UNIVERZA V LJUBLJANI
FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Matevž Bizjak

Implementacija pomnilniškega vmesnika v FPGA

DIPLOMSKO DELO
NA UNIVERZITETNEM ŠTUDIJU

Mentor: prof. dr. Patricio Bulić

Ljubljana, 2012

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: 01825/2012
Datum: 02.04.2012

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

Kandidat: **MATEVŽ BIZJAK**

Naslov: **IMPLEMENTACIJA POMNILNIŠKEGA VMESNIKA V FPGA**
IMPLEMENTATION OF A MEMORY INTERFACE IN FPGA

Vrsta naloge: Diplomsko delo univerzitetnega študija

Tematika naloge:

Implementirajte vmesnik za pomnilnik DDR SDRAM v vezju FPGA. Za implementacijo pomnilniškega vmesnika uporabite orodje MIG proizvajalca Xilinx. Poleg pomnilniškega vmesnika implementirajte vmesnik za serijsko komunikacijo UART, s katerim bo mogoče iz osebnega računalnika komunicirati s čipom FPGA ter dostopati do pomnilniškega vmesnika. Za implementacijo takega sistema uporabite vezje FPGA Virtex-6 na razvojni ploščici ML-605. Na osebnem računalniku izdelajte uporabniški grafični vmesnik, ki bo omogočal prenos podatkov (slik) na sistem na čipu FPGA. Grafični vmesnik načrtujte v okolju Microsoft Visual Studio 2010.

Mentor:
prof. dr. Patricio Bulić

Dekan:
prof. dr. Nikolaj Zimic

Fakulteta za računalništvo in informatiko • UNIVERZA V LJUBLJANI • 2

IZJAVA O AVTORSTVU

diplomskega dela

Spodaj podpisani Matevž Bizjak,

z vpisno številko 63070056,

sem avtor diplomskega dela z naslovom:

Implementacija pomnilniškega vmesnika v 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 28.06.2012 Podpis avtorja:

Zahvala

Za podporo in pomoč pri delu se zahvaljujem mentorju, prof. dr. Patriciu Buliću in svoji družini za vso vzpodbudo ter pozitivno mišljenje.

Kazalo

Seznam uporabljenih kratic in simbolov

FPGA - Field Programmable Gate Array
HDL - Hardware Description Language
LCD - Liquid Crystal Display
LED - Light Emitting Diode
JTAG - Joint Test Action Group
USB - Universal Serial Bus
VGA - Video Graphics Array
PCI - Peripheral Component Interconnect
ASIC - Application-Specific Integrated Circuit
CLB - Configurable Logic Blocks
IOB - Input/Output Block
VHDL - VHSIC Hardware Description Language
VHSIC - Very-High-Speed Integrated Circuits
IEEE - Institute of Electrical and Electronic Engineers
RTL - Register-Transfer Level
VHSIC - Very-High-Speed Integrated Circuits
DDR - Double Data Rate
SDRAM - Synchronous Dynamic Random-Access Memory
DIMM - Dual In-line Memory Module
SODIMM - Small Outline Dual In-line Memory Module
CAS - Column Address Strobe
RAS - Row Address Strobe
NOP - No Operation
MRS - Mode Register Set
WR - Write Recovery
WTR - WRITE to READ
CCD - CAS#-to-CAS#
PLL - Phase-Locked Loop

UART - Universal Asynchronous Receiver/Transmitter
MIG - Memory Interface Generator
UCF - User Constraints File
RTL - Register-Transfer Level
FIFO - First In, First Out
ASCII - American Standard Code for Information Interchange

Povzetek

Obstajajo računalniški problemi, kjer je procesiranje množice podatkov na splošnonamenskih procesorjih časovno neučinkovito in potratno. Integrirana vezja FPGA (angl. *Field-Programmable Gate Array*) uporabniku omogočajo, da sam opiše delovanje vezja, s tem pa je mogoče doseči procesiranje podatkov na strojnem nivoju. Cilj diplomske naloge je izdelava rešitve, ki omogoča prenos podatkov na vezje FPGA ter shranjevanje v pomnilnik z naključnim dostopom - RAM (angl. *Random Access Memory*), ki se nahaja na razvojni plošči s čipom FPGA. Izdelava logike, ki bi podatke na FPGA-ju tudi procesirala, ni bila cilj diplomske naloge.

Za namen diplomske naloge imamo na voljo dve razvojni plošči čipom FPGA proizvajalca Xilinx, enostavnejšo na osnovi FPGA čipa *Spartan-3A* ter kompleksnejšo na osnovi čipa *Virtex-6*. Izkaže se, da razvojna plošča Spartan-3A ni ustrezna za implementacijo zastavljenega cilja, zato je rešitev razvita za razvojno ploščo z Virtex-6. V diplomski nalogi je predstavljena uporaba pomnilniškega krmilnika z orodjem MIG (angl. *Memory Interface Generator*), implementacija modula, ki omogoča serijsko komunikacijo z vezjem FPGA ter povezava obeh delov s nadzornim vezjem, ki usmerja delovanje. Izdelan je tudi grafični uporabniški vmesnik, ki uporabniku omogoča izbiro vhodne datoteke s podatki, pošiljanje na FPGA ter shranjevanje vrnjenih podatkov iz vezja FPGA v novo datoteko.

S programsko simulacijo delovanja izdelane rešitve ter praktičnim preizkusom delovanja ugotovimo, da je uporaba pomnilnika RAM na razvojnih ploščah proizvajalca Xilinx za uporabnika enostavna in učinkovita. Izkaže se, da je prenos podatkov s serijsko povezavo počasen, zato bi bilo mogoče delo nadgraditi z implementacijo hitrejše komunikacije. Izdelano rešitev je mogoče preprosto dopolniti z logiko, ki bi sprejete podatke iz računalnika še procesirala in v tem primeru vrnila procesirane podatke.

Ključne besede:

FPGA, pomnilnik DDR SDRAM, krmilnik pomnilnika, serijska komunikacija, UART, grafični uporabniški vmesnik

Abstract

There are certain computational problems, where processing data using general purpose computer processing units is ineffective and time-consuming. FPGA (Field-Programmable Gate Array) integrated circuits allow designers configuring their operations for specific problems. This way, data can be processed on a hardware level. The purpose of this thesis is to design and implement a solution that will allow sending data from computer to FPGA and saving it in RAM (Random Access Memory), which is embeded in the FPGA development board. No logic, that could process data on FPGA will be implemented.

Two Xilinx FPGA development boards are used for the purpose of this thesis - a simple board based on Spartan-3A FPGA family and an advanced board based on Virtex-6 family. It turns out that Spartan-3A based development board is unsuitable for the intended design, therefore Virtex-6 based board is used. The thesis describes the implementation of memory controller using MIG (Memory Interface Generator) tool, the implementation of module, which allows serial communication with the FPGA board and connection of both modules with logic, that controls the complete design. A graphical user interface, that allows selecting an input file, sending it to FPGA and writing processed data to output file is also designed.

Design simulation and practical testing of designed solution confirm that interfacing RAM with Xilinx FPGA development boards is user friendly and efficient. We come to the conclusion that serial transmission of data is slow, so additional improvements could be made in this field. The final design could easily be upgraded with processing logic that would process received data from computer.

Key words:

FPGA, DDR SDRAM memory, memory controller, serial communication,

Poglavje 1

Uvod

Obstajajo računalniški problemi, kjer je potrebno procesirati veliko množico podatkov, vendar je procesiranje teh podatkov na splošnonamenskih procesorjih časovno zahtevno. Vezja FPGA omogočajo uporabniku, da sam opiše delovanje vezja in s tem doseže učinkovito in hitro procesiranje podatkov. Vezje, ki je izdelano za specifičen namen, zastavljeno nalogo praviloma opravi mnogo hitreje kot splošnonamenski procesor v osebnem računalniku, delovni postaji ali gruči računalnikov.

Namen diplomske naloge je izdelati grafičen uporabniški vmesnik, ki omogoča prenos podatkov na vezje FPGA in shranjevanje v pomnilnik z naključnim dostopom - RAM (angl. *Random Access Memory*), ki se nahaja na razvojni plošči s čipom FPGA. Izdelava logike, ki bi podatke na FPGA-ju tudi procesirala, ni cilj diplomske naloge, je pa to možna naknadna razširitev. Izdelana rešitev bo omogočala tudi prenos podatkov iz FPGA nazaj na računalnik. Za komunikacijo in prenos podatkov med vezjem FPGA in računalnikom bo uporabljena serijska povezava. Na FPGA-ju bo zato potrebno implementirati tudi asinhroni oddajnik in sprejemnik - UART (angl. *Universal Asynchronous Receiver/Transmitter*). Za izdelavo diplomske naloge imamo na voljo dve razvojni plošči z vezjem FPGA proizvajalca Xilinx - enostavno ter zmogljivejšo.

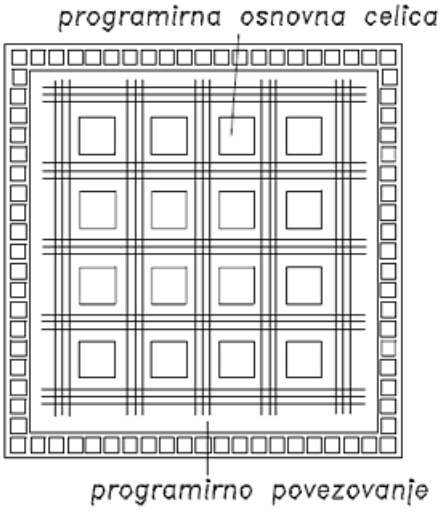
V drugem poglavju bo podano teoretično ozadje delovanja vezij FPGA, opis razvojnih plošč, ki jih imamo na voljo za implementacijo rešitve ter opis programiranja vezij FPGA. V tretjem poglavju bo opisano delovanje RAM-a in podrobnejši opis uporabljenega modula. V četrtem poglavju bo predstavljen načrt implementacije celotne rešitve in kratek opis neuspelega poskusa implementacije na šibkejši razvojni plošči. Peto poglavje je posvečeno implementaciji na zmogljivejši razvojni plošči. V prvem delu bo predstavljena izdelava krmilnika pomnilnika, v drugem izdelava modula UART ter v tretjem

povezava vseh modulov v zaključeno celoto. Sledi šesto poglavje, ki opisuje simulacijo delovanja izdelane rešitve. V zadnjem poglavju je predstavljena izdelava grafičnega vmesnika.

Poglavje 2

Splošno o FPGA

Platforma, za katero bomo razvili rešitev problema, se imenuje FPGA. FPGA je integrirano vezje, ki vsebuje programabilne logične celice, ki jih je mogoče med seboj povezovati. To prikazuje slika 2.1. Smisel čipov FPGA je, da uporabnik sam določi logično delovanje celotnega vezja s tem, ko opiše delovanje posameznih celic in tudi povezav med njimi. Za opis delovanja se najpogosteje uporablja HDL (angl. *Hardware Description Language*) jezike .



Slika 2.1: Splošni model zgradbe FPGA čipa - vidne so programirljive osnovne celice in povezave med njimi. Vir [2].

Programabilni čip FPGA je praviloma nameščen na razvojno ploščo, ki vključuje še nekatere dodatne periferne priključke ali naprave. Čip FPGA

programiramo preko JTAG (angl. *Joint Test Action Group*) vmesnika. Na razvojnih ploščah najdemo različno zmogljive čipe FPGA - od najpreprostejših do kompleksnih z mnogo logičnimi celicami in vhodno/izhodnimi povezavami. Uporabnost razvojne plošče je razširjena z različnimi komponentami, s katerimi čip FPGA komunicira in jih nadzoruje. Pogoste komponente, ki jih najdemo na razvojnih ploščah so zaslon LCD (angl. *Liquid Crystal Display*), kontrolne LED diode (angl. *Light Emitting Diode*), preproste tipke, stikala, različne vrste pomnilnikov itd. Med priključki, ki jih razvojne plošče vsebujejo, pa pogosto najdemo USB (angl. *Universal Serial Bus*), Ethernet, serijski, VGA (angl. *Video Graphics Array*), PCI (angl. *Peripheral Component Interconnect*) priključek, priključke za eksterne urine signale in še različne razširitvene priključke. Proizvajalci razvojnih plošč kot sta Xilinx in Altera nudijo tudi razvojna okolja, s katerimi z uporabo HDL jezikov razvijamo rešitve za FPGA.

Prednosti FPGA po [1] so predvsem:

- Možnost dodajanja novih funkcionalnosti po končanem razvoju
- Možnost delne spremembe vezja
- Nizki enkratni stroški razvoja v primerjavi z vezji, izdelanimi za določen namen

Posledica teh lastnosti je vedno večja uporaba vezij FPGA, ki so na nekaterih področjih začela nadomeščati vezja ASIC (angl. *Application-specific integrated circuit*) - vezja, ki so izdelana za določen namen. To velja predvsem za manjše serije čipov, saj so tako stroški nižji. Čipe FPGA se uporablja na področjih kot so digitalno procesiranje signalov, kriptografija, letalska industrija, bioinformatika, medicina in tudi kot prototipno okolje za izdelavo ASIC vezij.

2.1 Zgradba Xilinx FPGA

Pri izdelavi naloge bomo delali z vezji FPGA proizvajalca Xilinx. Vezja FPGA poleg osnovnih logičnih celic vsebujejo tudi dele, ki so v bistvu ASIC vezja, npr. pomnilnik, logika za upravljanje z urinim signalom itd. Glavne komponente vezja FPGA so torej:

- Konfigurabilni logični bloki - CLB (angl. *Configurable Logic Blocks*)

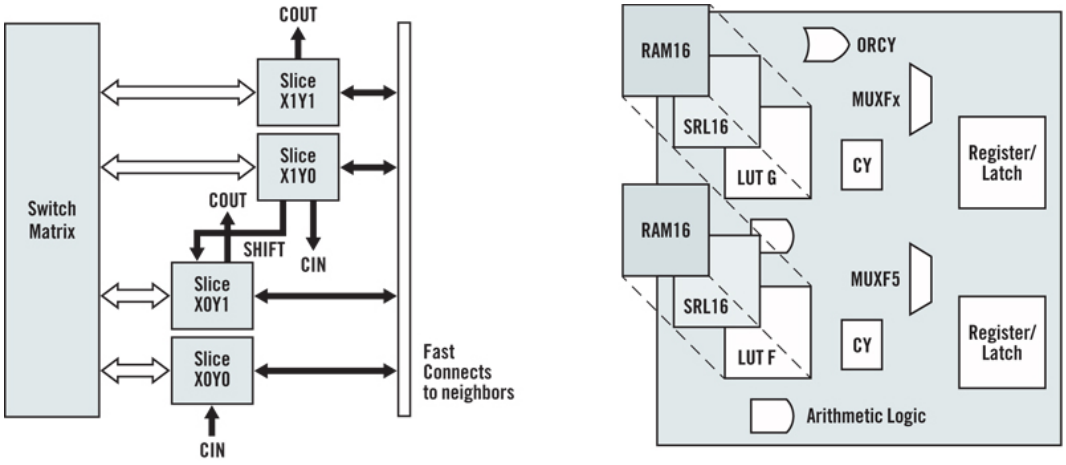
- Povezovalni kanali
- Vhodno/izhodni bloki - IOB (angl. *Input/Output Block*)
- Pomnilnik
- Logika za upravljanje z uro

Čip FPGA je sestavljen iz mreže med seboj povezanih CLB-jev. CLB so osnovne logične celice, katerim lahko določamo delovanje. En CLB je sestavljen iz večjega števila *rezin* (angl. *slice*), ki so povezane s preklopno matriko (angl. *switch matrix*) s 4 ali 6 vhodi, odvisno od čipa. Vsaka rezina vsebuje 4 iskalne tabele - LUT (angl. *Look-Up Table*), flip-flope, multiplekserje ter aritmetično logiko in pomikalne registre [5]. LUT je pomnilnik tipa statični RAM, katerega organizacija je odvisna od družine čipa FPGA. Lahko gre za organizacijo 16x1, 32x1 ali 64x1. Z enim LUT-om lahko realiziramo poljubno 4-vhodno, 5-vhodno ali pa 6-vhodno logično funkcijo. Ker gre za statični RAM, FPGA vezje ob izgubi napajanja izgubi tudi vsebino LUT-ov in s tem tudi delovanje, ki smo ga programirali. Potrebno je torej ponovno programiranje čipa. Logične funkcije, ki jih programer določi za izvajanje na FPGA, so torej realizirane v LUT-ih in s pomočjo multiplekserjev ter aritmetične logike v rezinah. Kompleksnejše logične funkcije se realizirajo tako s povezovanjem posameznih rezin, kot tudi CLB-jev. Del rezin se lahko uporabi tudi kot *porazdeljen pomnilnik* RAM, v tem primeru se podatki hranijo v LUT-ih. Opisana zgradba CLB-ja je prikazana na sliki 2.2 - na levem delu je prikazan CLB s 4 rezinami, na desni pa vsebina posamezne rezine.

Na čipu FPGA se nahaja na tisoče CLB-jev, zato je potrebno poskrbeti za učinkovite povezave med samimi CLB-ji ter tudi za povezave z vhodno/izhodnimi bloki - IOB-ji. Hitre vertikalne in horizontalne povezave so že prisotne na čipu, najbolj optimalno pot pri povezovanju komponent pa opravi programska oprema, ki ta korak tudi skrije pred uporabnikom [4]. Pri tem skrbi, da so povezave optimalne in ne povzročajo prevelikega zamika signalov.

IOB so bloki, ki omogočajo pretvorbo signalnih nivojev, priklop zunanjih naprav na čip FPGA in zajem zunanjih signalov. Omogočajo komunikacijo z logiko v čipu FPGA. Pretvorbo signalnih nivojev vršijo v skladu z določenimi vhodno/izhodnimi standardi. IOB-ji so organizirani v *banke*, kjer vsaka banka vsebuje določeno število nožic na čipu.

Omenili smo že, da del LUT-ov lahko deluje kot porazdeljen pomnilnik. Poleg tega večina čipov FPGA nudi še bločni pomnilnik RAM. Velikost pomnilnika je le nekaj 10 Mbit, kar je premalo za potrebe resnejšega procesiranja



Slika 2.2: Zgradba CLB-ja in posamezne rezine s komponentami, ki jih vsebuje. Vir [4].

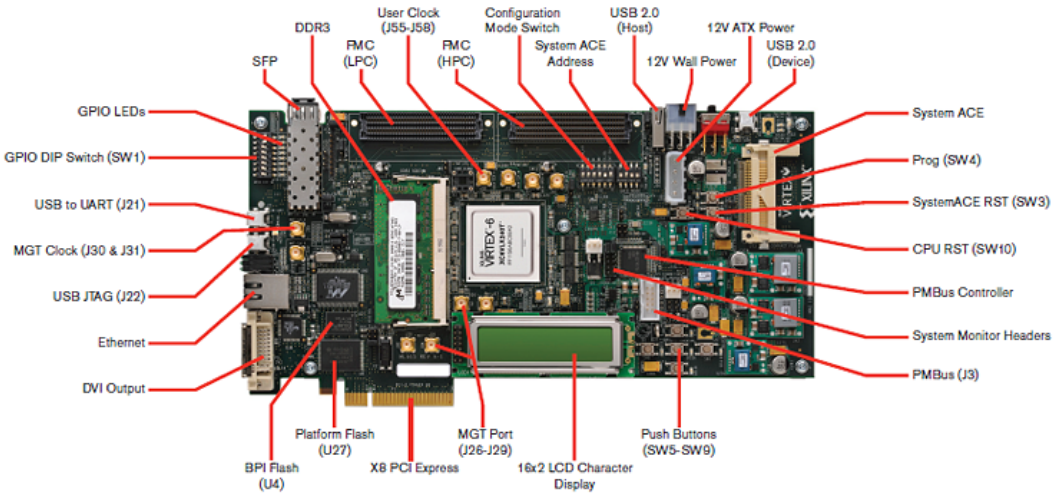
podatkov, npr. slik in zvoka. Zaradi premajhne velikosti bločnega pomnilnika ne bomo uporabljali.

Zadnja glavna komponenta čipov FPGA je del, ki upravlja z urinim signalom. Imenuje se *digitalni upravljalnik z uro - DCM* (angl. Digital clock Manager). Upravljalniki z uro omogočajo ustvarjanje velikega števila različnih frekvenc ure iz vhodne ure ter skrbijo za zmanjševanje faznega zamika (angl. *jitter*). Čipi vsebujejo več *globalnih povezav* (angl. *global clock lines*) za uro, ki so glavne linije, po katerih potuje urin signal. Globalne povezave dosežejo prav vsako pomnilno celico na čipu ter vse signale za resetiranje in omogočanje ure [5]. Obstajajo še *področne ure* (angl. *regional clocks*), vsak čip pa vsebuje več področij. Vsaka področna ura dosega vse elemente v svojem področju.

2.2 Uporabljeni razvojni plošči FPGA

Na voljo imamo dve razvojni plošči proizvajalca Xilinx. Prva je *Spartan-3A FPGA Starter Kit*, druga pa *Virtex-6 FPGA ML605 Evaluation Kit*. Prva, ki temelji na seriji *Spartan-3* čipov FPGA je bolj osnovna in preprostejša. ML605 je novejša in mnogo zmogljivejša in temelji na seriji čipov FPGA *Virtex-6*. Razvojno ploščo ML605 lahko vidimo na sliki 2.3, kjer so prikazane tudi glavne komponente in priključki, ki jih plošča vsebuje.

V tabeli 2.1 lahko opazimo, da je razvojna plošča s čipom Virtex-6 mnogo bolj zmogljiva kot plošča s čipom Spartan-3. ML605 vsebuje novejši pomnilnik



Slika 2.3: Razvojna plošča Virtex-6 ML605 in nekatere osnovne komponente. Vir [3].

DDR3 v primerjavi z DDR2, ki ga vsebuje plošča Spartan-3A. Plošči se razlikujeta tudi v količini pomnilnika RAM, vendar je za povprečno procesiranje podatkov dovolj velik že manjši izmed njiju.

	Spartan-3A	**Virtex-6 ML605**
Čip FPGA	XC3S700A-FG484	XC6VLX240T-1FFG1156
Število logičnih celic	13.248	241.152
Vgrajen oscilator	50 MHz	200 MHz - diferencialni
RAM pomnilnik	64 MB DDR2 SDRAM	512 MB DDR3 SDRAM
Frekvenca RAM	125-133 MHz	400 MHz
Serijska komunikacija	DB-9 priključek	USB-serial most

Tabela 2.1: Primerjava lastnosti obeh FPGA razvojnih plošč, ki so ključne za izdelavo rešitve problema.

Razlika med ploščama je tudi pri izvedbi serijske komunikacije po RS-232 standardu. Spartan-3A vsebuje standardni DB-9 priključek, kar pomeni, da mora tudi naš računalnik vsebovati tak priključek. Problem se pojavi, ker moderni osebni in prenosni računalniki tega priključka nimajo več. To je rešljivo z nakupom posebnega adapterja, ki serijski priključek DB-9 pretvori v priključek

USB, ob tem pa še pretvarja serijske signale na standardni USB protokol. Na računalniku se ta USB vhod predstavi tako, kot da imamo priključen serijski priključek DB-9. Razvojna plošča Virtex-6 ML605 vsebuje drugačno rešitev - adapter oz. most iz serijskega na USB priključek vsebuje že sama plošča, tako da jo lahko preprosto priključimo na računalnik preko USB kabla.

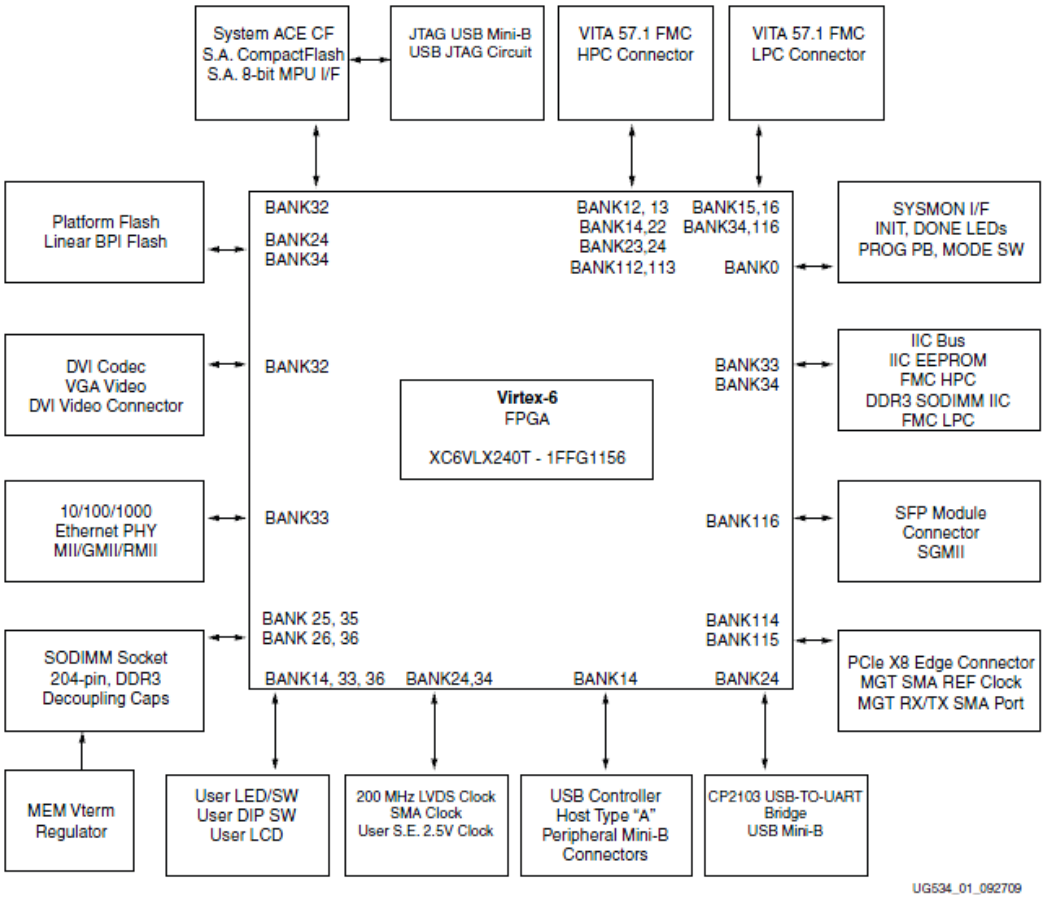
Slika 2.4 prikazuje povezavo čipa FPGA na plošči Virtex-6 ML605 s komponentami in razširitvenimi priključki na plošči. Vsaka komponenta je s čipom povezana z določenimi nožicami, te pa so razdeljene na t.i. *banke*. V mislih moramo imeti, da ob uporabi določene komponente nožice, ki so predvidene zanjo, postanejo zasedene, zato jih ni mogoče uporabiti za kakšen drug namen. Tak konflikt je mogoč, če želimo dostopati do DDR3 pomnilnika in hkrati tudi uporabljati 2-vrstični LCD zaslon na plošči. Problem je v tem, da obe komponenti zasedata nekatere nožice iz *banke 36*, s tem pa uporaba obeh komponent hkrati ni mogoča.

Xilinx nam nudi razvojno okolje *Xilinx ISE*, s katerim lahko razvijamo rešitve z uporabo jezika *VHDL* ali *Verilog* za obe razvojni plošči. Uporabljena verzija Xilinx ISE je 13.3.

2.3 Programski jezik VHDL

VHDL (angl. *VHSIC Hardware Description Language*) je jezik, s katerim se opisuje delovanje digitalnih elektronskih sistemov. Njegovi začetki segajo v leto 1980, ko je ameriška vlada začela s programom VHSIC (angl. *Very-High-Speed Integrated Circuits*), ki naj bi pospešil razvoj hitrih integriranih vezij. Jezik VHDL je kmalu zatem postal IEEE (angl. *Institute of Electrical and Electronic Engineers*) standard. VHDL omogoča opisovanje zgradbe vezja, oziroma kako je to sestavljeno iz manjših elementov in kako so ti elementi med seboj povezani [6]. Prav tako omogoča tudi simulacijo vezja, preden je to poslano v proizvodnjo. To močno zniža stroške načrtovanja integriranih vezij. Elektronska vezja lahko opisujemo kot module z vhodi in izhodi, kjer so izhodi funkcija vhodov. Kompleksnejša vezja si lahko predstavljamo kot več med seboj povezanih modulov. V jeziku VHDL so taki moduli poimenovani *entitete* (angl. *entities*), vhodi in izhodi pa so imenovani *vrata* (angl. *ports*). V VHDL je modul sestavljen iz večjega števila podmodulov, ti pa so *primerki* (angl. *instance*) ene entitete. Primerki so med seboj povezani s signali, ki vanje vstopajo preko vrat.

Tako opisana zgradba vezja je vidna na sliki 2.5. Točka (a) predstavlja modul oz. primerek entitete F, ki ima vhoda A in B ter izhod Y. Na točki (b)

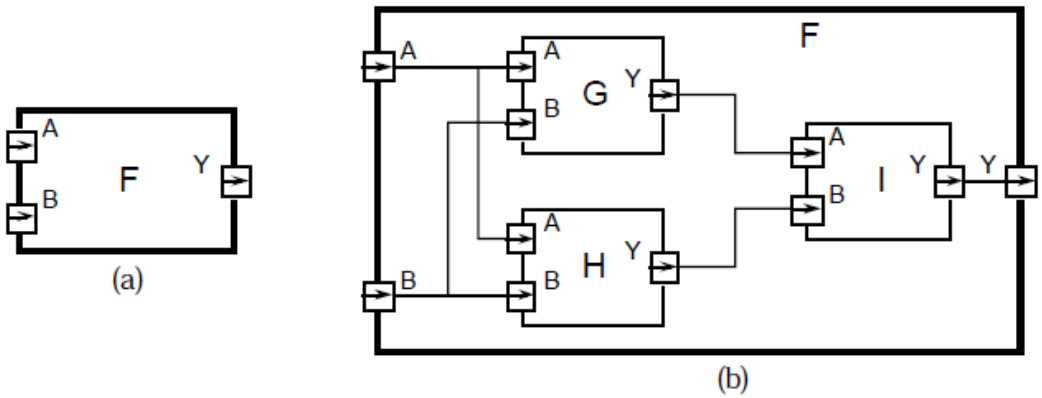


Slika 2.4: Čip FPGA XC6VLX240T na razvojni plošči ML605 s povezavami do komponent na plošči. Vir [7].

pa vidimo, da je modul F sestavljen iz treh modulov oz. primerkov entitet G, H in I. Takemu opisu vezij pravimo *strukturalni opis*. Na ta način z VHDL opisujemo hierarhijo vezja, vse dokler ne pridemo do manjših enot, ki jih ni več smiselno tako opisovati. Na tem mestu moramo opisati tudi notranje delovanje modula oz. kako vhodni signali določajo izhodne. Takemu opisu rečemo *funkcionalni* opis ali opis *obnašanja* (angl. *behavioral description*). Delovanje modula G na sliki 2.5 bi lahko opisali kot logično funkcijo 2.1.

$$Y = A \vee B \tag{2.1}$$

V tem primeru je izhodni signal Y funkcija vhodov A in B, če nad njima



Slika 2.5: Vezje je v VHDL opisano z moduli, ki so sestavljeni iz podmodulov, med seboj pa so povezani s signali. Vir [6].

uporabimo logično operacijo *ali*.

Za opisovanje in simuliranje delovanja vezij je potrebno vključiti še časovno komponento, saj vezja vsebujejo pomnilne elemente. To v VHDL izvedeno z vpeljavo urinega signala v module. Ura je signal, ki izmenično spreminja svoje stanje. VHDL definira posebno strukturo, imenovano *proces* (angl. *process*). Stanja vseh signalov, ki so znotraj procesa se bodo postavile na novo vrednost vsakič, ko se bo spremenil signal iz *občutljivostnega seznama* (angl. *sensitivity list*) procesa. Na ta način je mogoče opisovati sekvenčna vezja.

VHDL kot jezik vsebuje še mnogo struktur in načinov za opisovanje vezij, vendar je za grobo razumevanje tak opis zadosten. Načrtovanje vezja z VHDL lahko opravimo v razvojnem okolju *Xilinx ISE*, ki je namenjeno razvoju za FPGA. To orodje nam glede na opis delovanja vezja proizvede RTL (angl. *Register-Transfer Level*) shemo. Vmesni koraki vključujejo optimizacijo in postavitev elementov na čip. Delovanje vezja lahko preverimo s programsko opremo, ki opravlja simulacijo - v paket ISE je že vključen simulator *ISim*. Zadnji korak pred preizkušanjem vezja na dejanskem FPGA čipu, je še prevajanje modela, ki je izdelan z VHDL v logična vrata in žice, ki so postavljene na sam FPGA čip [8].

Poglavje 3

Pomnilnik DDR SDRAM

Glavni pomnilnik v računalniških sistemih je dandanes izdelan v tehnologiji DDR SDRAM (angl. *Double Data Rate Synchronous Dynamic Random-Access Memory*). Gre za kombinacijo različnih tehnologij. Kratica *RAM* predstavlja *pomnilnik z naključnim dostopom*. Izraz naključni dostop pomeni, da je čas za dostop do pomnilniške besede neodvisen od pred tem naslovljenih besed [10]. Pomnilnik RAM si lahko predstavljamo kot mrežo vrstic in stolpcev, kjer so posamezne pomnilniške besede na presečiščih le-teh. Za dostop do podatkov je potrebno pomnilniku dostaviti želen naslov vrstice in stolpca podatka.

Slika 3.1: Tipičen DDR3 SDRAM modul. Vir [11].

Predpona *D* pri izrazu DRAM predstavlja *dinamični* pomnilnik. Obsta-

jata dve izvedbi pomnilnika RAM in sicer dinamični ter statični. Dinamični pomnilniki so mnogo večje velikosti in so tudi počasnejši kot statični, so pa zato neprimerno cenejši. Razlog za tako razliko v zmogljivosti in ceni je v zgradbi pomnilniške celice. DRAM pomnilnik potrebuje za izvedbo pomnilniške celice le en tranzistor, SRAM pa ponavadi 6 [10]. Še ena pomembna razlika med DRAM in SRAM je osveževanje - DRAM pomnilniške celice je potrebno periodično osveževati, saj bi se v nasprotnem primeru njihova vsebina izbrisala. Osveževalni cikli so v bistvu preprosti bralni dostopi, saj se ob njih prebrana vsebina ponovno zapiše v pomnilnik.

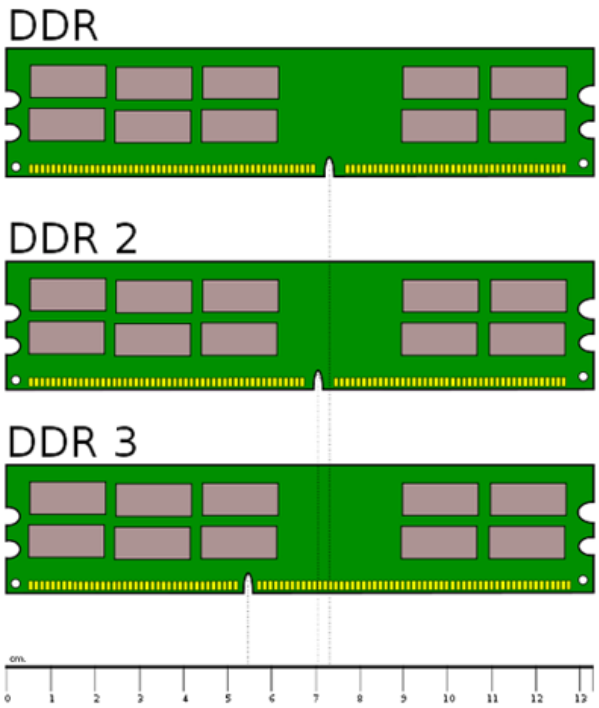
Predpona *S* pri izrazu SDRAM predstavlja *sinhrono* delovanje pomnilnika. Posledica tega je uvedba urinega signala, glavna sprememba pa je uporaba cevovodnega delovanja. Prednost sinhronskih RAM-ov pred asinhronskimi je v tem, da lahko izdajamo ukaze za dostop pomnilniku še preden so bili prejšnji izvedeni do konca [10].

Kratica DDR pred imenom SDRAM pomeni, da gre za pomnilnik z *dvojno hitrostjo*. Pri teh pomnilnikih se podatki prenašajo z dvakratno frekvenco ure. To bo mogoče opaziti v nadaljevanju pri podrobnejšem opisu implementacije krmilnika DDR SDRAM. Prenos podatkov z dvojno hitrostjo je implementiran tako, da se podatki prenašajo tako ob pozitivni kot tudi negativni urini fronti in ne le ob pozitivni. Ostaja več izvedb DDR tehnologije - DDR, DDR2 in DDR3. Posamezni primerki modulov so prikazani na sliki 3.2. Gre za novejše standarde pomnilnikov, ki se razlikujejo po vedno višjih frekvencah delovanja, nižji porabi energije ter še nekaterih bolj specifičnih tehnoloških napredkih, ki omogočajo vedno višje prenosne hitrosti.

Module RAM-a najdemo v več različnih pakiranjih. V osebnih računalnikih najdemo DIMM (angl. Dual In-Line Memory Module) DDR module. Gre za tiskano vezje, na katerem se nahaja več RAM čipov. Ti moduli imajo 64-bitno podatkovno pot, obstajajo pa izvedbe z različnim številom priključnih nožic. Pri prenosnih računalnikih so uveljavljeni SODIMM (angl. *Small Outline Dual In-line Memory Module*) moduli, saj se nahajajo na manjši ploščici ter vsebujejo manj priključkov. DDR3 DIMM moduli imajo 240 nožic, SODIMM moduli pa le 204 [13].

3.1 DDR3 SDRAM SODIMM na razvojni plošči ML605

Podrobneje bo predstavljen RAM modul, ki se nahaja na zmogljivejši razvojni plošči z Virtex-6 FPGA-jem - ML605. Gre za 512MB DDR3 SDRAM SO-



Slika 3.2: Primerjava različnih DDR SDRAM modulov. Vir [12].

DIMM proizvajalca *Micron Semiconductor Products* s točno oznako *MT4JSF6464HY-1G1*. Pomen delov celotne oznake je prikazan v tabeli 3.1. Prvi del označuje, da gre za 512MB DDR3 SDRAM SODIMM modul. Pripona *Y* označuje tip pakiranja, pripona *-1G1* pa označuje frekvenco delovanja. Najvišja frekvenca delovanja je 533 MHz pri CL = 7. *CL* označuje CAS Latency (angl. *Column Address Strobe*). To je zamik med trenutkom, ko zahtevamo dostop do stolpca podatkov in trenutkom, ko so podatki na voljo na priključkih [10]. Oznaka CL = 7 pomeni, da je ta čas dolg 7 urinih period. Pri tej frekvenci delovanja RAM zmore 1066 MT/s, kar pomeni 1066 milijonov prenosov na sekundo. Iz tega parametra izhaja tudi način za industrijsko označevanje hitrosti pomnilnika, v tem primeru je oznaka za RAM *PC3-10600* [14]. Ta RAM lahko deluje tudi pri frekvenci 400 MHz, v tem primeru je CL = 6. Pri RAM-u se podaja tudi parametre kot sta $t_{RCD} = 7$ (angl. *RAS to CAS delay*), $t_{RP} = 7$ (angl. *Row Precharge*). Parametri so podani v urinih periodah in veljajo za hitrost 533 MHz. Čas t_{RCD} označuje število urinih period med RAS (angl. *Row Address Strobe*) in CAS. Gre za razliko med časoma, aktiviracije RAS in CAS signalov. t_{RP} označuje čas ,ki je potreben za zapiranje

odprte vrstice pomnilnika, tako da bo mogoče dostopati do nove vrstice.

Del oznake	**Označuje**	**Pomen**
MT4JSF6464H	Model	512MB DDR3 SDRAM SODIMM
Y	Pakiranje	204-pin DIMM
-1G1	Frekvenca	DDR3-1066

Tabela 3.1: Pomen oznak DDR3 RAM-a na razvojni plošči ML605.

DDR3 SDRAM moduli potrebujejo diferencialni urin signal. To pomeni, da sta za uro potrebna dva signala, kjer je drugi negiran prvi signal in obratno. Ta pomnilniški modul vsebuje tudi vgrajen temperaturni senzor. Podatke o temperaturi je mogoče odčitavati z I^2C protokolom. Temperaturnega senzorja v tej nalogi ne bomo uporabljali.

3.1.1 Organizacija pomnilnika

Modul je velikosti 512 MB in ima organizacijo 64 M x 64. 64 M predstavlja število pomnilniških besed, x 64 pa označuje *pomnilniško širino* oz. število sosednjih bitov, do katerih se dostopa naenkrat. Zmnožek obeh vrednosti nam da vrednost 512 MB. Modul na površini tiskanega vezja je sestavljen iz 4 čipov velikosti 1 Gb (128 MB), ki skupaj tvorijo kapaciteto 512 MB. Vsak čip ima organizacijo 64 M x 16. Skupaj torej tvorijo organizacijo 64 M x 64, saj je uporabljeno *pomnilniško prepletanje* (angl. *memory interleaving*). Vsak čip imenujemo *banka* ali *modul*, v tem primeru gre za 4-kratno prepletanje [10]. S pomnilniškim prepletanjem je dosežena večja hitrost prenosa podatkov, saj do vseh bank lahko dostopamo hkrati, zato v bistvu hkrati dostopamo do 64 bitov podatkov in ne le do 16 bitov.

Vsak izmed štirih modulov po 1 Gb vsebuje 8 K (8192) vrstic in 1 K (1024) stolpcev. To je ena banka modula 1 Gb - vsak tak modul pa vsebuje 8 bank te velikosti. Iz vseh teh lastnosti lahko preverimo velikost 512 MB pomnilniškega SODIMM modula. Enačba 3.1 prikazuje vse potrebne parametre, ki jih je potrebno zmnožiti, enačba 3.2 pa nam da pravilen rezultat - 512 MB.

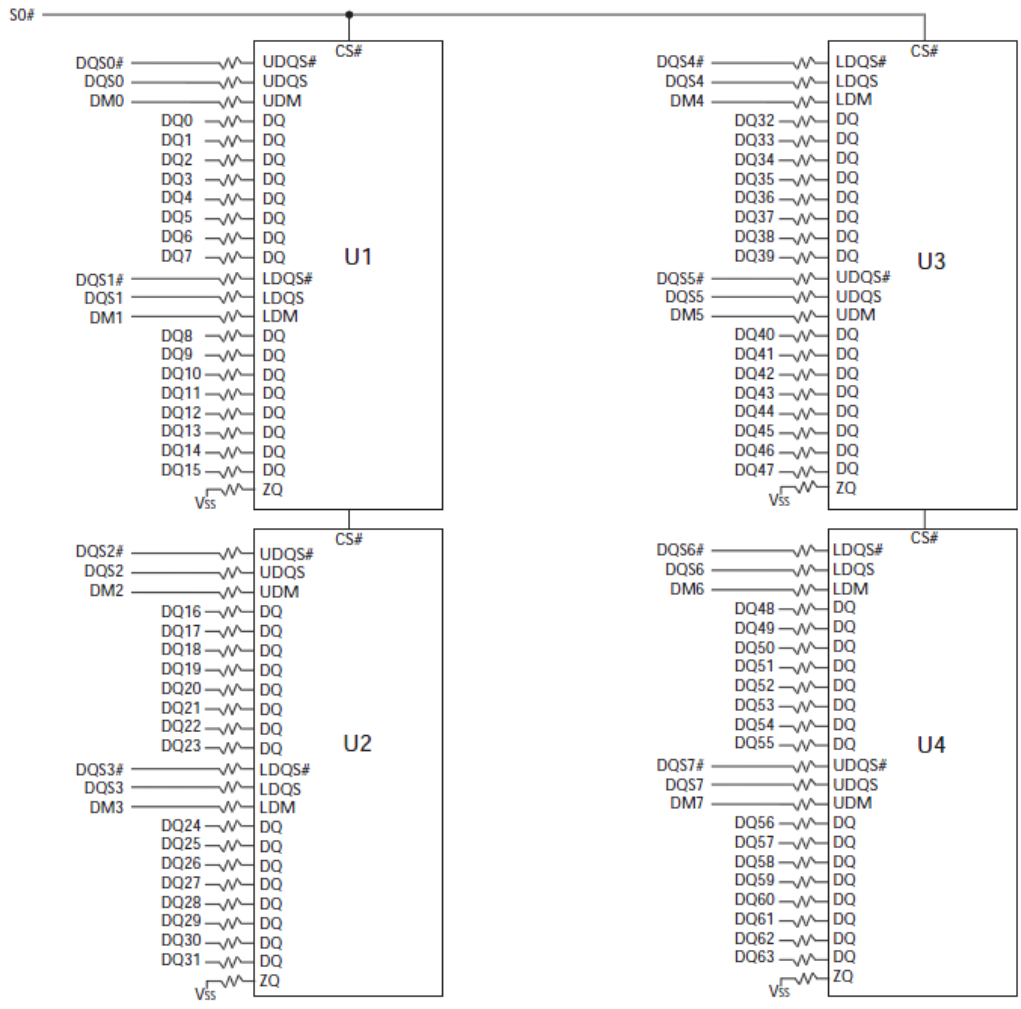
$$st\_vrstic * st\_stolpcev * st\_bank * pomn\_sirina * st\_cipov = velikost[b] \quad (3.1)$$

$$8192 * 1024 * 8 * 16 * 4 = 4.294.967.296b = 512MB \quad (3.2)$$

3.1.2 Opis priključkov

Gre za SODIMM modul, zato je število priključkov v primerjavi z DIMM moduli zmanjšano iz 240 na 204. V tabeli 3.2 so opisani vsi priključki pomnilnika. Nekateri izmed njih obsegajo več linij, kar je označeno z vrednostmi v oglatih oklepajih. Signali, ki vsebujejo znak #, so aktivni ob nizkem logičnem stanju.

Bločna shema modulov pomnilnika je prikazana na sliki 3.3. Vidimo, kako so podatkovni in nekateri kontrolni signali porazdeljeni po vseh štirih 1 Gb modulih, saj gre za pomnilniško prepletanje.



Slika 3.3: Bločni diagram 512 MB DDR3 SDRAM SODIMM pomnilnika z nekaterimi priključki. Vir [14].

3.1.3 Dostop do podatkov

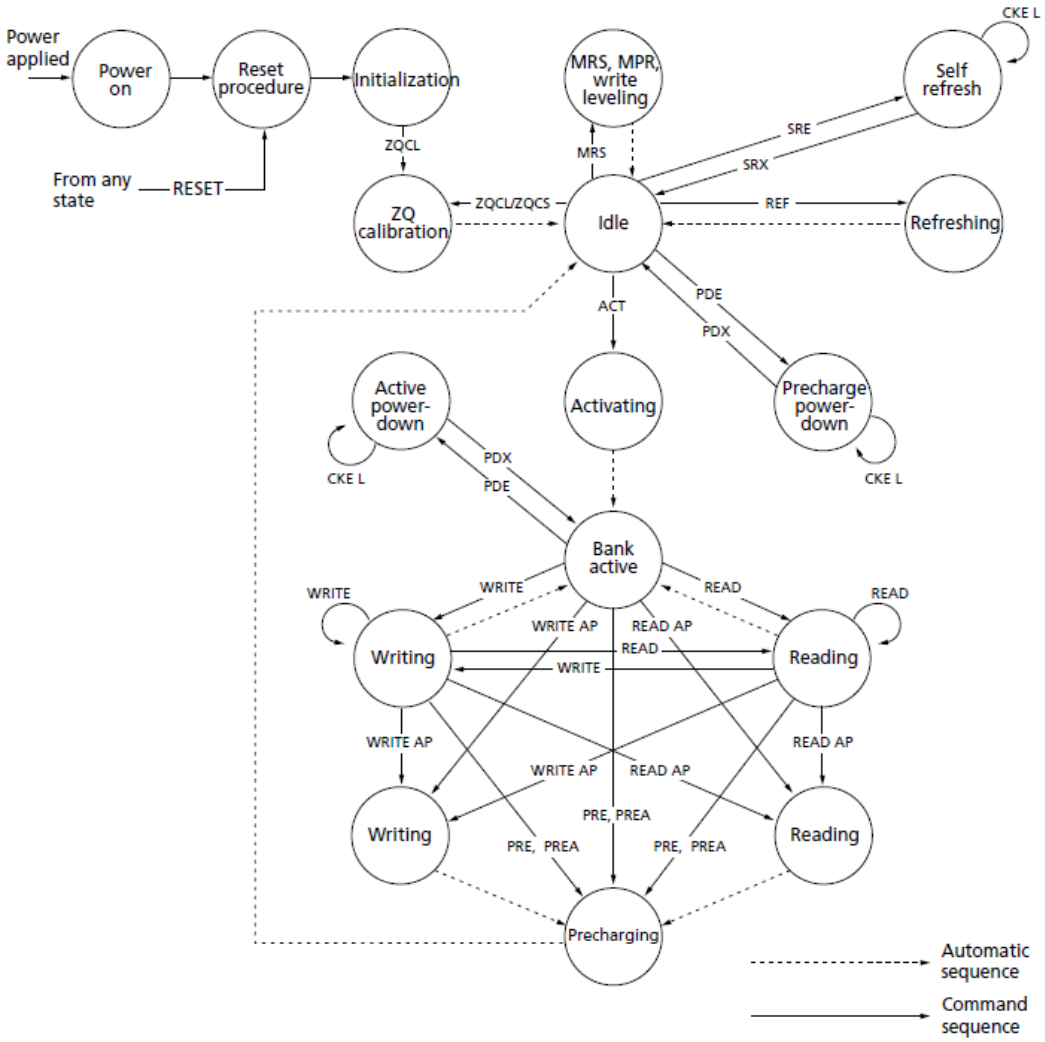
Ena izmed glavnih lastnosti DDR pomnilniške arhitekture je t.i. *prefetch* arhitektura. Gre za to, da je en bralni ali pisalni dostop do DDR3 SDRAM modula v bistvu širok 8n bitov [14]. To določa parameter *dolžina eksplozijskega cikla* (angl. *burst length*), ki je za ta RAM modul enak 8. Vsakič, ko bomo izvedli en ukaz, se bo torej dejansko izvedlo 8 dostopov do pomnilnika. Takšni dostopi do pomnilnika so hitri, saj se dostopa do pomnilniških besed v eni vrstici, celotna vrstica pa se ob dostopu v vsakem primeru prebere iz pomnilnika [10]. Takšnemu načinu dostopa do pomnilnika rečemo tudi *dostop do strani* (angl. *page mode*). Ob vsakem pisanju se v RAM zapiše 64 bitov podatkov, ker pa gre za 8 takih dostopov, se v bistvu z enim pisalnim ukazom zapisuje 512 bitov. Enak princip velja za bralne ukaze. Po potrebi lahko dostopamo tudi do manj podatkov v RAM-u, in sicer z uporabo podatkovnih mask. Uporaba eksplozijskih prenosov je eden izmed glavnih razlogov za veliko prenosno hitrost DDR pomnilnikov.

V nadaljevanju bo opisano delovanje enega 1 Gb modula, zato se poimenovanje signalov tega modula malenkostno razlikuje od poimenovanja signalov celotnega 512 MB modula. 1 Gb modul vsebuje 8 bank organizacije 16 M x 16, kar nam da organizacijo modula 64 M x 16.

Pomnilnik se lahko nahaja v različnih stanjih delovanja. Slika 3.4 prikazuje vsa možna stanja, v katerih se lahko opisan modul RAM nahaja. Črtkane puščice prikazujejo delovanje, ki ga RAM izvaja samodejno, polne puščice pa prikazujejo ukaze, ki jih izvaja uporabnik. Opazimo, da je potrebno RAM pred začetkom uporabe resetirati ter izvesti inicializacijo. Inicializacija je točno določen postopek, kjer je treba v pravilnem zaporedju postavljati določene signale in jih v stanjih držati predpisan čas. Vidimo, da je potrebno skrbeti tudi za osveževanje pomnilnika. Preden lahko izvajamo pisanja in branja, je potrebno izvesti tudi ukaz *ACTIVATE*.

Pomnilnik vsebuje več različnih ukazov, ukaz pa je izbran na podlagi stanja signalov *CS#*, *RAS#*, *CAS#*, *WE#* ob pozitivni fronti ure. Sledi kratek opis nekaterih najbolj pogosto uporabljenih ukazov:

- *NO OPERATION* - ukaz, imenovan tudi NOP, v bistvu ne izvaja ničesar. S tem je preprečeno izvajanje neželenih ukazov v stanjih kot je čakanje.
- *ZQ CALIBRATION LONG* - začetna kalibracija pomnilnika ob vklopu ali resetiranju.
- *ZQ CALIBRATION SHORT* - periodična kalibracija zaradi variacij temperature in napetosti.



Slika 3.4: Diagram prehajanja stanj RAM-a. Vir [15].

- *MODE REGISTER SET* - s tem ukazom moramo v postopku inicializacije v registre MR0-MR3 vpisati določene parametre in nastavitve za RAM. Tu gre za npr. dolžino in tip eksplozijskega prenosa, CAS latenco, način delovanja RAM-a, avtomatsko osveževanje itd.

- *ACTIVATE* - ukaz, ki odpre oz. aktivira vrstico neke banke, kar je potrebno pred dostopom do nje. Vrstica banke ostane aktivna, dokler ni izdan ukaz *PRECHARGE* za to banko. Pred odpiranjem druge vrstice v isti banki mora biti izdan ukaz *PRECHARGE*.

- *READ* - označuje začetek eksplozijskega bralnega cikla v aktivni vrstici. Pri tem se lahko uporabi *auto precharge*, ki vrstico zapre po koncu dostopa. Stolpec, kjer se bo izvedel prvi dostop, je potrebno postaviti na naslovni signal *A[2:0]*.

- *WRITE* - označuje začetek eksplozijskega pisalnega cikla v aktivni vrstici. Tudi tu je mogoč *auto precharge*.

- *PRECHARGE* - zapiranje oz. deaktivacija odprte vrstice v določeni banki ali v vseh bankah naenkrat. Banke so spet na voljo po času t_{RP}, razen v primeru avtomatskega precharge ukaza. Ko je banka zaprta, mora biti ponovno aktivirana z *ACTIVATE* ukazom, preden se lahko dostopa do podatkov.

- *REFRESH* - ta ukaz je potrebno izvesti vsakič, ko je potrebno osveževanje. *REFRESH* ukaz je v bistvu *CAS#-before- RAS#* (CAS# pred RAS#), saj najprej aktiviramo CAS# signal, nato pa še RAS#. Osveževanje je v bistvu preprosto odpiranje vrstice. V povprečju je potrebno vsakih 7.8 μs.

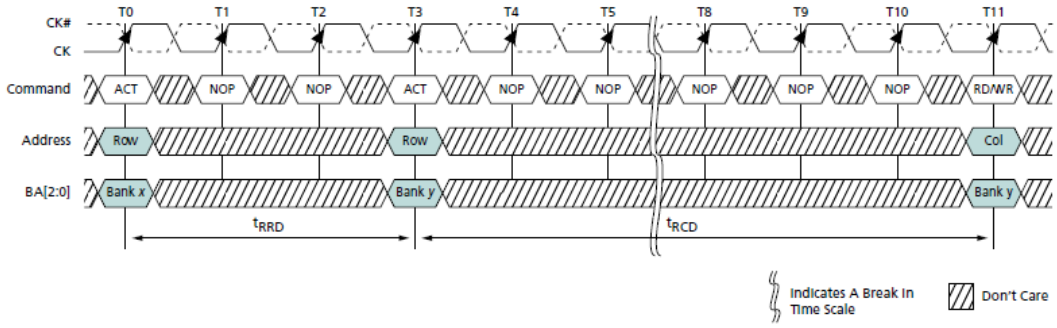
- *SELF REFRESH* - s tem ukazom se ohranja podatke v RAM-u, če je ostali sistem izključen iz napajanja. S tem ukazom RAM ohranja podatke brez eksternega urinega signala.

Ukaz *ACTIVATE*

Preden lahko izvajamo bralne in pisalne ukaze, je potrebno vrstico banke odpreti oz. aktivirati z ukazom *ACTIVATE*. Ko je izdan ukaz *ACTIVATE*, je potrebno čakati čas t_{RCD}, nato pa je mogoča uporaba pisalnih in bralnih ukazov. Z natančnim programiranjem je mogoče bralni ali pisalni ukaz dostaviti še preden bo potekel čas t_{RCD}. Za izvajanje ukaza *ACTIVATE* je potrebno na naslovno vodilo *A* postaviti naslov vrstice, na signal *BA[2:0]* pa naslov ene izmed osmih bank. Slika 3.5 prikazuje pravilen časovni potek signalov pri izvajanju ukaza.

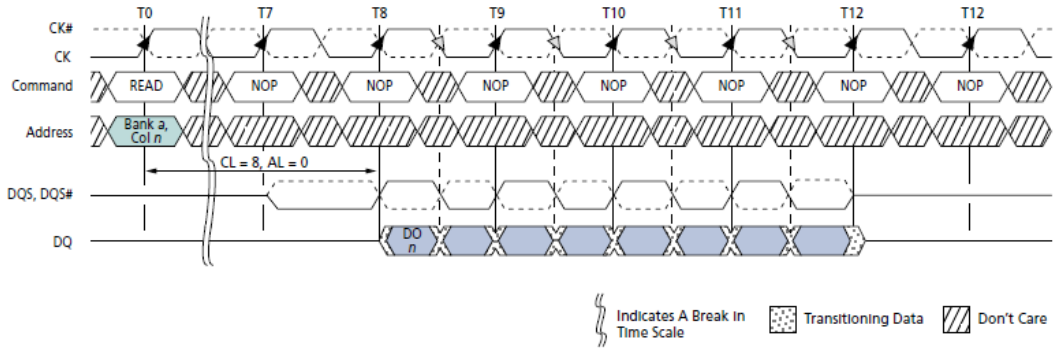
Ukaz *READ*

Bralni eksplozijski cikel se začne z izvedbo ukaza *READ*. Začetni naslov banke in stolpca, kjer se bo začel prvi dostop, je potrebno postaviti na naslovno vodilo. Ob uporabi avtomatskega zapiranja (angl. *auto precharge*) se bo vrstica po koncu eksplozijskega cikla zaprla. Prvi del prebranih podatkov bo



Slika 3.5: Časovni potek signalov pri izvajanju ukaza *ACTIVATE*. Vir [15].

na podatkovnem vodilu *DQ* na voljo čez čas, ki ga označuje *bralna zakasnitev* (angl. *READ latency*). Bralna latenca je vsota aditivne *CAS latence* (*AL*) in *CAS latence* (*CL*). Ti dve vrednosti določimo ob inicializaciji pomnilnika. Signala *DQS* in *DQS#* bosta označevala prihod podatkov na podatkovno vodilo - najprej sledi nizko stanje signala *DQS* in visoko *DQS#*, nato pa bosta ob veljavnosti novih delov podatkov ustrezno alternirala vrednosti, kot je to prikazano na sliki 3.6. Po koncu cikla bosta šla v visoko impedančno stanje - *High-Z*.



Slika 3.6: Časovni potek signalov pri izvajanju ukaza *READ*. Vir [15].

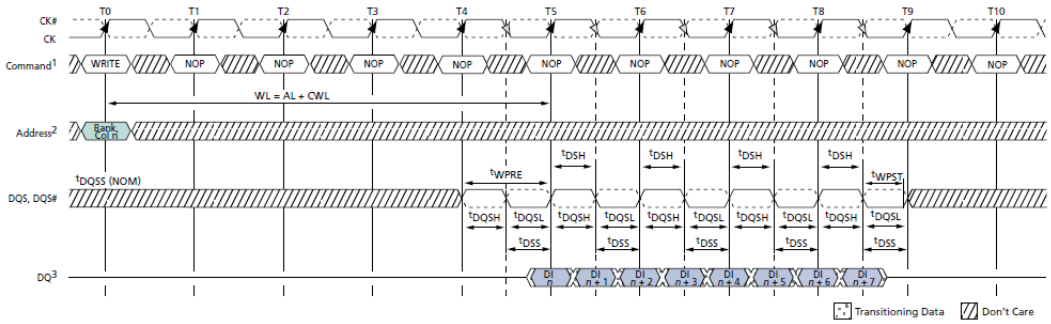
Z izvajanjem zaporednih bralnih ukazov je mogoče zagotoviti neprekinjen tok podatkov, kjer prvi podatek novega eksplozijskega cikla vsebuje tisti podatek, kjer se je prejšnji cikel končal.

S postavitvijo 10. bita naslovnega vodila v visoko stanje ob izstavitvi bralnega ukaza se bo ob koncu njegovega izvajanja izvedlo avtomatično zapiranje vrstice.

Ukaz *WRITE*

Pisalni eksplozijski cikel se začne z izvedbo ukaza *WRITE*. Začetni naslov banke in stolpca, kjer se bo začel prvi dostop, je potrebno postaviti na naslovno vodilo. Ob uporabi avtomatskega zapiranja (angl. *auto precharge*) se bo vrstica po koncu eksplozijskega cikla zaprla.

Primer enega eksplozijskega cikla pri pisanju v RAM je prikazan na sliki 3.7. Prvi podatek na podatkovnem vodilu bo registriran po preteku *pisalne zakasnitve* - WL (angl. *WRITE Latency*), ob pozitivni fronti signala *DQS*. Pisalna zakasnitev je vsota aditivne CAS latence in pisalne CAS latence. Ti dve vrednosti sta določeni ob inicializaciji pomnilnika. Podatke je pri pisanju mogoče maskirati s signalom *DM*, ki mora biti poravnan s podatki. Če bo posamezen bit maskirnega signala v visokem stanju, bo ustrezen podatkovni bit maskiran in se ne bo zapisal v pomnilnik.



Slika 3.7: Časovni potek signalov pri izvajanju ukaza *WRITE*. Vir [15].

Po koncu cikla bo podatkovno vodilo ostalo v visokem impedančnem stanju. Tudi *WRITE* ukaze je mogoče izvajati zaporedno in s tem v RAM zapisati tok podatkov. Pri tem mora naslednji pisalni ukaz slediti čez t_{CCD} (angl. *CAS#-to-CAS#*) časa. Naslednji *READ* ukaz lahko sledi po t_{WTR} (angl. *WRITE to READ*) časa, *PRECHARGE* ukaz pa čez t_{WR} (angl. *WRITE Recovery*) časa. Parametra t_{WTR} in t_{WR} sta odvisna od nastavitev RAM-a pri inicializaciji.

A[12:0]	Vhod	Vhod za naslov
BA[2:0]	Vhod	Naslov banke
CK0, CK0#	Vhod	Vhoda diferencialne ure
CKE0	Vhod	Omogočenje ure
DM[7:0]	Vhod	Podatkovna maska
ODT0	Vhod	Omogočenje terminacijskih uporaov (angl. *On-die termination*)
RAS#, CAS#, WE#	Vhod	Ukazni vhodi
RESET#	Vhod	Resetiranje
S0#	Vhod	Izbira čipa
SA[1:0]	Vhod	Vhod za serijski naslov za temperaturni senzor
SCL	Vhod	Serijska ura za temperaturni senzor
DQ[63:0]	Vhod/izhod	64-bitno vhodno/izhodno podatkovno vodilo
DQS[7:0], DQS#[7:0]	Vhod/izhod	Prožilni pulzi za podatke
SDA	Vhod/izhod	Serijski podatki iz temperaturnega senzorja
EVENT#	Izhod	Temperaturni dogodek
V_{DD}	Napajanje	Električno napajanje 1,5 V +/- 0,0075 V
V_{DDSPD}	Napajanje	Napajanje za temperaturni senzor
V_{REF}CA	Napajanje	Referenčna napetost za naslove, ukaze in kontrolne signale
V_{REF}DQ	Napajanje	Referenčna napetost za prožilne pulze
V_{SS}	Napajanje	Ozemljitev
V_{TT}	Napajanje	Terminacijska napetost

Tabela 3.2: Opis priključkov 512 MB DDR3 SDRAM SODIMM pomnilnika.

Poglavje 4

Načrt izvedbe naloge

Implementacija celotnega vezja, ki bi omogočalo prenos podatkov iz računalnika na razvojno ploščo FPGA, shranjevanje in nato branje iz pomnilnika RAM ter prenos nazaj na računalnik, realiziramo kot povezavo večjega števila modulov, kjer vsak opravlja svojo funkcijo. Kot omenjeno, je taka realizacija tipična za sisteme opisane z jezikom VHDL. Smiselna je razdelitev na naslednje module:

- Krmilnik za pomnilnik RAM
- Uporabniška aplikacija, ki dostopa do pomnilnika
  - UART modul
  - Registri za začasno shranjevanje podatkov

Proizvajalec čipov FPGA in razvojnih plošč, ki jih imamo na voljo, nam nudi orodje *MIG* (angl. *Memory Interface Generator*), s katerim ustvarimo krmilnik za dostop do pomnilnika RAM. Modul, ki vsebuje uporabniško aplikacijo, je sestavljen še iz dveh podmodulov - iz UART modula, katerega implementacija je opisana v poglavju 5.2.4, ter niza registrov, v katere se začasno shranjujejo podatki.

Najprej poskusimo z implementacijo rešitve problema na razvojni plošči Spartan-3A. V primeru neuspeha imamo še vedno na voljo drugo, zmogljivejšo ploščo.

4.1 Poskus implementacije na razvojni plošči Spartan-3A

Z orodjem MIG smo po navodilih iz vodiča [18] ustvarili krmilnik za pomnilnik DDR2 specifično za razvojno ploščo Spartan-3A. Krmilnik povežemo z izdelanim UART modulom ter uporabniško logiko, ki izvaja testna pisanja in branja v RAM. Na razvojno ploščo so bili poslani podatki za zapisovanje, npr. črke *"ABCDEFGH"*. Razvojna plošča sprejete podatke zapiše v pomnilnik RAM ter jih nato prebere in pošlje nazaj na računalnik po serijski povezavi. Izkazalo se je, da so podatki, ki jih plošča pošlje nazaj na računalnik, pogosto okvarjeni. Šlo je za podvojene črke ali povsem napačne sprejete znake. S preizkusom na LCD zaslonu na plošči se je izkazalo, da krmilnik in RAM delujeta. Testiranje le UART je prav tako delovalo. Problem je nastopil, ko smo povezali skupaj vse module. Pregled poročila, ki se ustvari ob sintezi vezja na sliki 4.1 razkrije, da so težave v doseganju *časovnih omejitev* vezja. Časovne omejitve vezja so določene v posebni datoteki z omejitvami formata *UCF* (angl. *User Constraints File*). To datoteko je ustvarilo orodje MIG, potrebni so bili le manjši dodatki, tako da lahko našo napako pri implementaciji izključimo.

vhdl_bl4 Project Status (05/28/2012 - 12:21:16)			
Project File:	test.xise	**Parser Errors:**	No Errors
Module Name:	vhdl_bl4	**Implementation State:**	Programming File Generated
Target Device:	xc3s700a-4fg484	• **Errors:**	No Errors
Product Version:	ISE 13.3	• **Warnings:**	403 Warnings (98 new)
Design Goal:	Balanced	• **Routing Results:**	All Signals Completely Routed
Design Strategy:	Xilinx Default (unlocked)	• **Timing Constraints:**	X 1 Failing Constraint
Environment:	System Settings	• **Final Timing Score:**	109150 (Timing Report)

Slika 4.1: Poročilo po sintezi vezja za Spartan-3A - opazimo problem s časovnimi omejitvami.

Smisel določanja časovnih omejitev je v tem, da z določanjem največjih dovoljenih zakasnitev na signalih in ostalih parametrih zagotovimo pravilno delovanje vezja. Pri zelo preprostih vezjih določanje časovnih omejitev ni potrebno, pomnilnik RAM pa je občutljivo integrirano vezje, ki deluje pri visokih frekvencah. Tu lahko zakasnitve povzročijo napačno delovanje. To je torej razlog za določitev mnogih časovnih omejitev pri določenih signalih.

Bolj podroben pregled poročila pokaže, da omejitvam ne ustreza več signalov. Na sliki 4.2 je prikazano poročilo za enega izmed njih - povezava med

```
================================================================================
Timing constraint: PERIOD analysis for net "infrastructure_top0/clk_dcm0/clk90dcm" derived from  NET "infrastructure_t
 1157 paths analyzed, 803 endpoints analyzed, 122 failing endpoints
 122 timing errors detected. (122 setup errors, 0 hold errors, 0 component switching limit errors)
 Minimum period is  15.608ns.
--------------------------------------------------------------------------------

Paths for end point main_00/test_bench0/Inst_register_file_serial/reg6_1 (SLICE_X30Y56.CE), 3 paths
--------------------------------------------------------------------------------
Slack (setup path):     -2.022ns (requirement - (data path - clock path skew + uncertainty))
  Source:               main_00/test_bench0/start_flag_counter_2 (FF)
  Destination:          main_00/test_bench0/Inst_register_file_serial/reg6_1 (FF)
  Requirement:          1.879ns
  Data Path Delay:      3.532ns (Levels of Logic = 1)(Component delays alone exceeds constraint)
  Clock Path Skew:      -0.369ns (1.489 - 1.858)
  Source Clock:         clk_0 rising at 0.000ns
  Destination Clock:    clk90_0 rising at 1.879ns
  Clock Uncertainty:    0.000ns

  Maximum Data Path: main_00/test_bench0/start_flag_counter_2 to main_00/test_bench0/Inst_register_file_serial/reg6_1
    Location             Delay type         Delay(ns)  Physical Resource
                                                       Logical Resource(s)
    -------------------------------------------------  -------------------
    SLICE_X29Y53.XQ      Tcko                  0.591   main_00/test_bench0/start_flag_counter<2>
                                                       main_00/test_bench0/start_flag_counter_2
    SLICE_X28Y52.G1      net (fanout=17)       0.689   main_00/test_bench0/start_flag_counter<2>
    SLICE_X28Y52.Y       Tilo                  0.707   main_00/test_bench0/Inst_register_file_serial/reg7_not0001
                                                       main_00/test_bench0/Inst_register_file_serial/reg6_not00011
    SLICE_X30Y56.CE      net (fanout=4)        1.234   main_00/test_bench0/Inst_register_file_serial/reg6_not0001
    SLICE_X30Y56.CLK     Tceck                 0.311   main_00/test_bench0/serial_data64<9>
                                                       main_00/test_bench0/Inst_register_file_serial/reg6_1
    -------------------------------------------------  ---------------------------
    Total                                      3.532ns (1.609ns logic, 1.923ns route)
                                                       (45.6% logic, 54.4% route)
```

Slika 4.2: Problematičen signal, ki ne ustreza časovnim omejitvam.

dvema flip-flopoma, ki povezuje števec in register. Problematičen je *čas za vzpostavitev* (angl. *setup time*) - čas, ko mora biti signal v mirovanju pred pozitivno urino fronto, da bo pravilno vzorčen [19]. Iz slike 4.2 vidimo, da ta signal ne zadošča časovni omejitvi največ *1,879 ns*, ampak ta največji dovoljeni čas prekorači, saj njegov čas mirovanja pred fronto znaša *3,532 ns*. Poročilo nam pove tudi, da je 45,6 % take zakasnitve rezultat vmesne logike na čipu, 54,4 % pa jo povzroči dolžina same povezave na čipu.

Poročilo sinteze torej razkrije, da so vzrok za napačno delovanje rešitve na razvojni plošči FPGA Spartan-3A preveč zakasnjene povezave med logiko, kar povzroči napačno in nepredvidljivo delovanje vezja. To hipotezo potrdi testiranje RAM krmilnika in UART-a posamezno, saj tako delujeta brez problemov. Prav tako takrat poročilo ne javi nobenih problemov s časovnimi zakasnitvami. Pri simulaciji delovanja vezja se je vezje obnašalo po pričakovanjih, torej brez problemov. Razlog za to je, da simulator ne simulira časovnih zakasnitev, ki so v tem primeru nastale.

Razlogi za tako veliko časovno zakasnitev so najverjetneje v konfliktu med postavitvijo UART modula ter krmilnika za RAM. Čip FPGA, ki ga plošča

Spartan-3A uporablja, je relativno "majhen", oz. vsebuje majhno število logičnih celic. Neuspešno povezovanje teh modulov je najverjetneje vzrok za neoptimalne povezave, ki pri tem nastanejo.

Vezje ni delovalo že pri prvih preizkusih, celotne rešitve pa sploh še nismo v celoti implementirali. Dodati bi bilo potrebno še kar nekaj logike, prav tako pa bi celotno delo diplomske naloge postalo praktično uporabno šele, ko bi dodali še logiko za procesiranje podatkov. To bi povzročilo še večje časovne zakasnitve, tako da celotna rešitev zanesljivo ne bi delovala. Zaradi tega razloga je bila sprejeta odločitev, da se rešitev implementira na mnogo bolj zmogljivi razvojni plošči Virtex-6 ML605, kjer podobnih problemov nismo pričakovali.

Poglavje 5

Implementacija na razvojni plošči Virtex-6 ML605

Podrobnejša implementacija rešitve bo opisana v nadaljevanju, saj se nedelujoča implementacija za razvojno ploščo Spartan-3A, ki je opisana v poglavju 4.1, na nekaterih mestih razlikuje od implementacije za Virtex-6 ML605.

5.1 Implementacija krmilnika za RAM

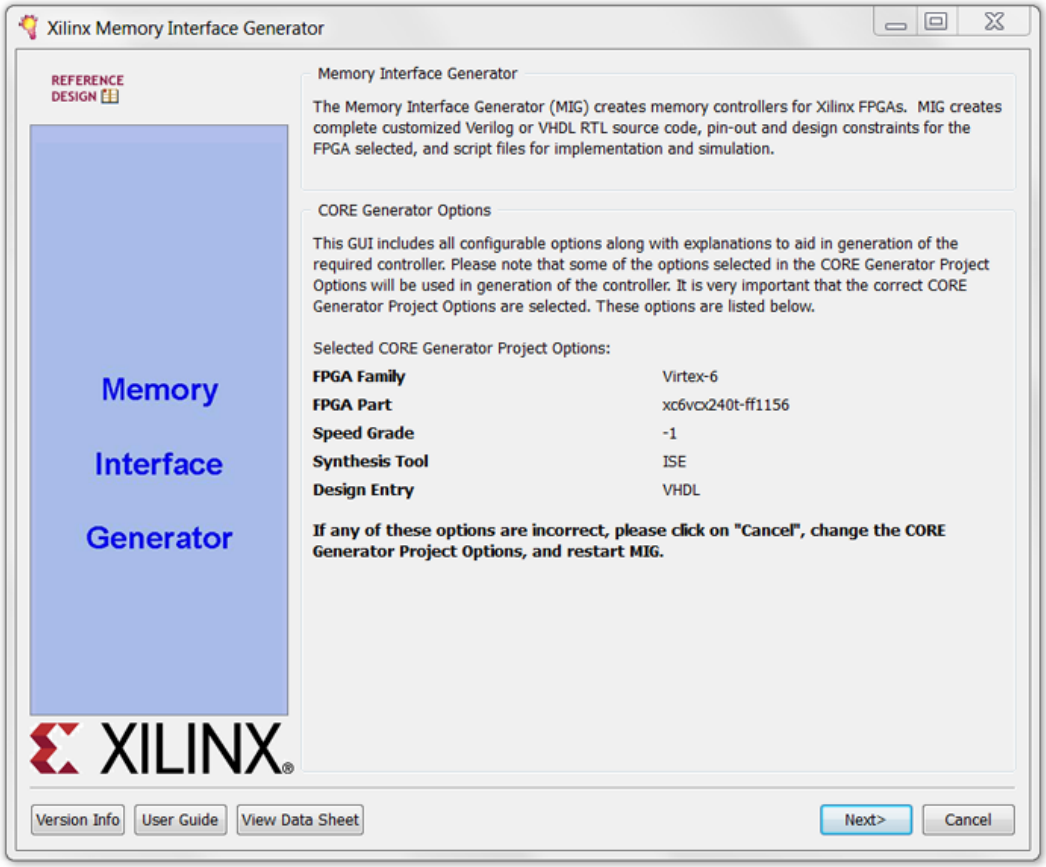
5.1.1 Uporaba orodja MIG

Na voljo nam je orodje MIG, s katerim ustvarimo krmilnik za pomnilnik. Gre za *čarovnika* (angl. *wizard*) z grafičnim vmesnikom, ki nas vodi skozi korake, s katerimi določamo vse potrebne parametre. Krmilnik je optimiziran za doseganje čim večjih hitrosti ter vsebuje tudi opcijski razvrščevalnik ukazov. Razvrščevalnik preureja ukaze tako, da je dostopni čas do pomnilnika krajši. Za naše potrebe ga ne potrebujemo, saj ukazov ne bomo izdajali zelo hitro, z njegovo odsotnostjo pa ne bomo po nepotrebnem zakomplicirali vezja. Smisel orodja MIG je nudenje čimbolj preprostega vmesnika za delo s pomnilnikom z uporabo čim manj signalov.

Ob zagonu orodja MIG se srečamo z zaslonom na sliki 5.3. Čarovnik nas nato vodi skozi naslednje pomembne korake:

1. Izbira čipa in programskega jezika
2. Izbira frekvence delovanja pomnilnika in točnega čipa RAM ter podatkovne širine (angl. *data width*)

3. Izbira *dolžine eksplozijskega cikla* (angl. *burst length*)
4. Nastavitev ure za FPGA
5. Izbira nožic na čipu FPGA (za podatke, uro in kontrolne signale)



Slika 5.1: Prvi korak pri ustvarjanju krmilnika za RAM je potrditev nastavitev FPGA čipa.

Za razvojno ploščo ML605, ki smo jo uporabili, je potrebno izbrati pravilne nastavitve - predvsem frekvenco ure, pravilen RAM čip in mesta za postavitev logike na čipu. Izbira pravilnih parametrov je opisana v dokumentu [21]. Slike 5.2, 5.3 in 5.4 prikazujejo izbrane parametre.

Na sliki 5.2 vidimo, da smo izbrali točen čip FPGA, ki je na plošči ML605. Izbrali smo tudi jezik, v katerem se ustvari krmilnik, in sicer VHDL. Na tem

mestu bi lahko izbrali tudi jezik Verilog. Izbrali smo tudi možnost, da nam MIG ustvari še testno okolje za preizkus delovanja. Določili smo tudi tip ure - gre za diferencialno uro, ki jo plošča ML605 že vsebuje.

Slika 5.3 prikazuje izbrane nastavitve RAM-a. Izbrana je frekvenca delovanja 400 MHz - gre za dvakratno frekvenco ure za FPGA (200 MHz). Kot omenjeno, pomnilnik DDR prenaša podatke ob obeh urinih frontah, zato potrebuje uro z dvakratno frekvenco. Tu lahko opazimo, da pomnilnik ne bo deloval pri svoji maksimalni frekvenci, ki znaša 533 MHz. Razlog za to je v tem, da višje frekvence delovanja RAM-a s čipom FPGA, ki ga razvojna plošča ML605 vsebuje, niso mogoče. To je torej edina mogoča izbira. V čarovniku smo nato izbrali točno določen pomnilniški čip SODIMM modula ter ustrezno podatkovno širino 64. Omogočili smo tudi uporabo podatkovne maske, če se bo kdaj pokazala potreba po prenosu manjše količine podatkov. S parametrom *ORDERING* in izbiro *Strict*, smo onemogočili razvrščanje ukazov RAM-u. Ostale parametre smo na tem koraku pustili na privzetih vrednostih. Mednje spada izbira dolžine eksplozijskega cikla, ki znaša 8 ter izbira CAS latence, ki znaša 6.

```
CORE Generator Options:
   Target Device                  : xc6vlx240t-ff1156
   Speed Grade                    : -1
   HDL                            : vhdl
   Synthesis Tool                 : Foundation_ISE

MIG Output Options:
   Module Name                    : mig_39
   No of Controllers              : 1
   Selected Compatible Device(s)  : --
   Hardware Test Bench            : enabled

FPGA Options:
   Clock Type                     : Differential
   Debug Port                     : ON
   Internal Vref                  : disabled

Extended FPGA Options:
   DCI for DQ, DQS/DQS#           : enabled
   DCI for Address/Control        : disabled
```

Slika 5.2: Prikazani so splošni parametri pri orodju MIG - izbira FPGA čipa, podatki o uri ter še nekatere druge nastavitve.

```
Controller Options :
   Memory                  : DDR3_SDRAM
   Interface               : NATIVE
   Design Clock Frequency  : 2500 ps (400.00 MHz)
   Memory Type             : SODIMMs
   Memory Part             : MT4JSF6464HY-1G1
   Equivalent Part(s)      : --
   Data Width              : 64
   ECC                     : Disabled
   Data Mask               : enabled
   ORDERING                : Strict

Memory Options:
   Burst Length (MR0[1:0])          : 8 - Fixed
   Read Burst Type (MR0[3])         : Sequential
   CAS Latency (MR0[6:4])           : 6
   Output Drive Strength (MR1[5,1]) : RZQ/7
   Rtt_NOM - ODT (MR1[9,6,2])       : RZQ/4
   Rtt_WR - Dynamic ODT (MR2[10:9]) : Dynamic ODT off
```

Slika 5.3: Prikazani so parametri izbranega čipa RAM, ki se nahaja na razvojni plošči ter njegove lastnosti.

Korak, kjer je bilo potrebno izbrati ustrezne nožice za vhodno/izhodne signale na FPGA-ju, je prikazan na sliki 5.4. Gre za specifično postavitev vhodno/izhodnih povezav krmilnika na čipu FPGA v določene banke FPGA. To je potrebno zaradi tega, ker bi poljubna postavitev povezav lahko onemogočila delovanje nekaterih komponent čipa, ali pa bi bila postavitev neoptimalna za zanesljivo delovanje krmilnika in pomnilnika. Možna napaka bi bila, da bi namesto banke na levi strani čipa izbrali tiste na desni. Razlog za izbiro leve strani je ta, da je to najbližja pot signalov do RAM modula. Opazimo, da največ povezav potrebujejo podatkovne in kontrolne oziroma naslovne poti. Skupaj krmilnik zaseda 142 vhodno-izhodnih povezav čipa FPGA.

Ob vseh vnesenih parametrih MIG ustvari dve mapi - v prvi se nahaja programska koda VHDL za krmilnik in testni projekt, ki izvaja branja in pisanja v pomnilnik. V drugi mapi je programska koda brez testnega projekta. Obe mapi vsebujeta datoteko *create_ise.bat*, ki ustvari projekt za ISE razvojno orodje. Tako pripravljen projekt je že skoraj pripravljen za uporabo, potrebnih je le še nekaj popravkov.

Krmilnik, ki nam ga pripravi MIG, je namenjen splošnemu čipu FPGA, ki smo ga izbrali. Čipi se nahajajo na različnih razvojnih ploščah, zato so poleg

```
Selected Banks and Pins usage :
    Data            :bank 25(40) -> Number of pins used : 35
                     bank 26(40) -> Number of pins used : 35
                     bank 35(40) -> Number of pins used : 24

    Address/Control:bank 36(40) -> Number of pins used : 25

    System Clock   :bank 34(40) -> Number of pins used : 9

    Master Banks   :bank 25(40)
    VRN/VRP        :bank 25(40) -> Number of pins used : 2
                    bank 35(40) -> Number of pins used : 2

    VREF           :bank 25(40) -> Number of pins used : 2
                    bank 26(40) -> Number of pins used : 2
                    bank 35(40) -> Number of pins used : 2
                    bank 36(40) -> Number of pins used : 1

    BUFR           :bank 26(40) -> Number of pins used : 1
                    bank 36(40) -> Number of pins used : 1

    BUFIO          :bank 25(40) -> Number of pins used : 3
                    bank 26(40) -> Number of pins used : 3
                    bank 35(40) -> Number of pins used : 2

    Total IOs used :    142
```

Slika 5.4: Specifična postavitev povezav krmilnika na FPGA čip z orodjem MIG

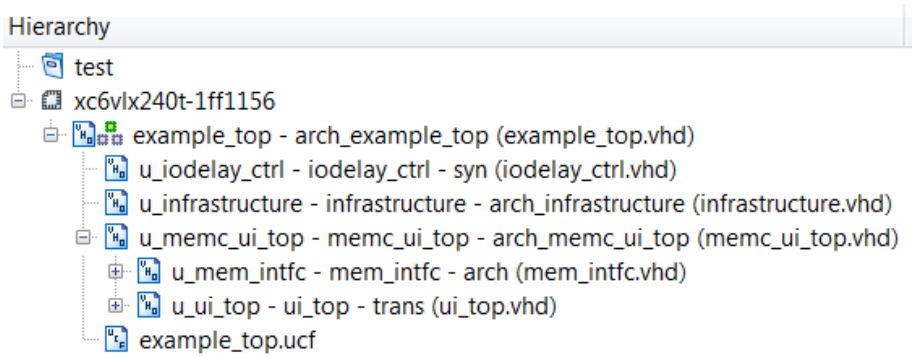
postavitve na čip, ki je že bila narejena posebej za uporabljen čip, potrebni nekateri popravki in dodatki programske kode [21]. Med popravke spadajo med drugim:

- Sprememba nastavitve za uro - 200 MHz diferencialna ura
- Odstranitev nepotrebnih signalov *scl* in *sda*, ki bi služili za odčitavanje temperature pomnilnika
- Sprememba nekaterih parametrov, npr. signal za resetiranje je aktiven, ko je visok
- Sprememba .ucf datoteke z omejitvami

Dokument [21] nam poda povezavo do programske kode krmilnika, ki že vsebuje vse popravke za ploščo ML605, vendar je ta v programskem jeziku Verilog. Ker smo želeli celotno rešitev izdelati v jeziku VHDL, je bilo potrebno VHDL kodo, ki jo MIG ustvari, ročno popraviti. Vsi popravki v dokumentu [21] niso v podrobnosti opisani, zato je bilo treba primerjati nespremenjeno in popravljeno kodo v jeziku Verilog, šele nato pa smo opažene spremembe prenesli na kodo VHDL.

5.1.2 Hierarhija pomnilniškega krmilnika

Hierarhijo posameznih VHDL modulov v projektu, ki smo ga ustvarili z MIG-om in izvedli še potrebne popravke za ploščo ML605, prikazuje slika 5.5. Poleg samega uporabniškega vmesnika za dostop do RAM-a, ki se imenuje *memc_ui_top*, ustvari še 2 modula, ki upravljata z urinim signalom. Preden se poglobimo v delovanje uporabniškega vmesnika za RAM, si oglejmo delovanje teh dveh modulov.



Slika 5.5: Hierarhija projekta s krmilnikom RAM-a, ustvarjena z MIG-om.

Prvi modul je *iodelay_ctrl*, drugi pa *infrastructure*. V modul *iodelay_ctrl* vstopita signala *clk_ref_p* in *clk_ref_n*, ki predstavljata diferencialno 200 MHz uro, ki se nahaja na razvojni plošči. Ta modul je instanca primitiva *IDELAYCTRL*, ki ga nudi Xilinx ISE. Njegova naloga je, da neprestano kalibrira *IODELAY* elemente in s tem izničuje okoljske vplive ne delovanje vezja. *IODELAY* elementi so definirani v *.ucf* datoteki, postavil pa jih je MIG. Izhod iz modula je nov signal *clk_200*.

Signal *clk_200* je nato vhod v drugi modul - *infrastructure*. Njegova naloga je generacija in distribucija ure in sinhronizacija signala za resetiranje. Prav

tako je vhod tudi signal za resetiranje, *sys_rst*. Ta modul ustvari dve uri, ki se nato uporabljata v celotnem dizajnu:

- *clk* - ura za večino interne logike (200 MHz)
- *clk_mem* - ura za DDR3 pomnilnik (400 MHz)

Logika v tem modulu skrbi tudi za pravilno izvedbo resetiranja ob pritisku na tipko, ki ga sproži. Poskrbi za to, da se najprej resetira krmilnik pomnilnika in *IDELAYCTRL* modul, nato pa še ostala logika. Pri tem uporablja *fazno zaklenjeno zanko*, t.i. PLL (angl. *Phase-Locked Loop*).

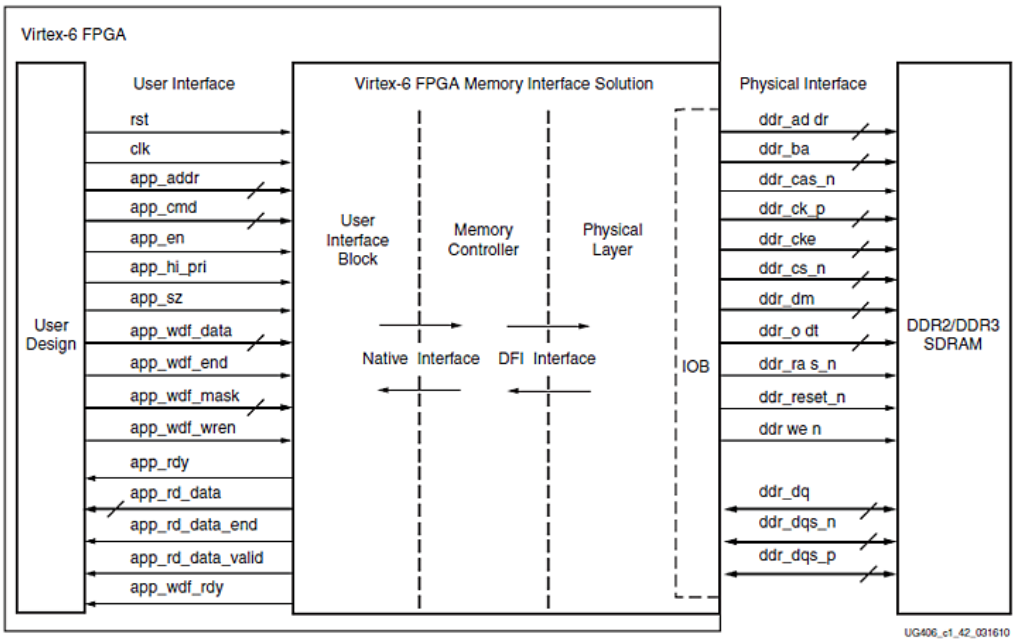
5.1.3 Vmesnik za dostop do pomnilnika

Kot je bilo že omenjeno, je smisel orodja MIG izdelava preprostega vmesnika za delo s pomnilnikom. Pomnilniški krmilnik, ki ga ustvari, ima na prednjem delu uporabniški vmesnik (angl. *User interface*). Gre za alternativo *domorodnemu vmesniku* (angl. *Native Interface*) [20], saj je z njim uporaba pomnilnika še dodatno poenostavljena. Z uporabniškim vmesnikom lahko programer pošilja pomnilniku ukaze za pisanje in branje. Programer komunicira s tem vmesnikom, tako je z minimalnim številom signalov mogoč popoln nadzor nad pomnilnikom. Na zadnjem delu krmilnika je *fizični vmesnik* (angl. *Physical Interface*), ki neposredno komunicira s pomnilnikom. Taka hierarhija je prikazana na sliki 5.6. Njen smisel je v tem, da je uporabniku na voljo poenostavljen uporabniški vmesnik, bolj napredni uporabniki pa lahko posežejo po domorodnem vmesniku, če želijo še bolj izboljšati hitrost dostopa do podatkov. Fizični vmesnik pa je implementacija protokola, po katerem se dostopa do DDR3 pomnilnika, a je za splošno uporabo preveč kompleksna. To je tudi razlog, da RAM potrebuje krmilnik, ki med drugim opravlja tudi osveževanje RAM-a.

Signali, ki jih je potrebno povezati z uporabniško logiko so opisani v tabeli 5.1.

Parameter *ADDR_WIDTH* ima vrednost 27 in nam pove, da za dostopanje do RAM-a potrebujemo naslov v obliki 27 bitne vrednosti. Posamezni biti naslova predstavljajo točno pozicijo pomnilniške besede v RAM-u, glede na format, ki je prikazan na sliki 5.7. Za izbiro kolone je potrebnih 10 bitov, za izbiro banke trije, za izbiro vrstice 13 in za izbiro ranga 2 bita, ki pa sta vedno enaka 0.

Parameter *APP_DATA_WIDTH* ima vrednost 256 in predstavlja širino podatkov, ki jih naenkrat damo krmilniku, da jih zapiše v RAM oz. širino podatkov, ki jih prebere iz njega. Gre torej za 256 bitov oz. 32 bajtov podatkov.



Slika 5.6: Krmilnik nam nudi preprostejši uporabniški vmesnik in bolj kompleksen domorodni vmesnik. Vir [20].

Parameter *APP_MASK_WIDTH* pa ima vrednost 32, kar pomeni da lahko maskiramo vsakega izmed 32 podatkovnih bajtov.

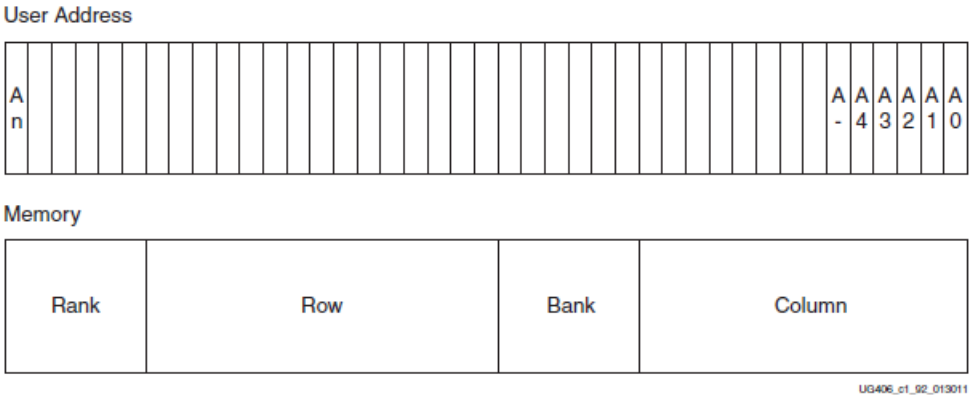
5.1.4 Potek pisanja in branja iz pomnilnika

Kot je bilo že omenjeno, je parameter *dolžina eksplozijskega cikla* enak 8 in je bil določen pri uporabi orodja MIG. Spomnimo, da to pomeni, da se bo z vsakim dostopom do pomnilnika dostopalo do 512 bitov podatkov.

Ko izvajamo pisanje ali branje, moramo krmilniku poslati le naslov v formatu, ki je bil opisan v prejšnjem poglavju. Logika krmilnika bo sama poskrbela za razporeditev vseh 512 bitov podatkov na prave lokacije po bankah, vrsticah in stolpcih pomnilnika.

Pisanje v RAM

Pisanje v pomnilnik s krmilnikom, ki ga ustvari MIG, je relativno preprosto. Možnih je več različnih načinov, opisan bo le tisti, ki je uporabljen v nalogi. Krmilniku moramo dostaviti vseh 512 bitov podatkov za pisanje. Prenosna



Slika 5.7: Format naslova za RAM pri uporabniškem vmesniku. Vir [20].

linija do krmilnika je široka le 256 bitov, torej najprej dostavimo spodnjih 256 bitov in nato še zgornjo polovico. Potek pisanja je sledeč:

1. Čakamo, da bo signal *app_wdf_rdy* v visokem stanju, to pomeni, da čakalna vrsta FIFO (angl. *First In, First Out*) lahko sprejme podatke za pisanje.

2. Prvo polovico podatkov postavimo na vodilo *app_wdf_data* ter aktiviramo signal *app_wdf_wren*, ki omogoči vpis v čakalno vrsto. Podatke držimo eno urino periodo.

3. Naslednjo periodo postavimo še drugo polovico podatkov ter aktiviramo signal *app_wdf_end*, ki označuje zadnji del podatkov. Še vedno držimo signal *app_wdf_wren*.

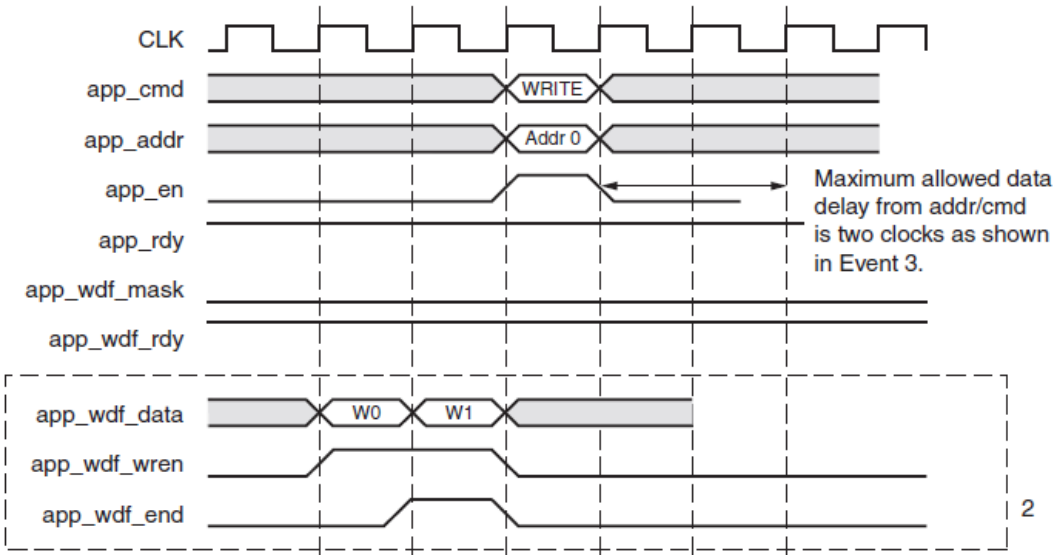
4. Deaktiviramo signala *app_wdf_wren* in *app_wdf_end*. Čakamo, da bo signal *app_rdy* v visokem stanju, kar označuje pripravljenost krmilnika za sprejem ukaza.

5. Krmilniku s signalom *app_cmd* sporočimo vrsto ukaza - pisalni ukaz ter s signalom *app_addr* še začetni naslov, kamor se bo izvajalo pisanje. Aktiviramo signal *app_en* in s tem pošljemo ukaz krmilniku.

Potek signalov pri pisanju na opisan način je viden na sliki 5.8.

Branje iz RAM

Branje iz pomnilnika poteka na podoben način kot pisanje in sicer:



Slika 5.8: Potek signalov pri pisanju v RAM. Vir [20].

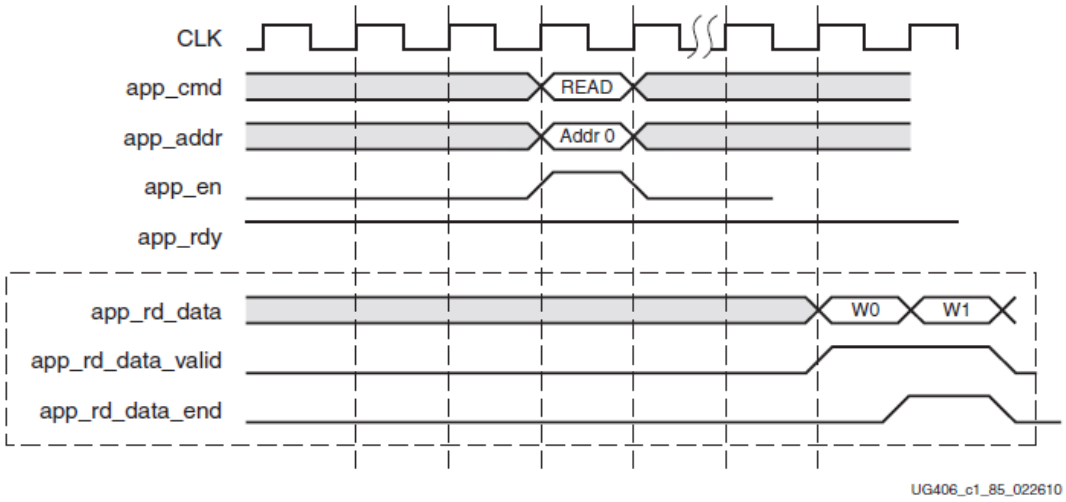
1. Čakamo, da bo signal *app_rdy* v visokem stanju - čakamo na pripravljenost za sprejem ukaza.

2. Signal *app_cmd* postavimo na vrednost, ki označuje bralni ukaz, na signal *app_addr* pa postavimo naslov prvega podatka. Aktiviramo signal *app_en* in s tem pošljemo krmilniku ukaz za branje.

3. Čakamo, da bo signal *app_rd_data_valid* aktiven. Označuje veljavnost spodnje polovice 512 prebranih bitov na vodilu *app_rd_data*.

4. V naslednji urini periodi se aktivira še signal *app_rd_data_end*, ki označuje veljavnost zgornje polovice podatkov na podatkovnem vodilu.

Potek signalov pri branju na opisan način je viden na sliki 5.9. Omenimo še, da je čas med sprejemom bralnega ukaza ob aktivaciji signala *app_en* in prihodom podatkov na podatkovno vodilo odvisen od več parametrov, med drugim tudi od periodičnega osveževanja pomnilnika, ki ga samodejno opravlja krmilnik. Če teh in še ostalih mogočih zakasnitev ne upoštevamo, je zakasnitev pri branju dolga 40, oziroma 38 urinih period, če je banka iz katere se bere že odprta. Te vrednosti veljajo za pomnilniško uro (400 MHz), ki ima dvakratno frekvenco osnovne ure (200 MHz).



Slika 5.9: Potek signalov pri branju iz RAM. Vir [20].

5.2 Komunikacija z razvojno ploščo

Potrebujemo način, ki omogoča prenos podatkov med računalnikom in razvojno ploščo FPGA. Eden izmed starejših, vendar uveljavljenih načinov komunikacije, je s pomočjo UART. Gre za asinhroni sprejemnik in oddajnik, ki omogoča serijsko komunikacijo preko serijske povezave. UART je na voljo v obliki namenskega čipa, lahko pa ga implementiramo sami s pomočjo FPGA.

Komunikacija poteka tako, da ima vsaka stran, torej prejemnik in oddajnik, svoj UART. Ob zahtevi za prenos le ta zaporedje bajtov podatkov pošilja bit za bitom, torej zaporedno. Prejemnik posamezne bite sprejema in jih združuje v popolne bajte. Pomembna komponenta UART je pomikalni register, saj omogoča serializacijo podatkov in prenos posameznih bitov [9].

UART sam po sebi ne generira dejanskih električnih signalnov, ki nato potujejo po kablu, ampak deluje le z internimi logičnimi nivoji [16]. Potreben je torej standard, ki omogoča serijski prenos bitov. Najbolj razširjen tak standard je RS-232. Natančno določa električne karakteristike signalov, njihove pomene ter tipe priključkov. RS-232 je bil nekoč primarni način za povezovanje perifernih naprav z računalnikom, vendar ga je zaradi počasne hitrosti prenosa in fizično velikih priključkov izrinil novejši, USB. Kljub temu je izjemno preprost za implementacijo in uporabo in zato ni popolnoma izumrl.

Standard RS-232 ne določa formata podatkov in prenosnih hitrosti. Te določa programer pri implementaciji UART modula.

Komunikacija lahko poteka enosmerno (angl. *simplex*), kot polni dupleks (angl. *full duplex*) ali polovični dupleks (angl. *half duplex*). Za naš problem je enosmerna komunikacija popolnoma zadostna, saj bomo vedno prenašali podatke le v eno smer naenkrat.

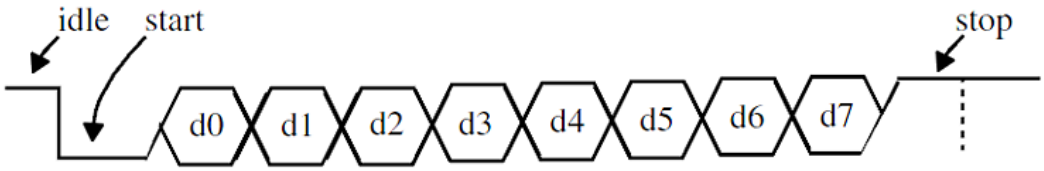
Priključek, ki ga priporoča standard RS-232, je imenovan *DB-25* in ima 25 nožic, vendar pogosto najdemo tudi *DB-9* 5.10, ki zavzema manj prostora, saj ima manj nožic. Za enostavno izvedbo serijske komunikacije niso nujno potrebne vse nožice, saj takrat potrebujemo le tri. Ena služi ozemljitvi, druga sprejemanju, tretja pa oddajanju podatkov. Poimenovani sta *RxD* ter *TxD*.

Slika 5.10: Pogosto uporabljen električni priključek pri serijski komunikaciji - DB-9.

5.2.1 Način komunikacije

Prenosna linija (RxD ali TxD) ima v mirovanju visoko logično stanje. Začetek prenosa označuje bit z nizkim logičnim stanjem, imenovan *start bit*, sledijo pa mu podatkovni biti, ki jih je tipično 8. Visoki biti so predstavljeni z visokim in nizki z nizkim logičnim stanjem. Konec prenosa označuje še en visok bit - imenovan *stop bit*. Potek take komunikacije je prikazan na sliki 5.11. Možnih je več različic podatkovnega okvirja, opisana je različica, ki je uporabljena v diplomski nalogi.

Potrebno je poudariti, da se po prenosni liniji ne prenaša urin signal. To pomeni, da morata sprejemna in oddajna stran uporabljati enake prenosne parametre, ki so vnaprej dogovorjeni. Med te spada prenosna hitrost, izražena



Slika 5.11: Potek komunikacije z UART, ko se prenaša 8 podatkovnih bitov.

v *baudih* (angl. *baud rate*). Baud predstavlja število prenesenih simbolov, v tem primeru bitov, na sekundo. Tipične hitrosti v baudih so 4800, 9200, 19200 in 115200.

5.2.2 Sprejemnik

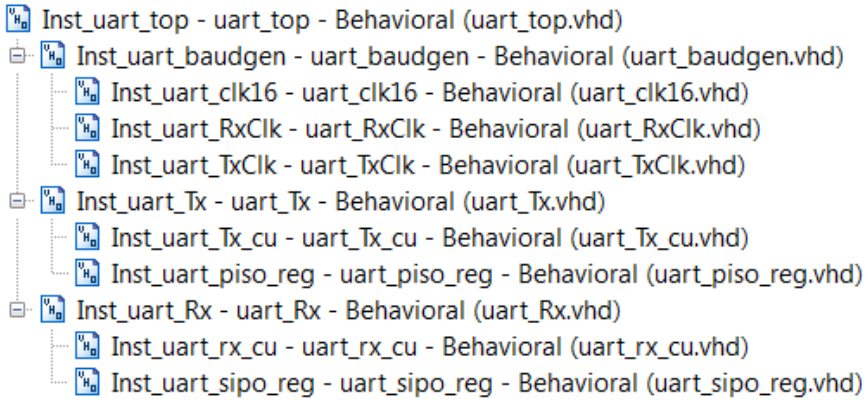
Sprejemni del uporablja uro, ki je večkratnik podatkovne hitrosti, v našem primeru je ta večkratnik 16. Ob fronti te ure bo sprejemnik preverjal stanje podatkovne linije in s tem prisotnost začetnega bita [16]. Če je stanje nizko vsaj polovico časa, ki je potreben za prenos bita, je bil start bit zaznan. Ob detekciji le tega začne sprejemnik preverjati stanje na liniji vsakič, ko se pričakuje sredina podatkovnega bita. Sprejeti biti se zapisujejo v pomikalni register. Po 8 prejetih bitih so v registru na voljo podatki, ki so bili sprejeti. Pri tem uporabimo zastavico, ki označuje, da je sprejet podatek na voljo.

5.2.3 Oddajnik

Sprejemnik deluje tako, da se podatke, ki so na vrsti za pošiljanje, najprej zapiše v pomikalni register. Nato generira začetni bit, premakne vsebino pomikalnega registra in na izhod postavi tako logično stanje, da ustreza trenutnemu bitu. Po prenosu vseh bitov se prenos zaključi s stop bitom. Zastavica označuje to, da je prenos trenutno v teku.

5.2.4 Implementacija UART na FPGA

Uporabimo modul UART, ki je predstavljen v [17]. UART je v VHDL realiziran kot modul *uart_top*, ki je sestavljen iz več podmodulov. Zgradba modulov je prikazana na sliki 5.12. Smiselna je delitev na modul, ki generira uro, s katero se sprejema in pošilja podatke - modul *uart_baudgen* - ter na modula za sprejem - *uart_Rx* - in pošiljanje - *uart_Tx*.



Slika 5.12: Hierarhičen pogled na implementacijo UART modula z VHDL.

Modul *uart_top* vsebuje vhodne in izhodne signale, ki so opisani v tabeli 5.2.

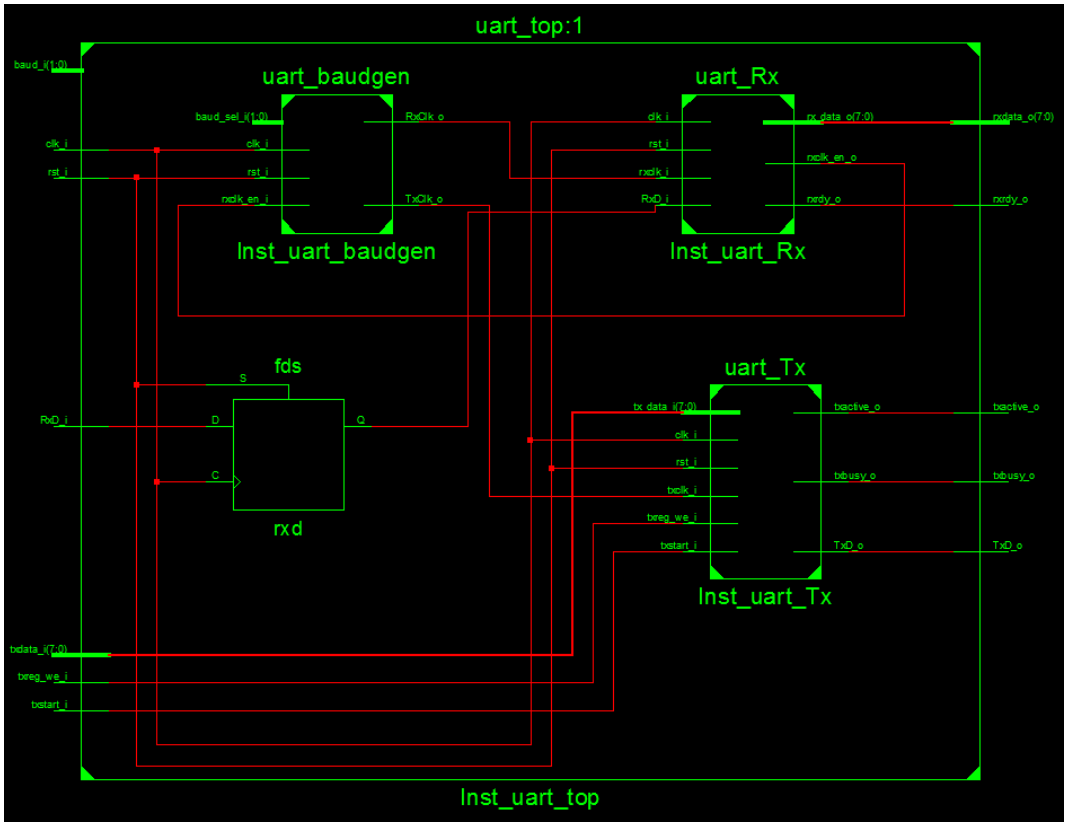
Xilinx ISE nam nudi izdelavo RTL (angl. *Register-Transfer Level*) sheme vezja. RTL shema pikazuje potek signalov med registri in logiko v vezju. Na sliki 5.13 vidimo modul *uart_top* z njegovimi vhodi in izhodi. Prikazane so tudi povezave vhodov in izhodov s tremi podmoduli, ki jih vsebuje. Vrhovni modul *uart_top* v bistvu vsebuje le te tri module ter D flip-flop, ki služi za sinhronizacijo vhodnega signala *RxD_i*.

Na sliki 5.14 vidimo celotno RTL shemo modula *uart_top*. Prikazani so vsi podmoduli do najnižjega nivoja, zato je nemogoče prikazati vse komponente zaradi njihovega velikega števila. Hiter pregled nam pove, da se v UART modulu nahaja veliko števcev, multiplekserjev ter osnovnih logičnih vrat. Vsekakor gre za modul, ki ne zasede veliko resursov FPGA čipa, saj zasede le okoli 100 rezin.

Modul za generacijo ure

Modul *uart_baudgen* vsebuje tri podmodule - *uart_clk16*, *uart_RxClk* in *uart_TxClk*. Vhodi in izhodi tega modula ter trije podmoduli so vidni na sliki 5.15.

Modul *uart_clk16* iz vhodne 200 MHz ure tvori uro, ki je 16-kratnik prenosne hitrosti po UART. Odločimo se za hitrost 115200 baudov - to je ena izmed višjih standardnih hitrosti. Izbira hitrosti v baudih je izvedena z 2-bitnim vhodnim signalom v modul *baud_sel_i*. Uro, ki ustreza izbrani hitrosti v baudih, realiziramo s števcem, ki šteje do neke vrednosti. Tako dobimo pulze, ki



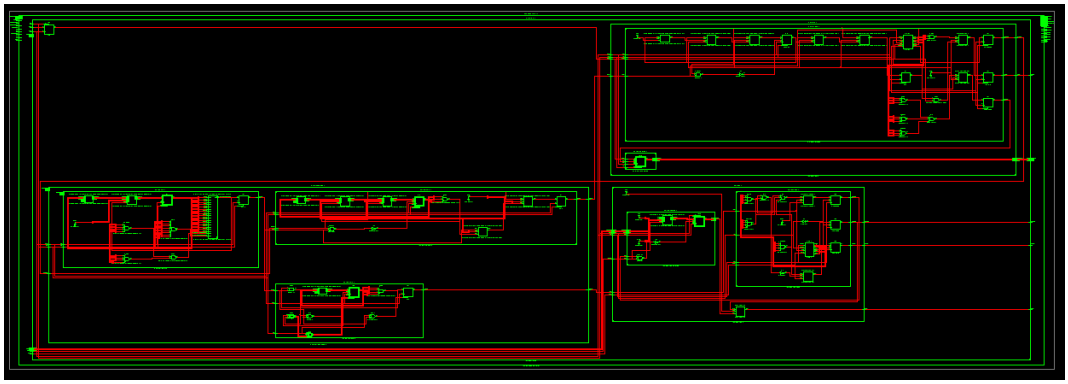
Slika 5.13: Shema vrhovnega UART modula *uart_top* s tremi podmoduli.

ustrezajo uri s 16-kratno frekvenco prenosne hitrosti. Izračunati moramo torej vrednost, do katere bo števec štel. S preoblikovanjem enačbe 5.1 pridemo do enačbe 5.2, ki nam pove vrednost, do katere je potrebno šteti.

$$prenosna\_hitrost[baud] = \frac{(\frac{frekvenca\_ure[Hz]}{vrednost\_stevca})}{16} \tag{5.1}$$

$$vrednost\_stevca = \frac{frekvenca\_ure[Hz]}{(prenosna\_hitrost[baud] * 16)} \tag{5.2}$$

Poleg te vrednosti vstavimo v enačbo 5.2 še frekvenco ure, ki znaša 200 MHz in ugotovimo, da je potrebno za to hitrost v modulu *uart_baudgen* šteti do vrednosti **109**. Ustvarjena ura, ki jo poimenujemo kar *clk16*, je izhod iz modula *uart_clk16* in se uporabi še za generiranje sprejemne in oddajne ure v



Slika 5.14: Shema vseh elementov modula UART in njegovih podmodulov.

drugih dveh modulih.

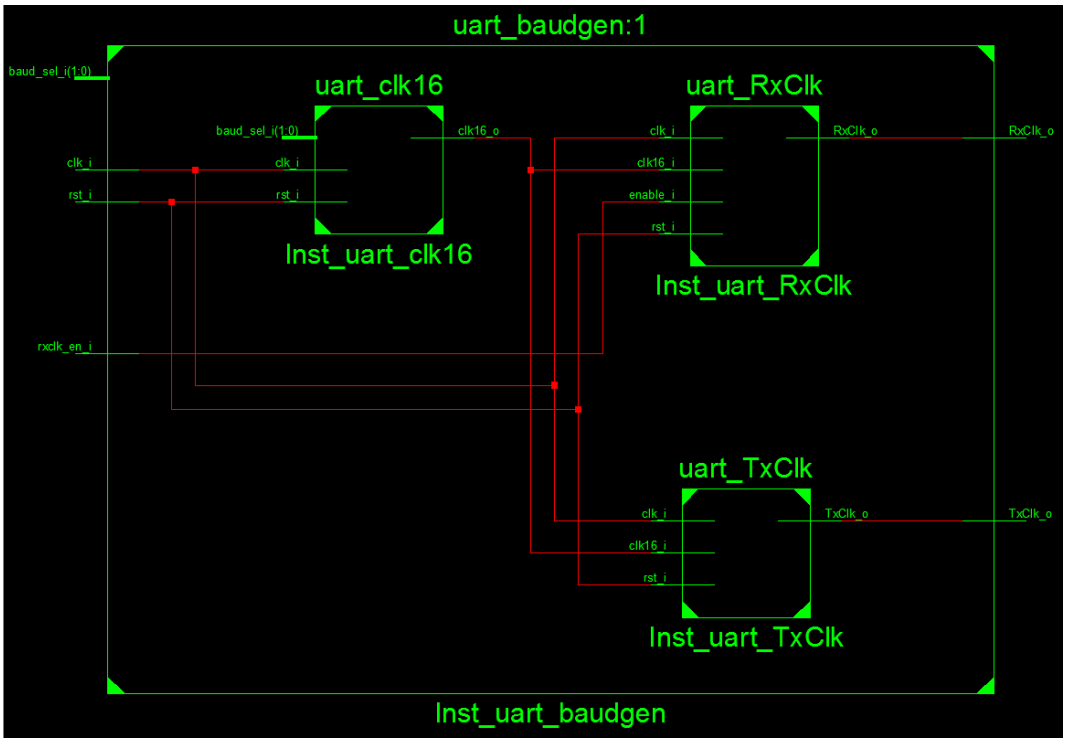
Sprejemna ura se v modulu *uart_RxClk* generira tako, da števec šteje 8 period vhodnega signala *clk16_i* - to je polovica časa za sprejem enega bita - in nato tvori pulz. To je izvedeno s procesom, ki se izvaja ob urinem signalu. Če ni aktiven signal za resetiranje, se števec ob visokem stanju signala *clk16* povečuje do vrednosti 8. Ko jo doseže tvori pulz v trajanju enega urinega cikla. Ta signal je imenovan *RxClk_o* in je izhod iz modula. Uporabi se ga v sprejemnem modulu *uart_Rx*.

Na enak način deluje modul *uart_TxClk*. Izhodni signal *TxClk_o*, ki je ura za oddajnik, se generira tako, da števec šteje 16 period ob visokem stanju signala *clk16* in nato prav tako tvori pulz.

Sprejemni modul

Sprejemni modul *uart_Rx* je sestavljen iz modula *uart_rx_cu*, ki služi kot kontrolna enota in modula *uart_sipo_reg*, ki služi kot pomikalni register. Shema sprejemnega modula je prikazana na sliki 5.16.

Vhod v kontrolno enoto je med drugim sprejemna ura iz modula *uart_RxClk*. Kontrolna enota je realizirana kot *Mealyjev* končni avtomat z 11 stanji. Avtomat preverja stanje vhodnega signala *RxD_i*, ob zaznanem start bitu pa s pomočjo sprejemne ure shranjuje sprejete bite enega za drugim v pomikalni register. Stanje prenosne linije *RxD_i* prebere vsakič, ko s sprejemno uro ugotovi, da gre za sredino trenutnega bita, ki se prenaša. Ob sredini bita aktivira signal *sipo_ce_o*, ki vodi v modul *uart_sipo_reg*. Ko se ta signal aktivira, se izvede zapis prebranega bita v 8-bitni pomikalni register ter izvede pomik v desno. Na ta način se v register zapiše vseh 8 sprejetih bitov. Takrat se akti-



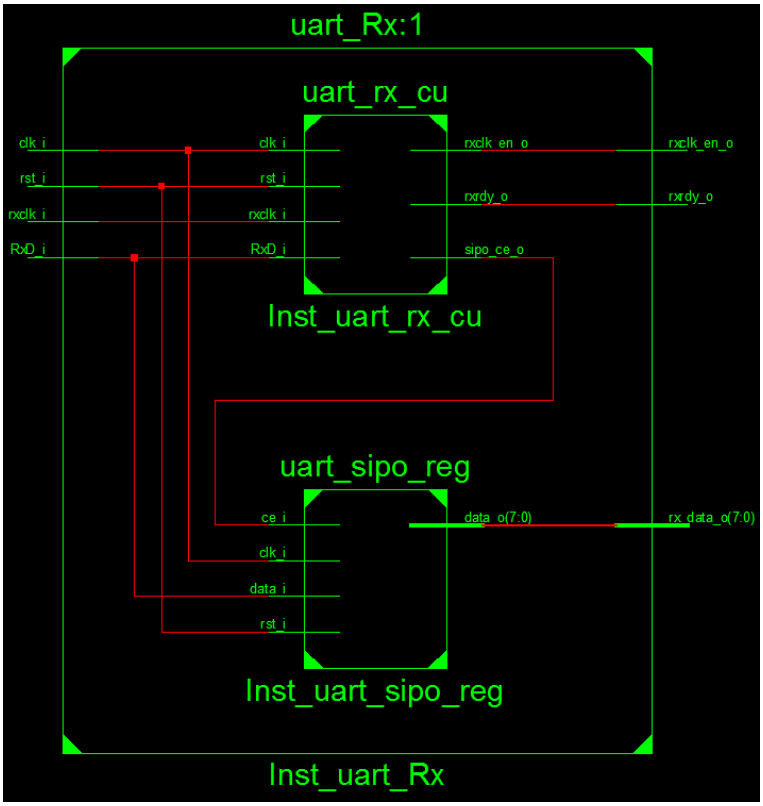
Slika 5.15: Shema modula *uart_baudgen* z vhodi, izhodi in tremi podmoduli.

vira izhodni signal *rxrdy_o*. To je zastavica, ki označuje veljavnost sprejetega bajta podatkov po serijski povezavi.

Oddajni modul

Oddajni modul *uart_Tx* je sestavljen iz modula *uart_Tx_cu*, ki služi kot kontrolna enota in modula *uart_piso_reg*, ki služi kot pomikalni register. Poleg teh dveh podmodulov vsebuje še multiplekser, ki oddajno linijo *TxD_o* drži v visokem stanju, ko prenos ni v teku. Shema oddajnega modula je prikazana na sliki 5.17.

1 bajt podatkov je potrebno najprej vpisati v pomikalni register modula *uart_piso_reg*. To storimo tako, da podatke postavimo na 8-bitni signal *data_i* in aktiviramo še signal *we_i*. Pomikalni register ima v tem primeru 10 bitov - najnižji bit predstavlja 0, ki označuje start bit. Sledi 8 podatkovnih bitov, najvišji bit pa je vrednost 1, ki predstavlja stop bit. Tudi tu kontrolna enota v modulu *uart_Tx_cu* vsebuje *Mealyjev* končni avtomat z 12 stanji. Vhoda v ta modul sta med drugim sprejemna ura iz modula *uart_TxClk* in še signal

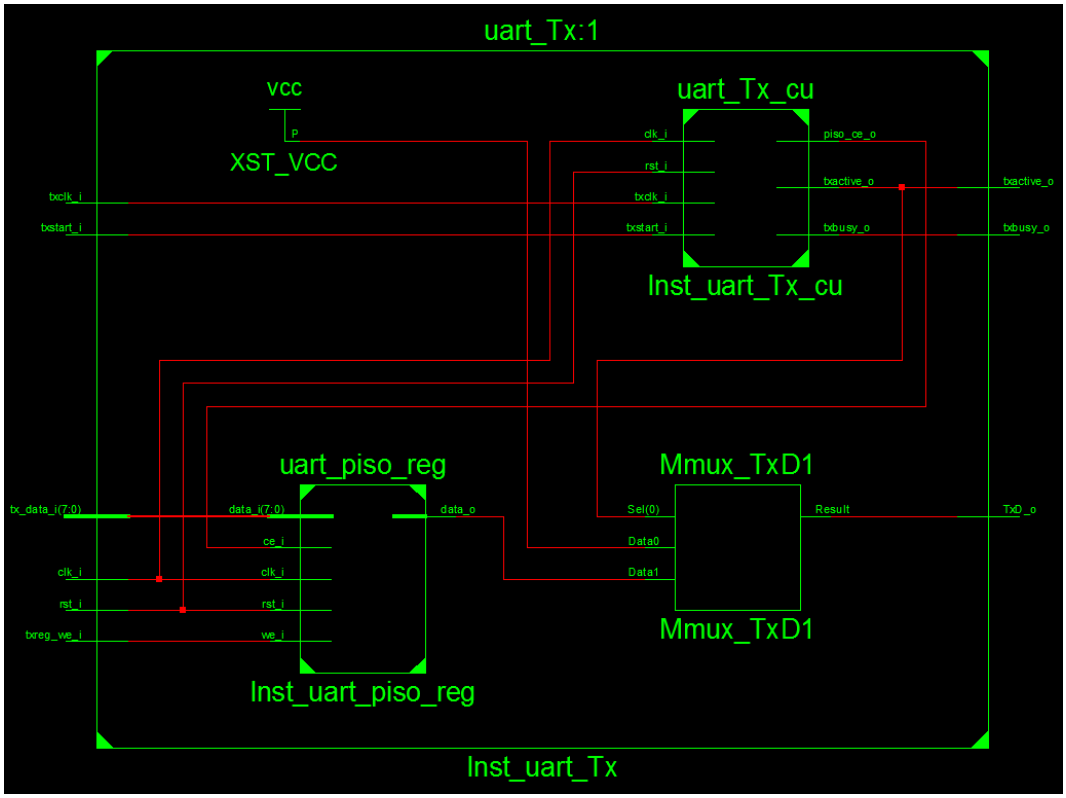


Slika 5.16: Shema modula *uart_Rx* z vhodi, izhodi in podmoduloma.

txstart_i, ki v visokem stanju označuje zahtevo za začetek pošiljanja podatkov. Avtomat čaka, da se bo ta signal aktiviral, ob njegovi aktivaciji pa aktivira izhodni signal *txbusy_o*, ki označuje, da je oddajnik zaseden. Avtomat bo nato čakal, da bo prenosna ura v visokem stanju, nato pa bo aktiviral signal *piso_ce*. Ta signal je vhod v modul *uart_piso_reg*. Vsakič, ko se bo aktiviral ta signal, bo izhod tega modula najnižji bit pomikalnega registra. Na začetku prenosa bo to start bit, nato podatkovni biti in stop bit. Na ta način se prenesejo vsi podatkovni biti.

5.3 Povezava modulov v celoto

Izdelan UART in krmilnik RAM-a je potrebno povezati in izdelati logiko, ki bo sprejemala podatke preko serijske povezave, jih zapisovala v RAM ter nato tudi prebrala iz RAM-a in poslala nazaj po serijski povezavi na računalnik. Mogoča



Slika 5.17: Shema modula *uart_Tx* z vhodi, izhodi in podmoduloma.

je enostavna dopolnitev z logiko, ki podatke še dodatno procesira preden jih zapiše v pomnilnik, s tem pa naša naloga dobi praktično uporabo.

Smiselna je rešitev z uporabo končnega avtomata, ki izvaja korake, s katerimi lahko uresničimo zapisovanje in branje iz RAM-a. V jeziku VHDL izdelamo modul, imenovan *app_controller*, ki komunicira s krmilnikom za RAM. Modul vsebuje instanco modula UART ter še modul, ki vsebuje niz registrov za začasno shranjevanje 512 bitov podatkov.

Uporabimo *Mealyjev* končni avtomat, kjer so prehodi med stanji odvisni od trenutnega stanja in od trenutnih vhodov. Gre za relativno preprost avtomat s 25 stanji. Njegovo poenostavljeno delovanje je predstavljeno z diagramom poteka na sliki 5.18.

Kot je razvidno iz diagrama poteka, potrebujemo dva števca. Prvi je števec kosov podatkov, kjer en kos predstavlja 64 B podatkov oz. toliko, kot se jih piše ali bere iz RAMa. Drugi števec je namenjen štetju posameznih bajtov enega kosa podatkov. Vsakič, ko UART sprejme 1 B podatkov, FPGA računalniku

pošlje kontrolni znak, ki pomeni zahtevo za naslednji bajt podatkov. Gre torej za asinhrono komunikacijo, saj FPGA dobi nove podatke takrat, ko izda zahtevo. Mogoča bi bila sinhrona rešitev, kjer bi računalnik pošiljal 1 B podatkov na npr. vsakih nekaj milisekund, vendar je izbrana rešitev bolj zanesljiva.

Podatki, ki jih UART sprejema, se bajt za bajtom shranjujejo v niz 64 8-bitnih registrov. Registrski niz je implementiran v modulu *register_file*. Ob pisanju v RAM se uporabi podatke iz teh registrov in se ga nato ponovno napolni s sprejetimi podatki iz UART. Ko so vsi kosi podatkov zapisani v RAM, sledi branje podatkov v registrski niz in nato pošiljanje posameznih bajtov z UART nazaj na računalnik. To branje iz RAM-a se ponavlja, dokler ne pošljemo na računalnik vseh kosov podatkov.

Brez težav bi lahko dodali še logiko, ki izvaja procesiranje podatkov, preden se ti zapisujejo v pomnilnik, pri tem bi šlo lahko za kakršnokoli vrsto podatkov, npr. slike ali zvok. Smisel procesiranja podatkov na FPGA je v doseganju mnogo večjih hitrosti, saj je procesiranje na strojnem nivoju mnogo hitrejše kot enako procesiranje na splošnonamenskem procesorju.

Rešiti je potrebno še problem, koliko kosov podatkov se bo prenašalo na FPGA. Odgovor je odvisen od same aplikacije in procesiranja podatkov, ki se ga naknadno lahko doda naši nalogi. V primeru, da bo šlo za procesiranje slik in ob predpostavki, da bo slika velika nekaj MB, en kos podatkov pa 64 B, se bo na ploščo prenašalo več 10.000 kosov podatkov. Gre za zelo veliko število podatkov, kar pomeni dolg čas prenosa s serijsko povezavo. Za demonstracijske namene lahko število kosov ustrezno zmanjšamo. Ko se odločimo za neko število kosov podatkov, ga zapišemo kot parameter v VHDL kodi, tako da bo plošča vedno pričakovala toliko podatkov.

Podatke se začne zapisovati v prvo banko, prvo vrstico in prvi stolpec. Naslov za dostop do RAM-a krmilniku podamo s signalom *app_addr_o*, ki je sestavljen iz 27 bitov. Na začetku vse bite signala postavimo na vrednost 0. Za vsak naslednji pisalni ali bralni ukaz ta signal ustrezno spreminjamo. To izvedemo tako, da ga iz tega začetnega tipa - vektorski signal - pretvorimo v celo število. Nato lahko naslov preprosto povečamo z operacijo seštevanja. Vsakič prištejemo desetiško vrednost 32. Tako bodo podatki zapisani v sosednje celice, vmes pa ne bo neporabljenih prostorov. Številka 32 zato, ker je to število pomnilniških besed, do katerih dostopa en RAM ukaz.

Slika 5.19 prikazuje končen pogled na hierarhijo VHDL modulov. Modul *example_top* predstavlja glavni modul, ki je izpeljan iz projekta *example_project*, ki ga ustvari MIG. Vsebuje modula *u_iodelay_ctrl* in *u_infrastructure*, ki upravljata z urinim signalom. Sledi modul *u_memc_ui_top*, ki predstavlja uporabniški vmesnik do RAMa ter vsebuje celotno logiko krmilnika. Zaradi

velikega števila podmodulov krmilnika RAM ti niso prikazani. Naslednji modul je *Inst_app_controller*, ki je instanca krmilnega avtomata - vsebuje modul UART ter še registrski niz. Na dnu hierarhije opazimo še datoteko *example_top.ucf*, ki je datoteka z uporabniškimi omejitvami, kot je bilo omenjeno že v predhodnih poglavjih.

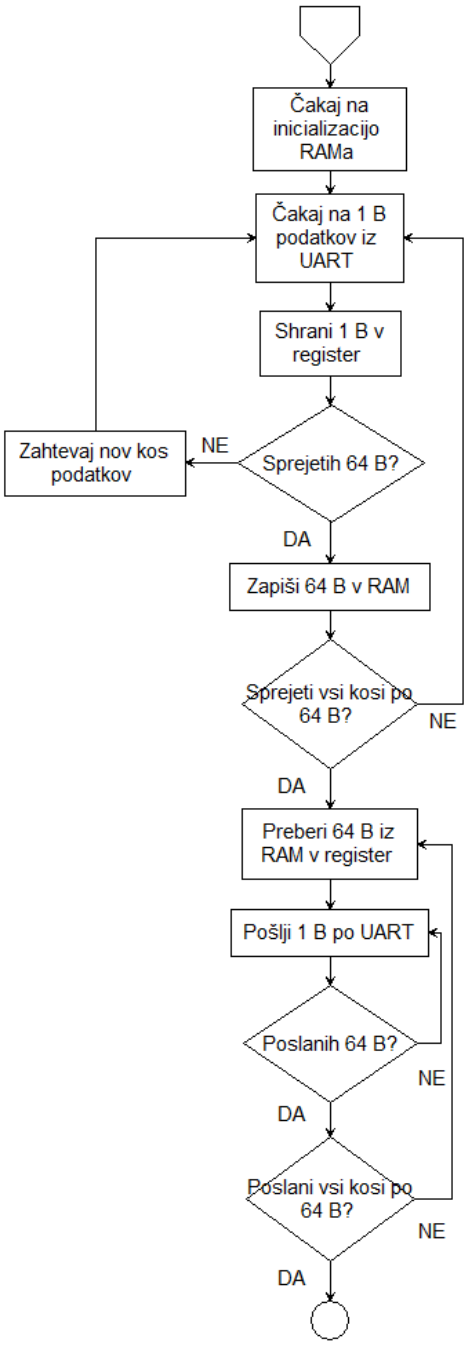
Ob izvedbi sinteze vezja na sliki 5.20 vidimo, da pri plošči Virtex-6 ML605 ni nikakršnih težav s časovnimi omejitvami. Vidna je tudi nizka stopnja zasedenosti logičnih celic plošče, saj je ta razvojna plošča zelo zmogljiva in ponuja veliko prostora za postavitev komponent.

rst	Vhod	Resetiranje - aktiven ob visokem stanju
clk	Vhod	Ura s polovično frekvenco hitrosti pomnilnika
phy_init_done	Izhod	Signal se aktivira, ko je opravljena inicializacija pomnilnika
app_addr [ADDR_WIDTH–1:0]	Vhod	Naslov za trenutni ukaz
app_cmd[2:0]	Vhod	Izbira ukaza za pomnilnik
app_en	Vhod	Kontrolni signal za začetek izvajanja ukaza
app_hi_pri	Vhod	Signal za povečanje prioritete trenutnega ukaza
app_sz	Vhod	Kontrolni ukaz, ki mora biti za DDR3 pomnilnik vedno 0
app_wdf_data [APP_DATA_WIDTH–1:0]	Vhod	Podatki za pisanje v RAM
app_wdf_end	Vhod	Signal, ki označuje drugo polovico podatkov za pisanje
app_wdf_mask [APP_MASK_WIDTH–1:0]	Vhod	Signal za maskiranje podatkov
app_wdf_wren	Vhod	Omogočanje pisanja v čakalno vrsto za pisanje
app_wdf_rdy	Izhod	Označuje pripravljenost čakalne vrste
app_rdy	Izhod	Označuje pripravljenost za sprejem novega ukaza
app_rd_data [APP_DATA_WIDTH–1:0]	Izhod	Prebrani podatki iz RAM
app_rd_data_end	Izhod	Označuje drugo polovico prebranih podatkov
app_rd_data_valid	Izhod	Označuje veljavnost prebranih podatkov

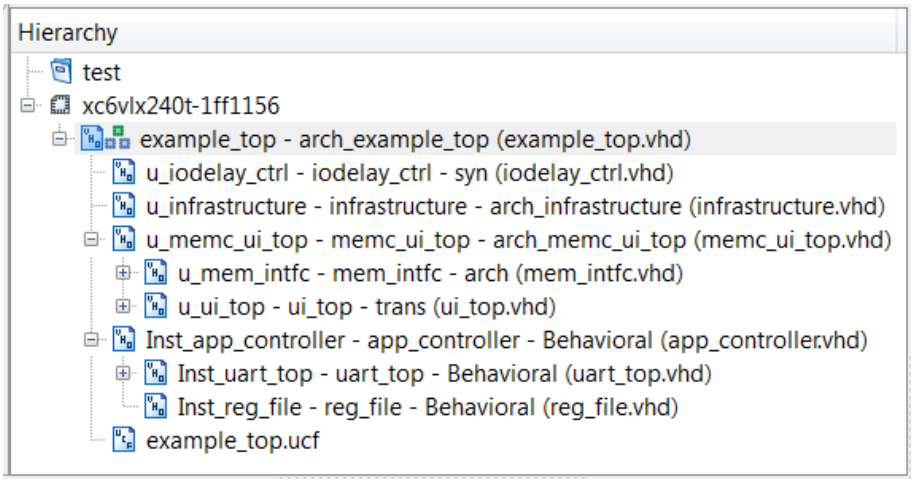
Tabela 5.1: Signali modula za uporabniški vmesnik, s katerimi ima opravka programer.

clk_i	Vhod	Ura
rst_i	Vhod	Signal za resetiranje
RxD_i	Vhod	Signal za sprejem serijskih podatkov
txstart_i	Vhod	Signal za začetek pošiljanja podatkov
txreg_we_i	Vhod	Omogočanje vpisa podatka za pošiljanje
baud_i[2]	Vhod	Nastavitev hitrosti prenosa
txdata_i[8]	Vhod	Podatki za pošiljanje
rxrdy_o	Izhod	Označuje pripravljenost sprejetih podatkov
txbusy_o	Izhod	Označuje, da je sprejemnik zaseden
txactive_o	Izhod	Označuje, da je pošiljanje v teku
TxD_o	Izhod	Signal za pošiljanje serijskih podatkov
rxdata_o[8]	Izhod	Sprejeti podatki

Tabela 5.2: Signali modula UART.



Slika 5.18: Diagram poteka krmilnega avtomata, ki povezuje UART s krmilnikom RAMa.



Slika 5.19: Hierarhičen pogled VHDL modulov.

example_top Project Status (06/03/2012 - 17:26:12)			
Project File:	test.xise	**Parser Errors:**	No Errors
Module Name:	example_top	**Implementation State:**	Programming File Generated
Target Device:	xc6vlx240t-1ff1156	**• Errors:**	No Errors
Product Version:	ISE 13.3	**• Warnings:**	1265 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	6,496	301,440	2%	
Number of Slice LUTs	5,487	150,720	3%	
Number used as Memory	508	58,400	1%	
Number of occupied Slices	2,722	37,680	7%	
Number of LUT Flip Flop pairs used	7,807			
Number with an unused Flip Flop	1,980	7,807	25%	
Number with an unused LUT	2,320	7,807	29%	
Number of fully used LUT-FF pairs	3,507	7,807	44%	
Number of unique control sets	346			
Number of slice register sites lost to control set restrictions	1,364	301,440	1%	
Number of bonded IOBs	126	600	21%	

Slika 5.20: Del končnega poročila pri sintezi vezja.

Poglavje 6

Simulacija delovanja

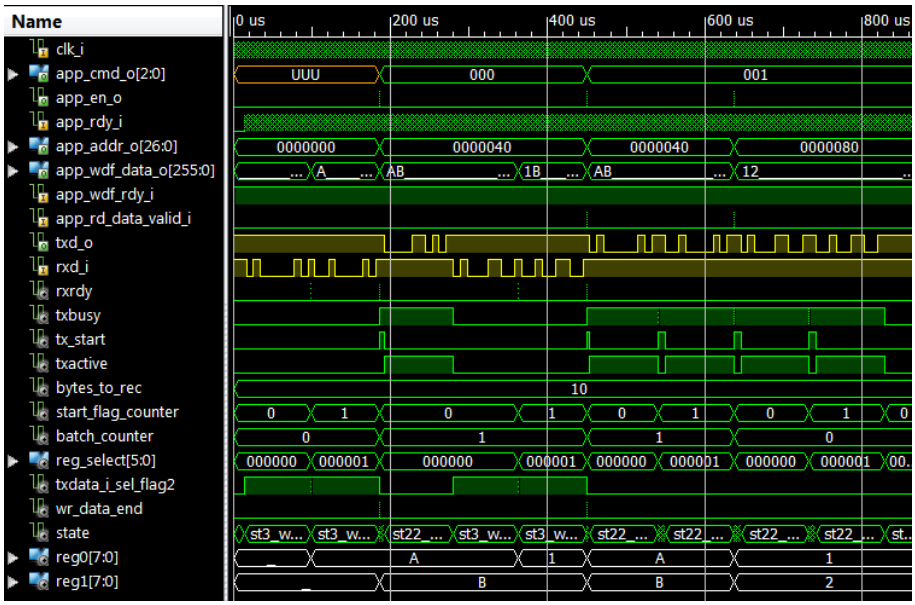
Razvojno okolje Xilinx ISE vsebuje tudi simulacijsko orodje ISim. Gre za nepogrešljivo orodje, saj nam omogoča izdelavo testnega okolja (angl. *test bench*). Sami določimo vhode v vezje, ki ga izdelujemo, nato pa nam simulator prikaže stanje notranjih signalov skozi čas.

Problem simulatorja je v tem, da ob naraščanju kompleksnosti vezja hitrost simulacije izjemno pada. Ob preverjanju delovanja modula UART je simulator za simulacijo poteka signalov za čas nekaj 100 μs potreboval le nekaj sekund. Ob prisotnosti krmilnika RAM v dizajnu je čas za simulacijo narasel na več ur. Izbrati je torej potrebno strategijo, ki nam omogoča simulacijo v še sprejemljivem času.

Smiselno je simuliranje posameznih modulov posebej in nato združevanje teh v končni dizajn. Tako najprej izvedemo simulacijo delovanja modula UART, nato pa še pomnilnika. Omeniti je potrebno, da simulator ISim sam po sebi ne omogoča simuliranja delovanja pomnilnika. To je mogoče le z modelom uporabljenega pomnilnika, ki ga ISim uporabi ter ga z njim simulira. Tak model generira MIG ob izdelavi krmilnika.

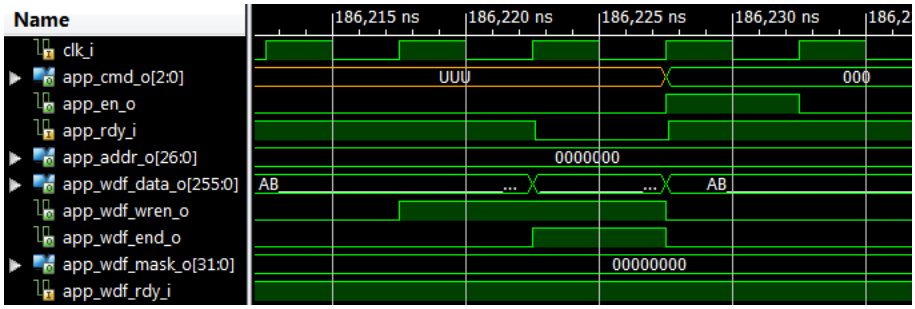
Za simulacijo celotnega dizajna nekoliko spremenimo nekatere parametre, in sicer število bajtov v enem podatkovnem kosu iz 16 zmanjšamo na 2. Zmanjšamo tudi število pričakovanih kosov podatkov na 2. Nato izdelamo testno okolje, ki simulira sprejem štirih bajtov podatkov. V prvem kosu podatkov razvojna plošča sprejme ASCII (angl. *American Standard Code for Information Interchange*) znaka *A* in *B*, v drugem pa *1* in *2*. Če želimo preveriti, ali dizajn res deluje, bi se morala v začasnem registru pojaviti znaka A in B in se zapisati v RAM. Nato se v registrih pojavita 1 in 2 ter se zapišeta v RAM. Sledilo naj bi branje iz pomnilnika - v registru pričakujemo znaka A in B ter nato pošiljanje obeh z UART. Slediti morata še prebrana znaka 1 in

2 v registru in pošiljanje z UART. Točno tako obnašanje vidimo na sliki 6.1, kar je potrditev pravilnega delovanja vezja.



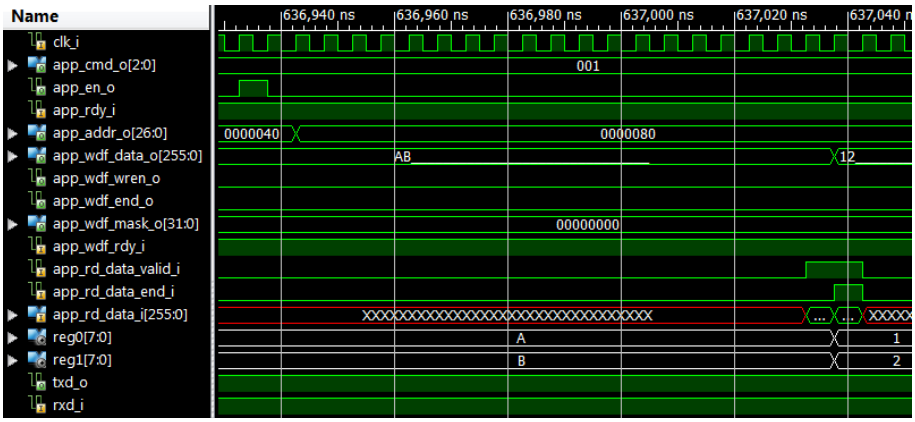
Slika 6.1: Simulacija delovanja vezja - sprejem podatkov, pisanje, branje in pošiljanje nazaj.

Na sliki 6.2 je bolj podrobno prikazano pisanje v pomnilnik. Opazimo, da je to izvedeno po enakem postopku kot je prikazano na sliki 5.8.



Slika 6.2: Simulacija pisanja v RAM.

Slika 6.2 prikazuje postopek branja iz RAMa. Tudi tu opazimo enak postopek kot na sliki 5.9. Vidna je tudi zakasnitev med potrditvijo bralnega ukaza in prihodom, ki znaša 19 urinih period ure.



Slika 6.3: Simulacija branja iz RAM-a.

Poglavje 7

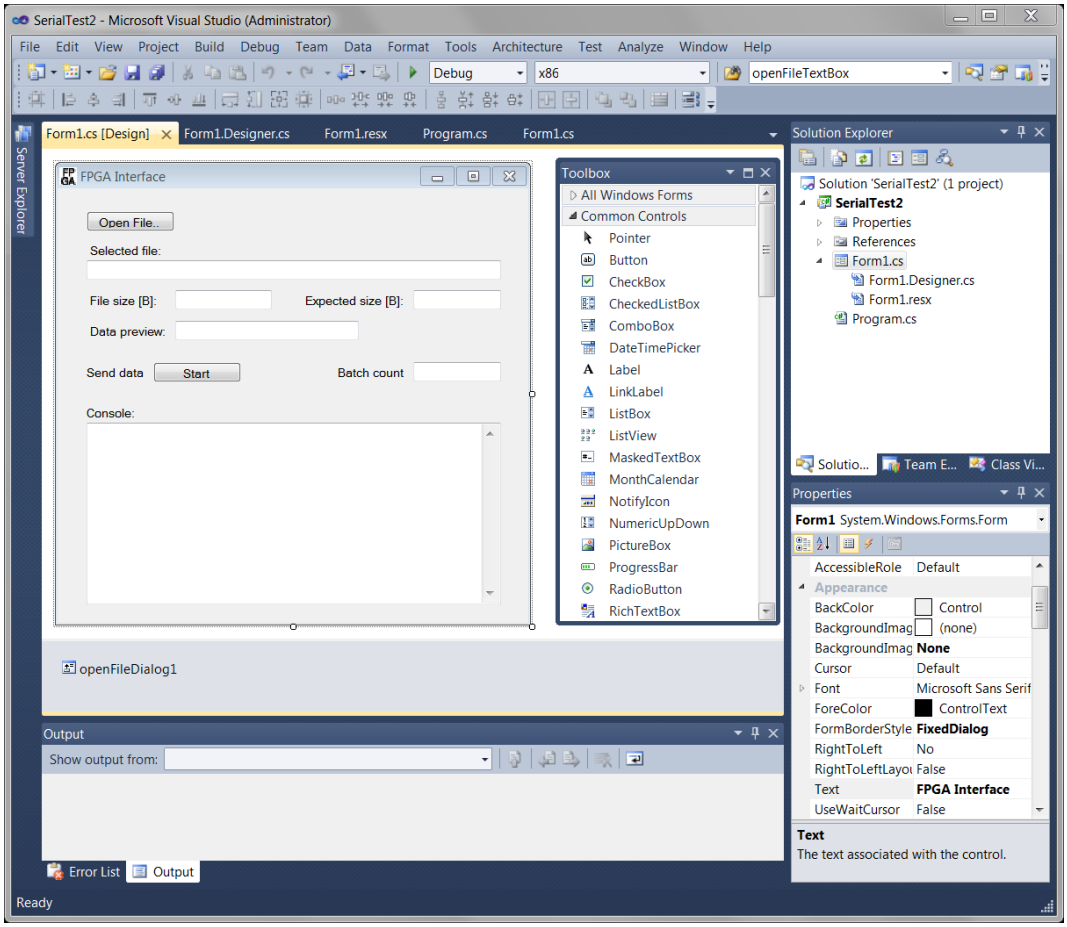
Izdelava uporabniškega grafičnega vmesnika

Preprosto pošiljanje in sprejemanje podatkov iz FPGA lahko izvedemo z izdelavo programa z grafičnim vmesnikom. Tak program izdelamo z uporabo razvojnega okolja *Microsoft Visual Studio 2010*, s programskim jezikom *C#*. Visual Studio nam omogoča enostavno oblikovanje grafičnega vmesnika s pomočjo *oblikovalca obrazcev* (angl. *form designer*). Z njim vse komponente (gumbe, napise, okvire za tekst, itd.) preprosto postavljamo na okno, ki predstavlja grafični vmesnik. Tem komponentam lahko priredimo akcije, ki se izvedejo, če uporabnik nad njimi izvede npr. levi klik z miško.

Slika 7.1 prikazuje pogled, kjer postavljamo elemente na grafični vmesnik. Odločimo se, da mora program omogočati pošiljanje podatkov na FPGA, ki se nahajajo v datoteki, nato pa podatke, ki jih FPGA vrne, shraniti v drugo datoteko. Za ta namen na vrh okna z grafičnim vmesnikom namestimo gumb, ki odpre brskalnik po datotekah, izbrana datoteka s podatki pa bo uporabljena kot vir podatkov. Ob izbiri datoteke se prikaže še pot do izbrane datoteke ter predogled prvih 8 bajtov vsebine datoteke. Prikaže se tudi velikost datoteke v bajtih in pričakovano velikost datoteke na FPGA. Ta parameter se mora ujemati s številom pričakovanih podatkov v VHDL kodi.

Grafični vmesnik vsebuje še gumb, s katerim začnemo prenos podatkov na FPGA, procesiranje (če je to implementirano) in sprejem podatkov ter zapisovanje v novo datoteko. Desno od gumba se nahaja še števec kosa podatkov, ki se trenutno prenaša na FPGA.

Sprejete in procesirane podatki iz FPGA-ja nam program zapiše v datoteko z enakim imenom kot ga ima vhodna datoteka, le da ji doda pripono *PROCESSED*. Velikost datoteke mora biti manjša ali enaka velikosti podatkov, ki jih



Slika 7.1: Oblikovanje grafičnega vmesnika z Visual Studio 2010.

pričakuje FPGA. Če bo velikost izbrane datoteke enaka pričakovani vrednosti, bodo na FPGA poslani vsi bajti datoteke. Če pa bo datoteka manjša, se bodo manjkajoči bajti podatkov nadomestili z ničlami, te pa bodo poslane na FPGA namesto manjkajočih podatkov. To tudi pomeni, da bo FPGA poslal nazaj podatke, ki vsebujejo dodane ničle. Ko so sprejeti vsi podatki in zapisani v datoteko, te ničle preprosto izbrišemo iz datoteke, saj je to najenostavnejša rešitev.

Aplikacija deluje tako, da se ob njenem zagonu kliče metoda *Form1_Load()*. *Form1* je ime osnovnega okna grafičnega vmesnika. Smiselno je, da se ob zagonu vzpostavi povezava z FPGA preko serijskega vmesnika. Komunikacija z uporabo serijske povezave je v Visual Studiu z uporabo C# preprosta. Izvedena je z ustvarjanjem novega objekta tipa *SerialPort*, ki mu podamo vse

potrebne parametre - številka COM vrat, hitrost prenosa v baudih, paritetne nastavitve, število podatkovnih bitov ter število stop bitov. Serijski vmesnik nam je dostopen preko privatne spremenljivke *_serialPort*, ki jo instanciramo z naslednjimi parametri:

```
SerialPort("COM30", 115200, Parity.None, 8, StopBits.One)
```

S tem je bil ustvarjen serijski vmesnik, na vratih COM30, s hitrostjo 115200 baudov, brez paritete ter z 8 podatkovnimi biti in enim stop bitom. V primeru, da serijska povezava na vratih COM30 še ni odprta, jo odpremo z metodo *_serialPort.Open()*. Če so vrata že zasedena, se uporabniku prikaže opozorilo z napako. Nastaviti je potrebno še parametre, kot so dovoljen čas za sprejem in oddajo podatkov ter določitev metode, ki se bo klicala ob dogodku, ko FPGA pošlje podatke. To storimo tako, da atributu _serialPort.DataReceived priredimo nov objekt na način:

```
SerialDataReceivedEventHandler(sp_DataReceived)
```

S tem je določeno, da se bo ob dogodku, ko so sprejeti podatki, klicala metoda *sp_DataReceived*.

Po zagonu aplikacije in inicializaciji serijske povezave je aplikacija v mirovanju. V zanki se preverja, ali se zgodi kakšen dogodek (v našem primeru pritisk na gumb). Ob pritisku na gumb "Open File...", se kliče metoda *btn_openFile_Click_1*. Ta izvede odpiranje dialoga za izbiro datoteke. Ob uspešni izbiri se ime ter pot do datoteke shranita v spremenljivki tipa niz, saj ju bomo potrebovali pri ustvarjanju nove datoteke. Izvede se tudi preverjanje velikosti datoteke. V primeru, da je ta manjša od pričakovane velikosti, se izračuna število manjkajočih bajtov podatkov in se zapiše v spremenljivko *requiredZeros*. S celoštevilskim deljenjem te vrednosti s 64 (število bajtov v enem kosu podatkov) pa dobimo število kosov podatkov, ki bodo vsebovali same ničle. Ta vrednost se shrani v spremenljivko *requiredEmptyBatches*.

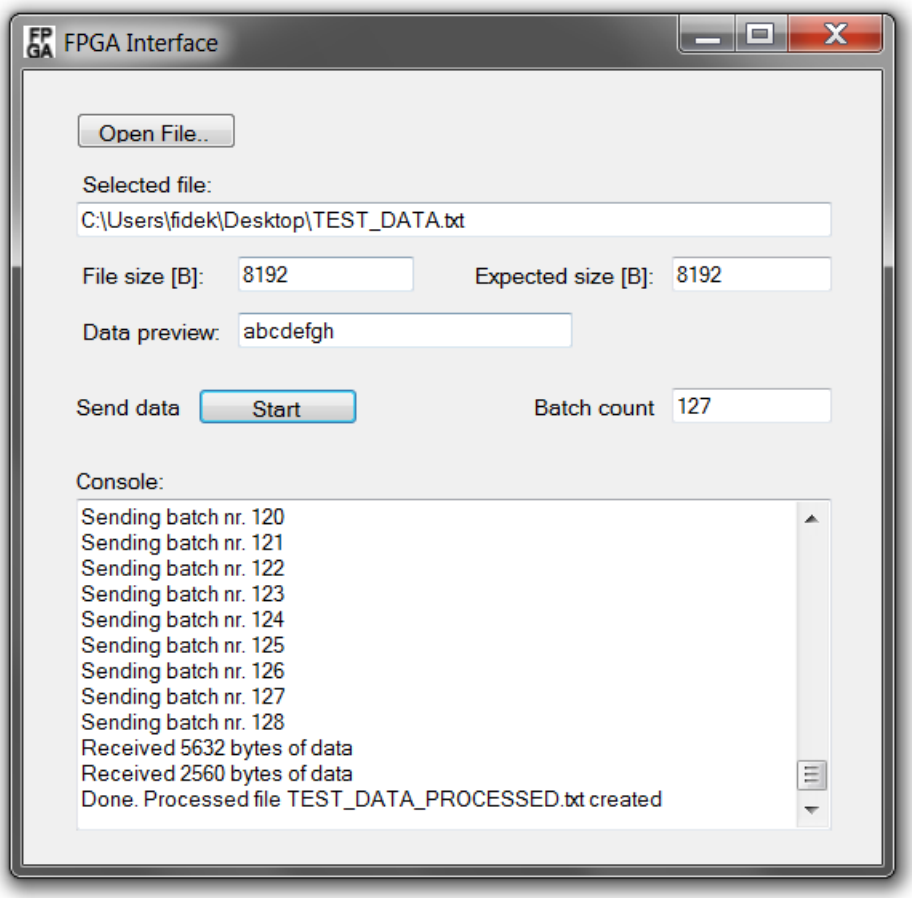
Aplikacija nato čaka na pritisk gumba "Start" - takrat se bo poklicala metoda *btnSendFile_Click*, ki v bistvu samo sproži klic metode *sendDataBatch()*. Ta metoda izvaja pošiljanje kosov podatkov. Najprej preveri, ali je bil zahtevan kos podatkov - na začetku je spremenljivka, ki to označuje v stanju "true" (resnično), po pošiljanju enega kosa podatkov pa se postavi na vrednost "false" (neresnično). Preveri se tudi, ali je števec kosov podatkov *batchCounter* manjši od števila vseh kosov *BATCH_NR*. V primeru, da sta oba pogoja izpolnjena, se glede na to, ali je potrebno del podatkov dopolniti z ničlami, izbere ustreznih

64 bajtov podatkov, ki se bodo poslali. Ti se shranijo v spremenljivko tipa niz *dataToSend* in se pošljejo na FPGA z metodo *_serialPort.Write(dataToSend);*.

Po poslanem kosu podatkov sledi sprejem kontrolnega znaka "X". Ko so sprejeti podatki, se izvede metoda *sp_DataReceived()* - v njej se sprejete podatke iz bralnega predpomnilnika prebere z metodo *_serialPort.ReadExisting()*. Prebrane podatke se preda metodi *si_DataReceived(string data)*, ki v primeru, da je sprejela kontrolni znak "X", ustrezno poveča števec *batchCounter*, spremeni stanje spremenljivke *nextBatchRequested* v "true" ter pokliče metodo *sendDataBatch()*. To se ponavlja, dokler niso poslani vsi kosi podatkov. Ko sprejet podatek ni kontrolni znak, to pomeni, da je FPGA začel pošiljati nazaj procesirane podatke. Takrat se kliče metoda *writeToFile()*. Ta metoda je enostavna - v primeru, da datoteka s pripono *PROCESSED* še ne obstaja jo ustvari, nato pa datoteki pripne sprejete podatke. Ta metoda se bo klicala, dokler bodo iz FPGA-ja prihajali podatki. Rezulat je nova datoteka s procesiranimi podatki iz FPGA-ja.

Na sliki 7.2 vidimo izgled aplikacije z grafičnim vmesnikom. Pričakovana velikost podatkov je v tem primeru postavljena na 8 kB, kar nam da 128 kosov podatkov po 64 B. Vidimo, da je bilo procesiranje končano, rezultat pa shranjen v novo datoteko.

Grafični vmesnik vsebuje tudi večje tekstovno polje, ki služi kot konzola za obveščanje uporabnika o trenutnem stanju. Ob zagonu in uspešni povezavi na serijska vrata se o tem obvesti uporabnika. Ob pošiljanju kosov podatkov na FPGA se sproti izpisuje, kateri kos se pošilja. Ob koncu pošiljanja na FPGA se tudi to izpiše na konzolo. Ob sprejemu podatkov iz FPGA-ja pa se vsakič izpiše, koliko bajtov podatkov je bilo sprejetih. Na koncu je uporabnik obveščen še o tem, v katero datoteko so bili zapisani rezultati iz razvojne plošče.



Slika 7.2: Uporabniški vmesnik v Windows okolju - procesiranje podatkov je bilo končano.

Poglavje 8

Zaključek

Po zamenjavi razvojne plošče je bil cilj naloge v celoti dosežen. Poskus izvedbe na plošči Spartan-3A je bil neuspešen zaradi neustreznosti le te za zastavljeno nalogo. Logika za modul UART ter krmilnik za pomnilnik DDR2 očitno nista bili združljivi, saj so bile zakasnitve na povezavah prevelike in so povzročile nezanesljivo delovanje. Rešitev za ta problem morda obstaja, vendar bi bil smisel naloge zgrešen, če bi se posvečali samo optimizaciji delovanja. Prav tako bi bila vprašljiva praktična uporabnost take naloge, saj bi bila z dodajanjem logike, ki procesira podatke na čipu FPGA, najverjetneje potrebna ponovna optimizacija.

Področje, kjer je prostor za izboljšave, je predvsem komunikacija z razvojno ploščo. Prenos podatkov po RS-232 standardu je počasen in zahteva neprimerno več časa kot samo procesiranje in dostopanje do pomnilnika na FPGA. Razvojna plošča Virtex-6 vsebuje dodaten USB priključek, ki bi lahko nadomestil serijsko povezavo po RS-232 standardu. V tem primeru bi bilo potrebno modul UART nadomestiti z ustreznim USB krmilnikom. Možnosti za izboljšave so tudi glede grafičnega vmesnika. Mogoče bi ga bilo razširiti tako, da bi z njim izbrali tip podatkov, ki se bodo procesirali, ta informacija pa bi se prenesla na FPGA, kjer bi se izvedlo ustrezno procesiranje.

Slike

Tabele

Literatura

[1] Wikipedia, "Field-programmable gate array", 2012. Dostopno na:
http://en.wikipedia.org/wiki/Field-programmable_gate_array

[2] N. Zimic, J. Virant, *Logično načrtovanje računalniških struktur in sistemov*, 1998, str. 123-125.

[3] Virtex-6 FPGA ML605 Evaluation Kit, Product Brief, 2009. Dostopno na:
http://www.xilinx.com/publications/prod_mktg/ml605_product_brief.pdf

[4] Xilinx, "FPGA", 2012. Dostopno na:
http://www.xilinx.com/fpga

[5] Virtex-6 Family Overview, Product Specification, 2012. Dostopno na:
http://www.xilinx.com/support/documentation/data_sheets/ds150.pdf

[6] Peter J. Ashenden, *The VHDL Cookbok, First Edition* 1990, pogl. 1-1. Dostopno na:
http://tams-www.informatik.uni-hamburg.de/vhdl/doc/cookbook/VHDL-Cookbook.pdf

[7] ML605 Hardware, User Guide, 2011. Dostopno na:
http://www.xilinx.com/support/documentation/boards_and_kits/ug534.pdf

[8] Wikipedia, "VHDL", 2012. Dostopno na:
http://en.wikipedia.org/wiki/VHDL

[9] The University of New Mexico, "Hardware Design with VHDL", *Design Example: UART*, 2008. Dostopno na:
http://www.ece.unm.edu/~jimp/vhdl_fpgas/slides/UART.pdf

[10] D. Kodek, *Arhitektura in organizacija računalniških sistemov*, Šenčur: Bi-Tim, 2008, pogl. 8, dodatek C.

[11] Hothardware, 2012, Dostopno na:
http://hothardware.com

[12] Wikipedia, "DDR SDRAM", 2012. Dostopno na:
http://en.wikipedia.org/wiki/DDR_SDRAM

[13] Wikipedia, "DDR3 SDRAM", 2012. Dostopno na:
http://en.wikipedia.org/wiki/DDR3_SDRAM

[14] DDR3 SDRAM SODIMM Features, "MT4JSF6464H – 512MB", Micron, 2007. Dostopno na:
http://www.datasheets.org.uk/indexdl/Datasheets-SW2/DSASW0020696.pdf

[15] 1Gb DDR3 SDRAM Features, "MT41J64M16 – 8 Meg x 16 x 8 banks", Micron, 2006. Dostopno na:
http://download.micron.com/pdf/datasheets/dram/ddr3/1Gb_DDR3_SDRAM.pdf

[16] Wikipedia, "Universal asynchronous receiver/transmitter", 2012. Dostopno na:
http://en.wikipedia.org/wiki/Universal_asynchronous_receiver/transmitter

[17] Patricio Bulić, *Načrtovanje in sinteza digitalnih in mikroprocesorskih sistemov v FPGA*, 2007, pogl. 5.11. Dostopno na:
http://ucilnica1011.fri.uni-lj.si/file.php/104/vhdl_zapiski.pdf

[18] Interfacing Micron DDR2 Memories to Xilinx Spartan-3A/AN FPGAs, A Step-By-Step Guide, 2008. Dostopno na:
http://edge.rit.edu/content/P10662/public/old/FPGA/InterfacingXilinxFPGAwithMicronDDR.pdf

[19] Wikipedia, "Flip-flop (electronics)", 2012. Dostopno na:
http://en.wikipedia.org/wiki/Flip-flop_(electronics)

[20] Virtex-6 FPGA Memory Interface Solutions, User Guide, 2011. Dostopno na:

http://www.xilinx.com/support/documentation/ip_
documentation/ug406.pdf

[21] ML605 MIG Design Creation, 2010. Dostopno na:
http://www.xilinx.com/support/documentation/boards_and_kits/
xtp047.pdf