprva smo vzorčno sliko pretvorili v tekstovno datoteko, ki je vsebovala podatke v heksadecimalnem zapisu, v enakem vrstnem redu, kot jih modul za obdelavo slike prejema v končni realizaciji (slika 5.2). Vsaka vrstica v tekstovni datoteki predstavlja en 256-bitni blok podatkov, ki ga prejme modul.



Slika 5.3: Rezultati testiranja na simulatorju. Pod vsako sliko je zapisan uporabljen filter, metoda normalizacije ter normalizacijski pomik (npr. „slr3“ pomen trikratni logični pomik v desno).

Pri testiranju smo nato uporabili VHDL knjižnjico *ieee.std_logic_textio*, ki nam omogoča uporabo tekstovnih datotek med testiranjem. Rezultate, ki jih smo jih dobivali iz modula, smo zapisovali (slika 5.4) v novo tekstovno datoteko. Dobljeno tekstovno datoteko smo nato z majhnim Java programom pretvorili v binarno slikovno datoteko PGM. Preizkusili smo nekaj različnih konvolucijskih jeder, rezultati so vidni na sliki 5.3. Opazimo lahko, da so pri filtrih, kjer je potrebno uporabiti normalizacijski pomik, slike lahko nekoliko temnejše. To je rezultat tega, da uporabljamo poenostavljeno normalizacijo; logični pomik v desno namesto pravega deljenja. To pomeni, da moramo včasih deliti z večjim številom, kot je treba, rezultat pa je temnejša slika. Za tako rešitev smo se odločili zaradi večje hitrosti in enostavnosti.

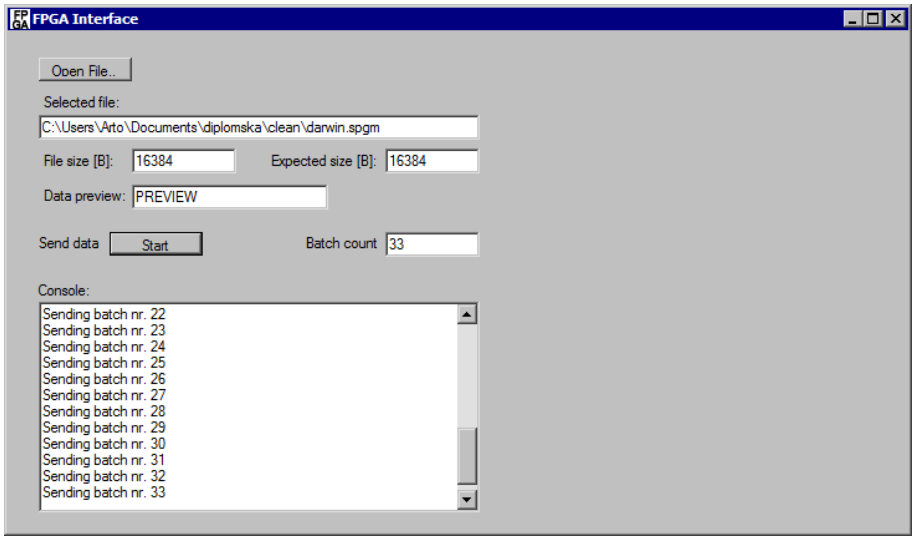
```
write_proc: process
begin
  while true loop
    wait until data_rdy = '1';

    report "Writing";
    hwrite(OL, data_out);
    writeLine(outfile, OL);

    end loop;
end process;
```

Slika 5.4: Proces za pisanje v datoteko v naši testni datoteki VHDL.

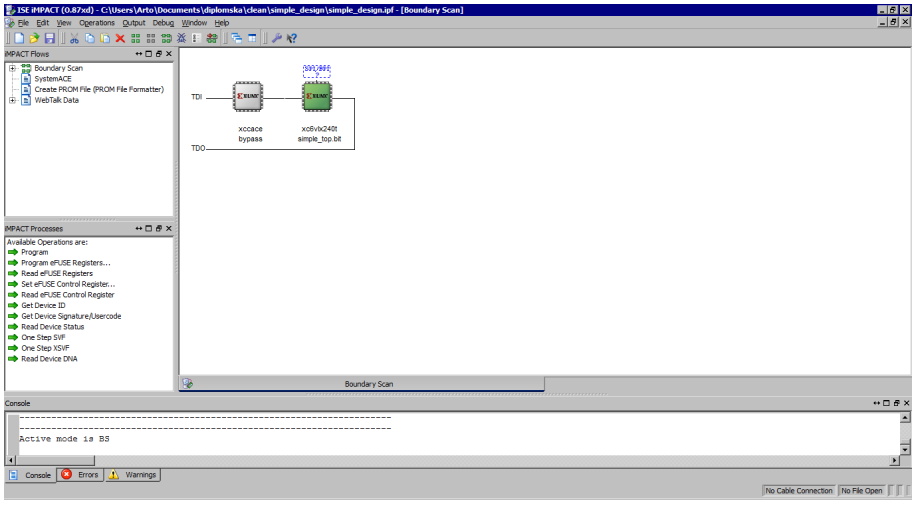
5.3 Testiranje na platformi Xilinx Virtex-6

Ko smo modul stestirali na simulatorju, smo ga preizkusili še na platformi Xilinx Virtex-6. To je vključevalo tudi testiranje pošiljanja ter prejemanja podatkov po UART (kar sicer ni del naše diplomske naloge) ter zapisovanje in branje v bločni pomnilnik RAM. Po programiranju platforme v programu Xilinx ISE Impact(slika 5.6), smo na platformo podatke prek serijskega vmesnika poslali s pomočjo programa napisanega v C# (slika 5.5 avtor programa je Matevž Bizjak [5]). Program smo morali pred uporabo malo prilagoditi, saj je bilo možno le pošiljanje tekstovnih datotek, kar za naše potrebe ni bilo primerno. Program smo spremenili tako, da je prek njega možno pošiljati vse vrste datotek.

Prejemanje, procesiranje ter pošiljanje se je izvedlo pravilno. Rezultate smo primerjali še z rezultati, ki jih dobimo, ko enake konvolucijske matrike uporabimo v grafičnem programu GIMP. Vidimo lahko, da se ujemajo z rezultati, ki smo jih dobili na platformi Xilinx Virtex-6, razlika je le v tem, da GIMP drugače obravnava robne primere, zato pride do manjših razlik na robovih. Dve primerjavi sta vidni na sliki 5.7.



Slika 5.5: Program za pošiljanje podatkov prek serijskega vmesnika.



Slika 5.6: Programiranje platforme Virtex-6 v Xilinx ISE Impact.

Slika 5.7: Primerjava rezultatov, ki jih dobimo z uporabo konvolucijske matrike v GIMP-u in na naši implementaciji na platformi Xilinx Virtex-6.

Poglavje 6

Zaključek

Uspelo nam je izdelati delujočo in fleksibilno implementacijo procesiranja slik s pomočjo konvolucijskih jeder. Za metodo z obrobljenimi kvadrati se nismo odločili, saj porabi veliko več prostora za hrambo slike v pomnilniku, prav tako pa se je prvemu vtisu navkljub izkazala za bolj problematično. Metoda s hranjenjem treh vrstic slike na žalost zaradi zahtevnosti velikih pomikalnih registrov v vezjih FPGA žal ni bila izvedljiva za uporabljeno velikost slike. Odločili smo se za običajno implementacijo s pomikalnimi registri. Implementirali smo štiri module. Preprost krmilnik za bločni RAM skrbi za komunikacijo z bločnim pomnilnikom RAM ter omogoča enostavno zamenjavo z drugimi krmilniki. Modul za konvolucijo izvaja konvolucijo bloka slike velikosti 3×3 piksle s konvolucijskim jedrom enake velikosti. Modul za obdelavo slike s pomočjo modula za konvolucjio skrbi za obdelavo celotne slike. Vrhovni modul povezuje modul za obdelavo slike, bločni RAM ter modul UART. Implementacijo smo sprva testirali v simulatorju, nato pa tudi na platformi Xilinx Virtex-6. Rezultate, ki smo jih prejeli od platforme smo primerjali z rezultati uporabe enakega konvolucijskega jedra v programu GIMP in ugotovili, da se ujemajo. Razlika je bila le v obravnavi robnih pikslov, kar smo tudi pričakovali. Mogoča je uporaba poljubnih konvolucijskih jeder velikosti 3×3. Za večjo splošno uporabnost naše implementacije, bi jo lahko nadgradili tako, da bi podatke o velikosti slike ter konvolucijsko jedro prejela

prek komunikacijskega vmesnika skupaj s sliko. Možni prilagoditvi bi bili tudi obdelava več različnih slik z istim konvolucijskim jedrom ali uporaba več različnih konvolucijskih jeder na eni sliki. Omogočili smo enostavno nadgrajevanje z različnimi krmilniki RAM, kar bi bilo potrebno ob morebitni praktični uporabi našega vezja. Za praktično uporabo bi bilo treba izbrati tudi hitrejši način prenosa, vendar to ni bil namen te diplomske naloge, saj je modul UART služil le kot pomoč pri demonstraciji delovanja.

Slike

Tabele

Literatura

[1] R. Bracewell, *The Fourier Transform and Its Applications*, str. 25-50, 1965.

[2] Wikipedia, *Convolution*, 2013, Dostopno na: http://en.wikipedia.org/w/index.php?title=Convolution&oldid=533480706

[3] Peter J. Ashenden, *Digital Design, An Embeded Systems Approach Using VHDL*, str. 400–423, 2007.

[4] ML605 Product Brief, Dostopno na: http://www.xilinx.com/publications/prod_mktg/ml605_product_brief.pdf

[5] Matevž Bizjak, *Implementacija pomnilniškega vmesnika v FPGA : diplomsko delo*, 2012