Univerzitet u Beogradu
Elektrotehnički fakultet

Vladimir Petrović, 89/2010

Međuprocesorska komunikacija između MicroBlaze i ARM Cortex-A9 procesora na ZYNQ-7000 platformi

diplomski rad

mentor:
prof. dr Lazar Saranovac

Beograd, jun 2014.

Sažetak

U ovom radu je prikazana međuprocesorska komunikacija između MicroBlaze procesora, koji je instanciran u programabilnoj logici ZYNQ-7000 sistema na čipu, i Dual core ARM Cortex-A9 procesora iz ASIC dela čipa. Na ARM procesorima je pokrenut Linux operativni sistem. Na Linuxu je pokrenuta aplikacija koja posredstvom drajvera razmenjuje poruke sa MicroBlaze procesorom.

Ključne reči: Međuprocesorska komunikacija, programabilni sistem na čipu, MicroBlaze procesor, ARM Cortex-A9, Linux

Sadržaj

Glava 1

Uvod

Razvojna filozofija računarskih sistema sve više favorizuje paralelizam u obradi podataka, što kao posledicu ima razvoj velikog broja izuzetno kompleksnih računarskih sistema sa nekoliko procesora (procesorskih jezgara - *processor cores*) koja imaju za cilj da neki složen posao završe za malo vremena. U razvoju jednoprocesorskih sistema, danas se nailazi na ozbiljne prepreke i ograničenja. Npr. brzina rada memorija ograničava performanse procesora jer se dešava da se mnogo više vremena troši na upis i čitanje podataka nego na samu njihovu obradu. Takođe, sve su više izražena ograničenja u razvoju paralelizma na nivou instrukcija (ILP - *Instruction Level Parallelism*) jer je benefit od usložnjavanja hardvera za pajplajn sve manji. Povećanje učestanosti rada već vrlo složenih sistema bi izazvalo drastične zahteve za dodatnom potrošnjom energije. [1] Ovo su sve razlozi zbog kojih kompanije sve više forsiraju višeprocesorske sisteme za razne primene, počev od opšte-namenskih računara sa nekoliko procesorskih jezgara, preko namenskih računara različitih primena, pa do signalnih i grafičkih procesora koji neretko imaju par stotina (pa i hiljada) jezgara.

Potrošnja energije elektronskih uređaja postaje jedan od najzahtevnijih faktora u dizajnu. Zbog toga se sve više pojavljuju heterogeni višeprocesorski sistemi gde u jednom čipu postoji procesor velikih performansi ali i velike potrošnje, koji vrši sve složene poslove, i procesor manjih performansi ali i manje potrošnje koji preuzima kontrolu kada zahtevi nisu veliki. Neki od primera su *Sitara* procesor firme *Texas Instruments* i veliki broj procesora vrlo ozbiljnih performansi na bazi ARM-ove *big.LITTLE* arhitekture kao što su procesori iz *Qualcom Snapdragon* 808 i 810 i *Samsung Exynos* 5 *Octa* serija. [2]. U sistemima na čipu sa heterogenim višeprocesorskim sistemom, procesor manjih performansi često može raditi kao koprocesor ili kao procesor koji izvršava neke specifične operacije kao npr. kontrolu potrošnje celog sistema na čipu. [3]

Kod svih višeprocesorskih sistema postoje neki deljeni resursi. To je neizostavno memorija, ali deljeni resursi su vrlo često i periferije. Jasno je da zbog toga mogu postojati problemi istovremenog pristupa resursima ili, u sistemima kod kojih dva procesora imaju nezavisne keš memorije, problemi sa koherencijom keša (*cache coherence*). Zbog toga je neophodno da postoji dovoljno dobar sistem za sinhronizaciju pristupa memorijskim lokacijama kao i komunikaciju između procesora kada oni obavljaju poslove koji su međusobno zavisni.

Za sinhronizaciju pristupa resursima gotovo uvek mora da postoji neki protokol uzajamnog isključivanja (*mutual exclusion - mutex*). Implementacija ovog protokola zavisi od primene i od toga šta je sve implementirano u hardveru. Moguće je i da postoje periferije koje su zadužene za kontrolu pristupa nekim resursima (*mutex* periferije) koje znatno olakšavaju sinhronizaciju prilikom pisanja softvera. [4]

Komunikacija između dva procesora može da bude direktna ako postoji poseban interfejs od jednog ka drugom procesoru preko koga se razmenjuju informacije, ili, češće, indirektna gde postoji neki memorijski prostor rezervisan za razmene poruka između procesora. [5] Takođe,

radi brže i jednostavnije komunikacije, za razmene poruka između procesora se često koristi *mailbox* periferija. Ova periferija radi na principu generisanja prekida procesoru kada god se, od strane drugog procesora, u neku njenu memorijsku lokaciju upiše poruka namenjena tom procesoru. Prvi procesor prihvatanjem zahteva za prekid prepoznaje da mu je stigla poruka, čita odgovarajuću lokaciju iz *mailbox* periferije i obrađuje poruku koja je poslata. [6]

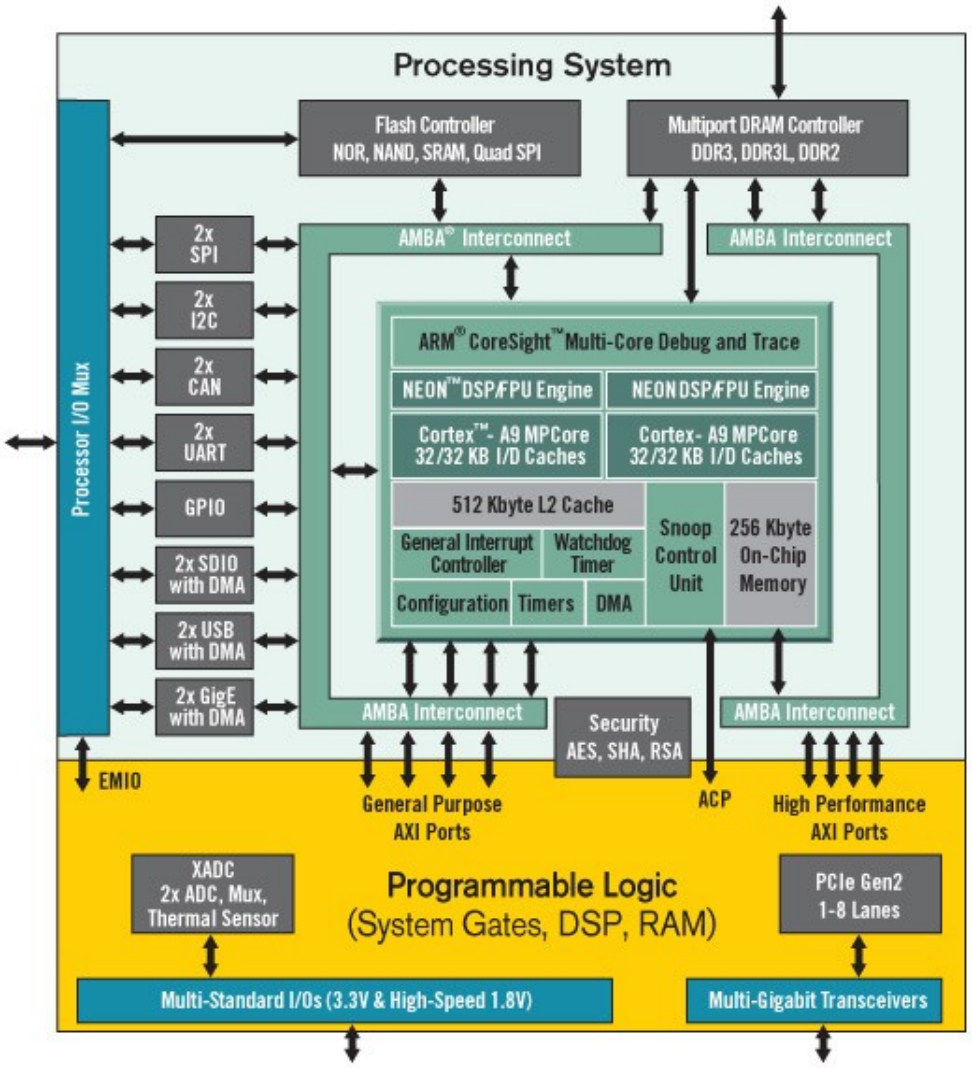
U ovom radu je implementirana slična komunikacija između dva različita procesora. Prvo je implementiran *MicroBlaze* procesor sa pratećim periferijama u hardverski programabilnom delu $Xilinx$-ovog $ZYNQ-7000$ sistema na čipu, a zatim je implementirana međuprocesorska komunikacija između *MicroBlaze* procesora i *Dual core ARM Cortex-A9* procesora koji se nalazi u ASIC delu čipa zajedno sa velikim brojem periferija i interfejsa i na kome je pokrenut *Linux* operativni sistem.

U glavi 2 ovog rada je opisana korišćena hardverska platforma. U glavi 3 je opisana realizacija hardvera u programabilnoj logici za potrebe međuprocesorske komunikacije, dok je u glavi 4 opisan softver u kome je implementirana ta komunikacija. Na kraju, u glavi 5, dati su rezultati i diskusija rezultata. Prilog radu su dva dodatka u kojima su dati programski kodovi (dodatak A) i uputstvo za korišćenje alata za dizajn hardvera i softvera, instaliranje Linux-a i pokretanje celog sistema (dodatak B).

Glava 2

Opis hardverske platforme

Za realizaciju ovog rada korišćena je ZC706 razvojna ploča za $Xilinx\ ZYNQ-7000$ programabilni sistem na čipu ($Xilinx\ Zynq-7000\ All\ Programmable\ SoC\ ZC706\ Evaluation\ Kit$). Na ploči se nalazi programabilni sistem na čipu XC7Z045-2FFG900C iz ZYNQ-7000 serije. Na slici 2.1 je prikazana blok šema ovog čipa.

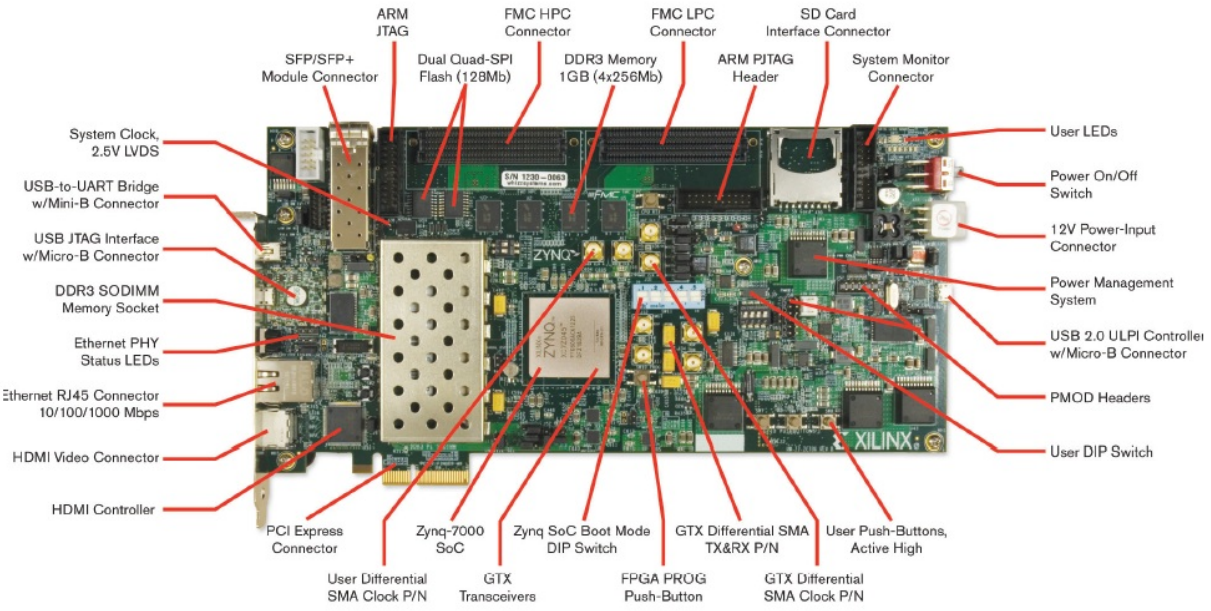


Slika 2.1: Blok šema ZYNQ-7000 ZC706 sistema na čipu

Na ASIC delu čipa (*PS - Processing System*) se nalazi $Dual\ core\ ARM\ Cortex-A9$ sa dodatnim NEON ekstenzijama za digitalnu obradu signala. Procesor raspolaže sa po 32kB L1

keš memorije za instrukcije i podatke po jezgru i 512kB zajedničke L2 keš memorije. Svako jezgro raspolaže sa po jednim 32-bitnim tajmerom i watchdog tajmerom, kao i zajedničkim 64-bitnim tajmerom. ZYNQ-7000 PS poseduje on-chip memoriju (*on-chip memory - OCM*) koja se sastoji od 256 KB RAM-a i 128 KB ROM-a (BootROM) koji se koristi prilikom pokretanja sistema. Prekidni kontroler je GIC pl390 prekidni kontroler čijim kontrolnim i statusnim registrima se pristupa preko nezavisne privatne magistrale. Interna magistrala u čipu je ARM AMBA 4.0 (*AMBA - Advanced Microcontroller Bus Architecture*) sa AXI4 interfejsom (*AXI - Advanced eXtensible Interface*). U ASIC delu čipa se nalaze još i DMA kontroler, DDR kontroler, fleš kontroler, po dve I^2C, SPI, CAN i UART periferije, jedna GPIO periferija, kao i dva ethernet kontrolera i dva SD kontrolera. Omogućeno je i debagovanje preko standardnog JTAG (IEEE 1149.1) interfejsa. PS (*Processing System*) ima AXI4 interfejs ka FPGA delu čipa (*PL - Programmable Logic*), tako da je omogućen pristup svim periferijama instanciranim u FPGA delu koje podržavaju AXI interfejs. [7]

Na slici 2.2 je prikazana ploča sa obeleženim najvažnijim delovima. Za korišćenje Linux operativnog sistema koji je pokrenut na ploči korišćen je UART1 iz ASIC dela ZYNQ-7000 čipa koji je povezan sa USB na UART konvertorom na ploči. Ethernet interfejs je korišćen za prenos fajlova sa host računara na ploču. Ploča ima DIP prekidač kojim se podešava način pokretanja sistema: čitanjem iz fleš memorije, čitanjem sa SD kartice ili preko JTAG interfejsa. U ovom radu je korišćeno butovanje sa SD kartice. Na ploči postoji i USB JTAG interfejs preko *Digilent* modula kojim se omogućava programiranje programabilne logike i debagovanje aplikacija. Za testiranje su korišćene i LE diode i korisnički tasteri na ploči. U programabilnoj logici čipa je instanciran MicroBlaze, pa je za njegov ispis korišćen IP UART-a koji je povezan na PMOD interfejs na ploči preko koga se mogu povezati periferije čiji signali nisu već izvedeni na pinove čipa, već je ostavljeno korisniku da odabere koje će periferije koristiti (SPI, CAN, drugi UART koji nije izveden na USB na UART adapter itd.). [8]



Slika 2.2: Razvojna ploča za ZYNQ-7000 ZC706 sistem na čipu

Glava 3

Implementacija potrebnog hardvera u FPGA

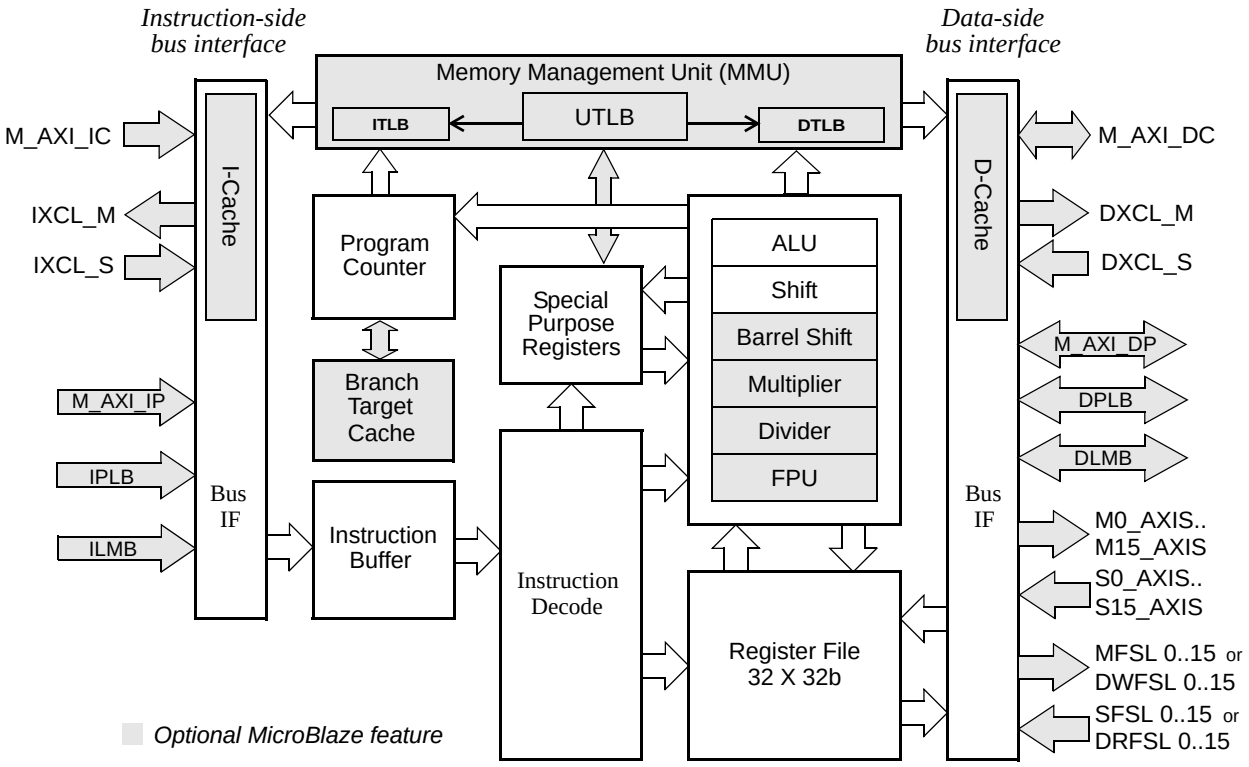
U ovoj glavi će biti opisan sistem koji je realizovan za potrebe međuprocesorske komunikacije između MicroBlaze procesora i ARM Cortex-A9 procesora. Za sve potrebne funkcionalnosti i periferije postoje gotovi IP (*IP - Intellectual Property*) blokovi koje je razvio Xilinx i koji su dostupni u alatima za dizajn i sintezu hardvera. Za dizajn, sintezu i implementaciju korišćen je Vivado Design Suite 2013.4. Najpre će ukratko biti opisan MicroBlaze soft core procesor, zatim i nekoliko ključnih periferija u dizajnu, a na kraju opis celog sistema.

3.1 MicroBlaze soft core procesor

MicroBlaze™ soft core procesor spada u grupu RISC (*RISC - Reduced Instruction Set Computer*) procesora i optimizovan je za implementaciju u FPGA čipovima kompanije Xilinx. Procesor je dizajniran tako da je vrlo konfigurabilan tako da se podešavanjem IP bloka mogu odabrati tačno određene funkcionalnosti koje su potrebne. Na slici 3.1 prikazana je blok šema MicroBlaze procesora preuzeta iz specifikacije. Sve sivo osenčene funkcionalnosti su opcione. Zajedničko za sve konfiguracije je da procesor poseduje 32 32-bitna registra opšte namene. Reč instrukcije je 32-bitna sa 3 operanda i dva adresna moda. Adresna magistrala je 32-bitna i procesor poseduje pajplajn (*single issue pipeline*). Pajplajn može biti podešen da radi sa 3 ili 5 nivoa izvršavanja instrukcije u zavisnosti od toga da li se žele postići veće performanse nauštrb površine zauzete na čipu. MicroBlaze može da radi i u *big endian* i u *little endian* formatu, zavisno od toga kako je podešen.

Verzije MicroBlaze procesora koje su podržane u Xilinx FPGA čipovima serije 6 i 7 podržavaju AXI4 interfejs. Za pristup glavnoj memoriji (npr. blok RAM memoriji iz FPGA) MicroBlaze poseduje posebnu LMB magistralu (*LMB - Local Memory Bus*). Takođe, ako je korisniku potrebno da instancirana neke koprocesore u hardveru, MicroBlaze podržava FSL interfejs (*FSL - Fast Simplex Link*) za komunikaciju sa ovim komponentama. Često je da koprocesori vrše neke specifične kompleksne operacije (npr. obradu signala ili druga izračunavanja), pa je korisno imati poseban interfejs za pristup rezultatima njihovog rada.

MicroBlaze može biti konfigurisan da poseduje jedinicu za upravljanje memorijom (*MMU - Memory Management Unit*) što omogućava pokretanje operativnih sistema koji zahtevaju hardversko dohvatanje stranica i zaštitu memorije (npr. *Linux*). [9]



Slika 3.1: Blok šema MicroBlaze procesorskog jezgra

3.2 Magistrala i periferije

U realizovanom sistemu su za pristup memorijama i memorijski mapiranim periferijama korišćene dve magistrale čije interfejse podržava MicroBlaze. Lokalna magistrala (*LMB - Local Memory Bus*) je korišćena za pristup memoriji za intrukcije i podatke, kao i pristup periferiji *I/O Module* o kojoj će kasnije biti reči. Za sve ostale periferije u programabilnoj logici, ali i ASIC delu čipa korišćen je AXI4 interfejs. U ovom odelju će prvo ukratko biti objašnjen ovaj interfejs i način povezivanja periferija preko AXI interkonekcijskog bloka, a zatim će biti opisane i neke važne periferije.

3.2.1 AXI4 interfejs i interkonekcijski blokovi

AXI je interfejs ARM AMBA magistrale. Prva verzija AXI interfejsa se pojavila 2003. godine sa AMBA 3.0 magistralom, dok je druga verzija, AXI4, predstavljena sa AMBA 4.0 magistralom 2010. godine.

AXI predstavlja interfejs između jednog master i jednog slejv uređaja koji međusobno razmenjuju informacije. Više uređaja može biti međosobno povezano korišćenjem tzv. interkonekcijskih blokova. Xilinx pruža gotov IP blok *AXI Interconnect IP* koji omogućava povezivanje više master uređaja sa više slejv uređaja. Magistrala je pun dupleks magistrala, a to omogućava pet različitih kanala od kojih se sastoji AXI4 interfejs:

- Kanal za adresu čitanja (*Read Address Channel - AXI-AR*)
- Kanal za adresu upisa (*Write Address Channel - AXI-AW*)
- Kanal za podatak koji se čita (*Read Data Channel - AXI-R*)
- Kanal za podatak koji se upisuje (*Write Data Channel AXI-W*)

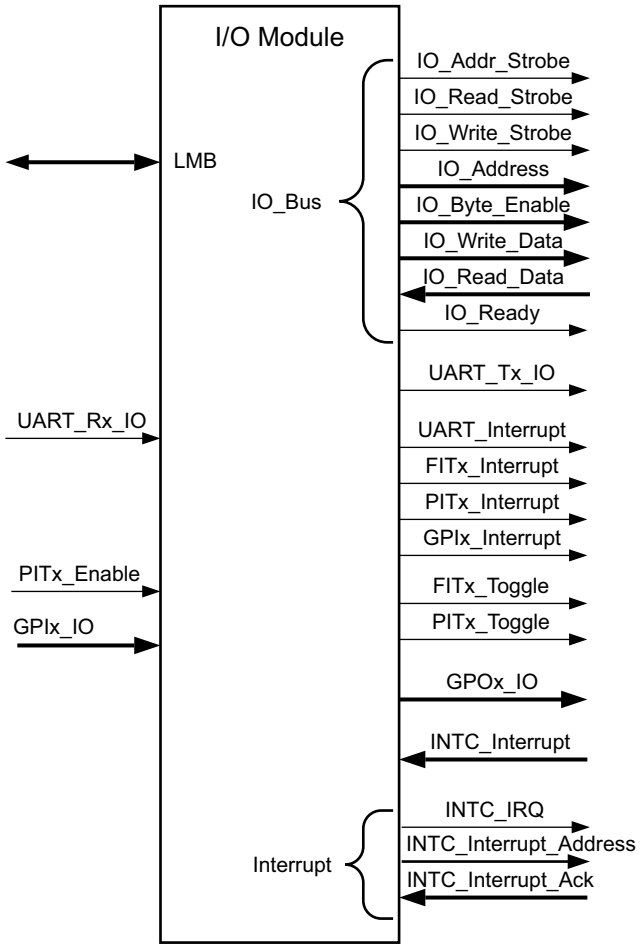
- Kanal za odgovor o statusu upisa (*Write Response Channel AXI-B*)

Burst režim rada je ograničen na maksimalno 256 transakcija. [10], [11]

3.2.2 Ulazno izlazni modul (*I/O module*)

Ulazno izlazni modul (*I/O module IP*) je periferija koja podržava određeni set standardnih ulazno izlaznih funkcija i namenjena je sistemima sa MicroBlaze procesorom. Registrima ove periferije se pristupa preko LMB (*Local Memory Bus*) magistrale. Na slici 3.2 je prikazan blok dijagram sa karakterističnim signalima ovog IP bloka preuzeta iz specifikacije.

Periferija poseduje nezavisnu magistralu za pristup eksternim uređajima (*I/O bus*) koja nije korišćena u ovom radu, UART periferiju, programabilne interval tajmere (*PIT*), fiksne interval tajmere (*FIT*), GPIO portove i prekidni kontroler koji služi za registrovanje prekida izazvanih od internih periferija ulazno izlaznog modula, ali i eksternih prekida kojih najviše može biti 16. Svaki GPI (ulazni port) može izazvati prekid u prekidnom kontroleru kada god se desi promena ulaznog signala. Prekidni kontroler pored eksternih i GPI prekida, prihvata još i prekide tajmera i UART-a iz ulazno izlaznog modula. Prekidi su vektorski, a kada se desi prekid, prekidni kontroler šalje zahtev za prekid MicroBlaze procesoru i adresu prekidnog vektora preko posebnog interfejsa za prekide označenog na slici 3.2 sa *Interrupt*. Zahtev se šalje preko *INTC_IRQ* linije, adresa prekidnog vektora preko *INTC_Interrupt_Address* linija, a signali o prihvatanju prekindog zahteva od strane procesora stižu preko *INTC_Interrupt_Ack* linija. [12]

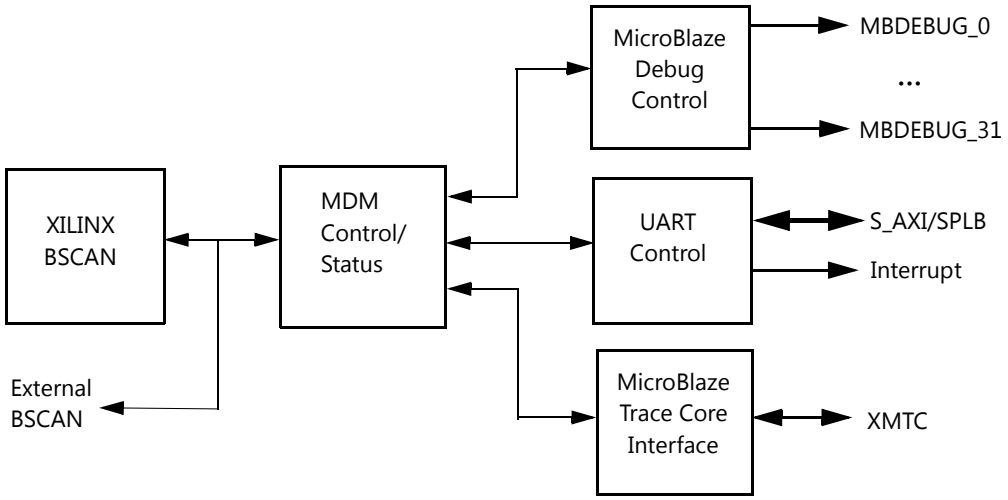


Slika 3.2: Blok dijagram ulazno izlaznog modula (*I/O module*)

3.2.3 *MicroBlaze debug* modul (*MDM*)

MicroBlaze debug module je IP blok koji omogućava JTAG debagovanje softvera pokrenutog na MicroBlaze sistemu. Na slici 3.3 prikazana je blok šema ovog IP bloka. MDM instancira Boundary-Scan (BSCAN) blok ili koristi eksterni BSCAN koji omogućava debagovanje. Modul je povezan sa MicroBlaze procesorom preko MBDEBUG interfejsa preko koga se kontroliše izvršavanje programa i čitaju potrebni podaci za prikaz u softveru za debagovanje. Jedan takav softver je *Xilinx Microprocessor Debugger (XMD)* koji omogućava debagovanje iz komandne linije.

MDM ima i UART čiji RX i TX signali se prenose preko FPGA JTAG porta preko koga se ostvaruje komunikacija sa XMD softverom. MicroBlaze može da pristupa ovom UART-u preko AXI ili PLB interfejsa. Sistem se može podesiti da MicroBlaze koristi UART iz debug modula umesto nekog drugog UART-a što može olakšati proces debagovanja. [13]



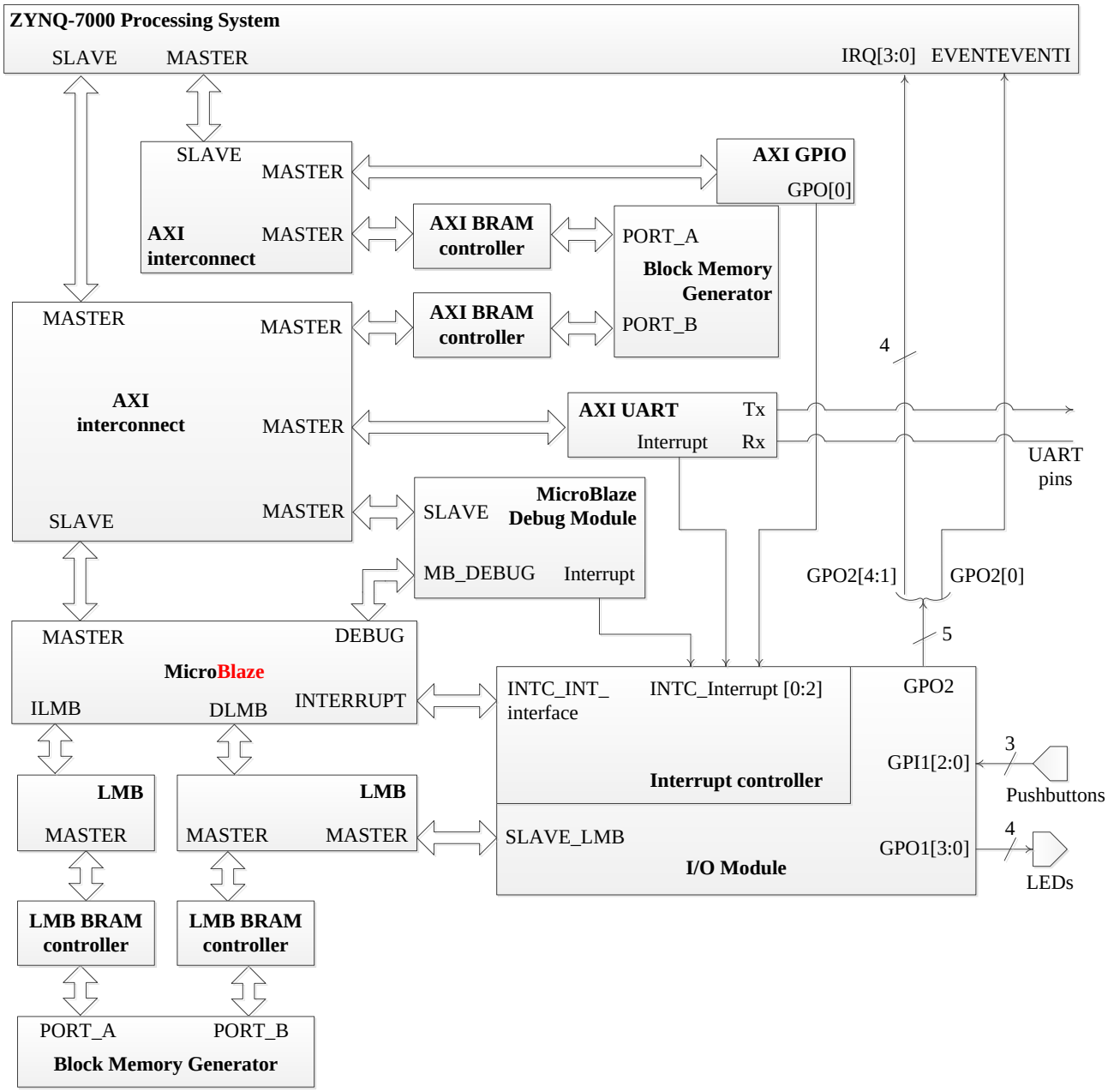
Slika 3.3: Blok šema *MicroBlaze debug* modula

3.3 Opis realizovanog sistema

Na slici 3.4 je prikazana blok šema hardvera koji je implementiran na ZYNQ-7000 čipu. MicroBlaze pristupa programu preko ILMB (*Instruction Local Memory Bus*) interfejsa a memoriji za podatke preko DLMB (*Data Local Memory Bus*) interfejsa. Programska memorija i memorija za podatke MicroBlaze procesora su realizovane u blok RAM sekcijama FPGA dela čipa. Za to je korišćen IP blok *Block Memory Generator* koji realizuje dvoportnu RAM memoriju. Ovoj memoriji se pristupa preko BRAM kontrolera koji su prilagođeni LMB interfejsu (*LMB BRAM Controller*). Preko LMB magistrale MicroBlaze pristupa i registrima ulazno izlaznog modula (*I/O module* na slici). Kako bi se omogućilo debagovanje instanciran je i IP blok MicroBlaze debug modula kao što je to prikazano na slici.

Kako bi MicroBlaze mogao da pristupa periferijama koje podržavaju AXI interfejs, instanciran je IP blok AXI interkonekcije (*AXI Interconect*) preko koga MicroBlaze može da pristupi određenim periferijama implementiranim u ASIC delu čipa (*ZYNQ-7000 Processing System*), AXI UART periferiji (IP blok *AXI Uartlite*) preko koje se vrši ispis na terminal PC računara, UART-u iz MDM-a, kao i deljenoj memoriji koja služi za razmenu poruka sa ARM procesorima. Ova deljena memorija je, kao i glavna memorija MicroBlaze procesora, realizovana u blok RAM sekcijama FPGA dela čipa i pristupa joj se uz pomoć AXI blok RAM kontrolera (AXI BRAM

Controller). Deljenoj memoriji na isti način imaju pristup i ARM Cortex-A9 procesori iz PS-a preko svog AXI interkonekcijskog bloka.



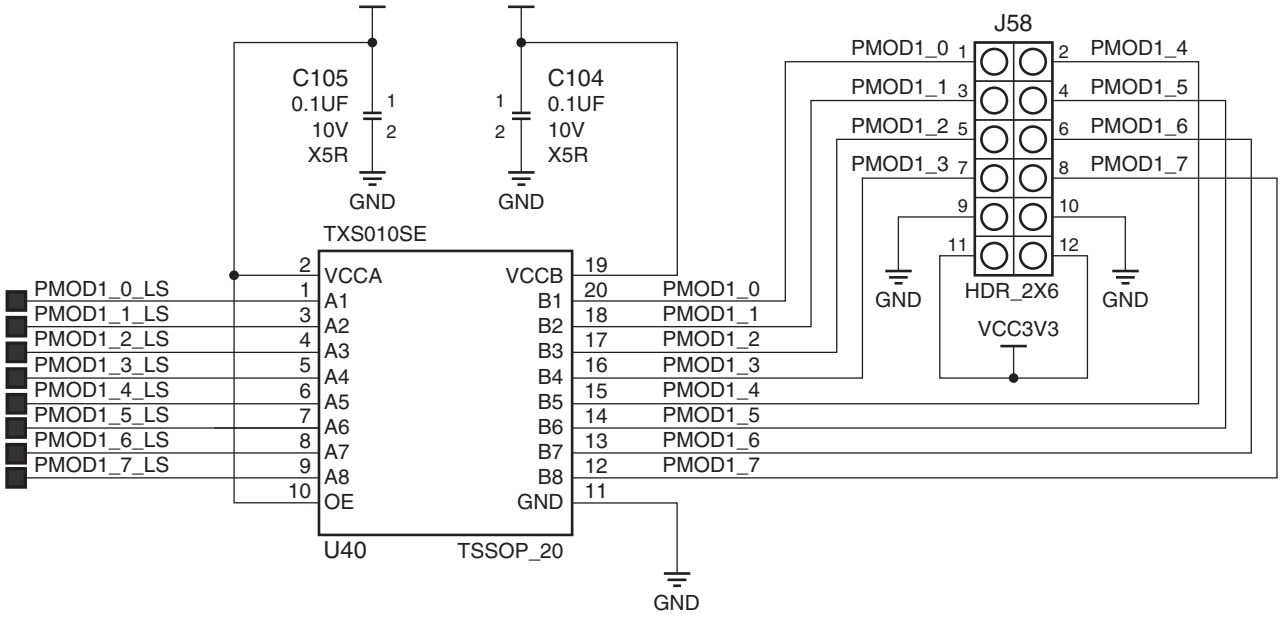
Slika 3.4: Blok šema realizovanog hardvera u programabilnoj logici ZYNQ-7000 čipa

Slanje poruke od MicroBlaze-a ka ARM procesorima i obrnuto funkcioniše tako što jedan procesor upiše podatak koji predstavlja poruku u deljenu blok RAM memoriju, a zatim pošalje prekid drugom procesoru. Drugi procesor čita tu poruku u prekidnoj rutini. Kako bi se omogućilo generisanje prekida od ARM procesora ka MicroBlaze procesoru, instanciran je IP blok *AXI GPIO* konfigurisan da radi samo kao izlazni port. Signal najnižeg bita GPO porta se vodi kao eksterni prekid na ulaz prekidnog kontrolera ulazno izlaznog modula MicroBlaze procesora. Postavljanjem logičke jedinice na ovaj port, ARM aktivira zahtev za prekid. Za generisanje prekida od strane MicroBlaze procesora, iskorišćen je GPO2 port ulazno izlaznog modula. Najniži bit ovog porta je povezan sa *EVENTEVENTI* signalom PS-a. Ovaj signal je jedan od signala *Event* interfejsa i služi da se njegovim generisanjem probudi jedan ili oba ARM Cortex-A9 procesora koji su prethodno bili u *standby* stanju iniciranom WFE instrukcijom (*WFE - Wait For Event*). [7] Signali GPO[4:1] su dovedeni na IRQ[3:0] ulaze

PS-a. Trigerovanjem nekog od ova četiri signala, generisaće se prekid u prekidnom kontroleru PS-a i u zavisnosti od toga da li je dozvoljen, ARM procesori će uraditi neku akciju ili ne. Ovim je omogućeno da oba procesora mogu da pošalju prekidni zahtev drugom procesoru i upišu poruku u deljenu memoriju što omogućava emulaciju mejlboks mehanizma indirektne komunikacije između procesora.

Za testiranje komunikacije između procesora korišćeni su i korisnički tasteri i LE diode na ploči. Oni su povezani na GPIO1 port *I/O Module* bloka. GPI1 port je podešen da postoji mogućnost generisanja prekida u slučaju pritiska tastera na ploči.

Kako je na ARM procesorima pokrenut Linux operativni sistem, za korišćenje korisničkog interfejsa operativnog sistema iskorišćen je UART1 u ASIC delu čipa. Ovaj UART je, kao što je već rečeno, povezan sa USB na UART konvertorom preko koga se ostvaruje komunikacija sa računarom. Da bi se obezbedio odvojen ispis koji generiše MicroBlaze u programabilnoj logici je instanciran UART (IP blok *AXI Uartlite*). Međutim, na ploči postoji samo jedan, već iskorišćen, konvertor USB na UART za komunikaciju sa računarom, pa su signali RX i TX UART-a iz programabilne logike dovedeni na PMOD interfejs (prikazan na slici 3.5) koji služi da se periferije koje nemaju drugu konekciju povežu sa nekim eksternim uređajem. Za povezivanje se koriste pinovi 1 (za Rx signal) i 3 (za Tx signal) J58 konektora. Ovi pinovi su povezani sa računarom drugim USB na UART konvertorom, čime je omogućen nezavistan ispis koji generiše MicroBlaze i koji generiše Linux koji se izvršava na ARM procesorima.



Slika 3.5: Šema PMOD GPIO interfejsa na ploči

Tokom dizajna u Vivado Design Suite programu mogu se izvršiti i određena podešavanja ASIC dela čipa (PS). Ta podešavanja se odnose na to koje periferije će biti korišćene, koji sve interfejsi postoje prema programabilnoj logici i na remapiranje nekih pinova. U ZYNQ-7000 PS-u su uključene sledeće **periferije**: UART_1, Ethernet_0, USB_0, GPIO, SD_0, I^2C_0 i QSPI interfejs za fleš kontroler. Takođe, podešavanja se odnose i na to koje periferije su dostupne iz programabilne logike preko AXI interfejsa. Tako, MicroBlaze vidi samo određene periferije iz PS-a.

U tabeli 3.1 su prikazane adrese periferija i memorija koje koristi MicroBlaze procesor. Adrese periferija koje su dostupne i MicroBlaze procesoru i ARM procesorima su podešene da budu iste radi konzistentnosti. Zbog toga se DDR kontroler iz PS-a nalazi na adresi 0x0000000. Podrazumevana vrednost resetnog vektora MicroBlaze procesora je 0x00000000, ali je zbog DDR kontrolera, resetni vektor podešen na 0xC0000000. Zbog toga će MicroBlaze po resetu

prvo izvršiti instrukciju skoka na adresu 0xC0000000, a zatim nastaviti da izvršava kod koji se nalazi u blok RAM memoriji.

Tabela 3.1: Memorijski prostor MicroBlaze procesora

Periferija/memorija	**Startna adresa**	**Krajnja adresa**
Glavna BRAM memorija	0xC0000000	0xC001FFFF
I/O module	0xC4A00000	0xC4A10FFF
MicroBlaze debug module	0x41400000	0x41400FFF
AXI Uartlite	0x40600000	0x4060FFFF
Deljena BRAM memorija	0x40000000	0x40000FFF
PS UART1	0xE0001000	0xE0001FFF

FPGA deo čipa se programira odmah nakon uključenja napajanja što obavlja FSBL (*First Stage Boot Loader*) program koji se izvršava na jednom od ARM Cortex A-9 procesora. O ovome će biti reči u narednom poglavlju u kome se opisuje softver.

Glava 4

Softver

U prethodnom poglavlju je opisan hardver koji je implementiran na ZYNQ-7000 platformi za potrebe međuprocesorke komunikacije između MicroBlaze procesora i ARM Cortex-A9 procesora. U ovom poglavlju je opisan softver u kome je implementirana ta komunikacija koristeći implementirani hardver. Za razvoj softvera za MicroBlaze i generisanje inicijalizacionih programa za ceo sistem, kao i za debagovanje koda koji izvršava MicroBlaze, korišćen je Xilinx SDK (Software Development Kit) 2013.4.

MicroBlaze izvršava program koji čita iz svoje blok RAM memorije za instrukcije. Program je standalone aplikacija dok se na ARM-ovim procesorima izvršava Linux. Za komunikaciju sa MicroBlaze procesorom, napisan je Linux drajver iz koga se može pristupiti deljenoj memoriji za komunikaciju i u kom je opisano kako se obrađuju prekidi koje generiše MicroBlaze. Drajver komunicira sa aplikacijom u Linux-u. U aplikaciji se obrađuju poruke koje stignu od MicroBlaze procesora i na osnovu njih se odlučuje koja poruka se šalje MicroBlaze procesoru.

U ovom poglavlju je najpre objašnjen postupak inicijalizacije sistema i pokretanja operativnog sistema. Zatim je opisan protokol komunikacije između procesora i na kraju sama implementacija aplikacije za MicroBlaze, drajvera za komunikaciju i aplikacije u Linux-u.

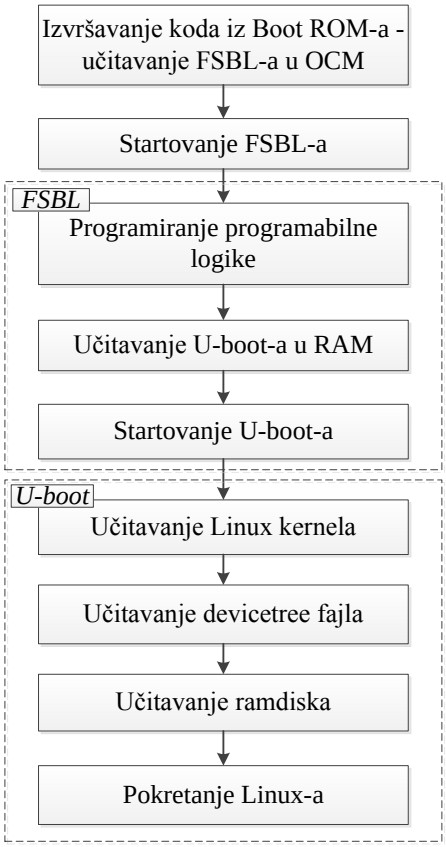
4.1 Inicijalizacija i pokretanje sistema

Na MicroBlaze procesoru se izvršava standalone aplikacija, dakle bez operativnog sistema. Kao što je opisano u prethodnom poglavlju, program koji izvršava MicroBlaze se nalazi u blok RAM memoriji FPGA čipa. Ni jedan drugi sistem ne može da pristupi programskoj memoriji MicroBlaze procesora, što znači da učitavanje programa mora da se obavi prilikom programiranja FPGA. Za kreiranje bitstream fajla koji se spušta u FPGA čip i koji ima inicijalizovanu programsku memoriju korišćen je Xilinx-ov program $Data2MEM$. [14] Ovaj program koristi bitstream fajl (.bit) generisan u softveru za implementaciju hardvera (Vivado Design Suite) i menja sadržaj blok ram memorije programom iz *.elf* izvršnog fajla. Nakon programiranja FPGA novim inicijalizovanim bitstream fajlom, MicroBlaze kreće sa izvršavanjem programa.

Na ARM procesorima je pokrenut Linux operativni sistem. Xilinx je razvio svoju distribuciju Linux-a *PetaLinux* koja je temeljno testirana i konfigurisana za rad na Xilinx-ovim čipovima. PetaLinux u ovom radu je baziran na Linux kernelu 3.12. Proces podizanja Linux-a i programiranja FPGA prikazan je na slici 4.1.

Po uključenju napajanja i otpuštanju reseta, inicijalizacija sistema na ZYNQ čipu započinje izvršavanjem koda iz Boot ROM-a. Program iz Boot ROM-a inicijalizuje jedan od ARM procesora i potrebne periferije za početak dohvatanja prvog loader programa, *First Stage Boot Loader*-a (*FSBL*), iz neke memorije, npr. fleš memorije ili sa SD kartice u zavisnosti od toga kako su podešeni prekidači na ploči. U ovom radu je korišćeno učitavanje sa SD kartice. Program iz Boot ROM-a učitava FSBL u on-chip memoriju (OCM) u okviru PS-a. FSBL program se kreira

na osnovu hardverske specifikacije koja se generiše u programima za implementaciju hardvera. Za kreiranje ovog loader programa je korišćen Xilinx SDK 2013.4.



Slika 4.1: Tok inicijalizacije sistema i pokretanja Linux-a

FSBL program najpre vrši inicijalizaciju PS-a, a zatim programira programabilnu logiku na čipu ako postoji bitstream fajl za njeno programiranje. Nakon programiranja FPGA, FSBL učitava drugi loader program (*second stage boot loader*) ili standalone aplikaciju u DDR memoriju. U slučaju pokretanja Linux-a, učitava se *u-boot* program koji predstavlja boot loader za Linux. Na kraju FSBL predaje kontrolu učitanom programu iz DDR memorije.

U-boot učitava Linux kernel (uImage) u DDR memoriju, učitava device tree i na kraju ramdisk za root fajl sistem Linux-a. Nakon učitavanja svih potrebnih fajlova u-boot predaje kontrolu Linux kernelu kada se i pokreće Linux operativni sistem. [15] U dodatku B ovog rada je opisan detaljan postupak generisanja FSBL-a, bitstream fajla za programiranje FPGA, kompajliranja u-boot-a i Linux kernela i generisanja device tree (.dtb) fajla uz pomoć Xilinx-ovih alata Vivado Design Suite, Xilinx Software Development Kit i data2MEM.

4.2 Protokol komunikacije

Kao što je već opisano u glavi 3, razmena poruka između procesora funkcioiše tako što prvo jedan procesor upiše podatak u deljenu blok RAM memoriju, a zatim pošalje prekid drugom procesoru čime ga obaveštava o tome da je poruka spremna za čitanje. Drugi procesor tokom obrade prekida, čita poruku iz deljene memorije i tu poruku obrađuje.

U realizovanom sistemu protokol komunikacije je takav da kada se desi neki događaj koji registruje MicroBlaze (npr. pritisak tastera na ploči), taj procesor šalje prvu poruku ARM procesoru koja nosi informaciju o tome koji se događaj desio. Aplikacija koja je pokrenuta na Linuxu stalno komunicira sa drajverom i proverava da li je neka poruka stigla. Kada se u okviru drajvera pročita poruka, na prvi sledeći zahtev aplikacije ta poruka se prosleđuje aplikaciji. U aplikaciji se na osnovu primljene poruke donosi odluka koja akcija odgovara primljenoj poruci, a zatim se preko drajvera šalje poruka MicroBlaze-u i očekuje se odgovor o statusu da li je akcija uspešno obavljena. MicroBlaze, po prijemu komandne poruke, izvršava akciju koja je naznačena i šalje poruku uspešnog statusa, a ukoliko nije mogao da protumači poruku ili se desila neka druga greška šalje poruku o grešci.

U narednom odeljku je opisan jednostavan primer međuprocesorske komunikacije u kom su za demonstraciju korišćeni tasteri i LE diode na razvojnoj ploči.

4.3 Realizacija međuprocesorske komunikacije

Realizovan je primer u kome pritisak nekog od korisničkih tastera aktivira zahtev za prekid GPI1 porta I/O modula u kome je i prekidni kontroler za MicroBlaze procesor. U prekidnoj rutini se određuje koji od tri tastera je pritisnut i šalje se poruka ARM procesorima tako što se prvo upiše podatak na ofset 0x00000000 u deljenoj BRAM memoriji, čija je osnovna adresa 0x4000000, a zatim se pošalje prekid na liniji IRQ3 sa slike 3.4. Ovaj prekid je povezan na ulaz 91 prekidnog kontrolera u ASIC delu čipa. U tabeli 4.1 prikazane su sve poruke koje MicroBlaze šalje ARM procesorima. DONE_ACK i ERROR_ACK poruke se šalju kao odgovor na zahtevanu akciju od strane ARM procesora.

Tabela 4.1: Poruke koje MicroBlaze procesor šalje ARM procesorima

Poruka	**Kod poruke**	**Značenje**
TAST1_PRESSED	0x1111	Pritisnut taster 1
TAST2_PRESSED	0x2222	Pritisnut taster 2
TAST3_PRESSED	0x3333	Pritisnut taster 3
DONE_ACK	0x5555	Akcija uspešno obavljena
ERROR_ACK	0xAAAA	Poruka nije prepoznata ili je došlo do druge greške

Za pisanje, kompajliranje i linkovanje aplikacije za MicroBlaze korišćen je *Xilix Software Development Kit* 2013.4 koji za kompajliranje i linkovanje aplikacije koristi *arm-xilinx-linux-gnueabi* toolchain. Xilix SDK sam generiše makefile za aplikaciju na osnovu parametara i podešavanja u svom grafičkom interfejsu.

Za komunikaciju sa MicroBlaze procesorom napisan je drajver koji prosleđuje poruke koje MicroBlaze šalje Linux aplikaciji i nazad. Realizovan je character device drajver sa kojim aplikacija komunicira uz pomoć *ioctl*[1] funkcije. *ioctl* funkcija može da primi tri komande prilikom poziva iz aplikacije: SEND_COMMAND, RECEIVE_STATUS i RECEIVE_ACTION. Pozivom *ioctl* funkcije sa komandom SEND_COMMAND, drajver šalje poruku specificiranu trećim argumentom *ioctl* funkcije (prvi je pokazivač na device fajl koji je potrebno otvoriti iz aplikacije da bi se pristupilo drajveru, a koji se kreira prilikom instaliranja drajvera). Pozivom sa komandom RECEIVE_ACTION, *ioctl* upisuje na adresu trećeg argumenta podatak koji predstavlja poruku koju je drajver primio od MicroBlaze procesora. Ako trenutno nije primljena poruka, na adresu trećeg argumenta se upisuje odgovarajući podatak preko koga aplikacija zna da još uvek nije stigla nova poruka. Poziv *ioctl* funkcije sa komandom RECEIVE_STATUS je isti kao i poziv sa RECEIVE_ACTION komandom, ali je kreirana nova komanda radi razlikovanja očekivanih poruka.

Drajver prima poruku od MicroBlaze procesora u callback funkciji koja se poziva prilikom obrade zahteva za prekid, smešta je u lokalnu promenljivu i na zahtev aplikacije preko poziva *ioctl* funkcije prosleđuje poruku aplikaciji.

Drajver šalje poruku upisom podatka u deljenu memoriju na ofset 0x00000008 i postavljanjem logičke jedinice na izlaz AXI GPIO porta koji se nalazi na adresi 0x41200000. Prekid je aktivan sa nivoom, što znači da će MicroBlaze imati višestruki zahtev za prekid ukoliko drajver ne postavi nulu na AXI GPIO port pre izlaska programa MicroBlaze procesora iz prekidne rutine. Zbog toga se u prekidnoj rutini prekida koji ARM procesor generiše MicroBlaze procesoru, prvo pošalje drugi zahtev za prekid ARM procesoru na liniji IRQ2, što je u prekidnom kontroleru ASIC dela čipa ulaz 90. Ovaj zahtev za prekid je zapravo acknowledge signal da je prekid koji je poslao drajver prihvaćen. U callback funkciji koja se poziva prilikom obrade ovog prekida koji emulira acknowledge signal, postavlja se logička nula na izlaz AXI GPIO porta čime je završeno slanje poruke.

[1] potpis *ioctl* funkcije je: `int driver_ioctl(struct file *f, unsigned int cmd, unsigned long arg)`

Aplikacija koja je napisana za demonstraciju međuprocesorske komunikacije u *while* petlji stalno poziva *ioctl* funkciju sa komandom RECEIVE_ACTION. Kada se konačno desi da je poruka primljena, u aplikaciji se na osnovu toga koja je poruka primljena odlučuje koja će se komandna poruka poslati MicroBlaze procesoru. Poruka koja se šalje se sastoji iz dva bajta kao što je to prikazano u tabeli 4.2.

Tabela 4.2: Format poruke koju ARM procesor šalje MicroBlaze procesoru

15 — 8	7 — 0
LEDx	AKCIJA

Prvi bajt označava diodu nad kojom treba da se izvrši neka akcija (LED0, LED1, LED2 ili LED3), a drugi bajt akciju koja treba da bude izvršena (ON, OFF, TOGGLE). U tabeli 4.3 prikazani su kodovi svih bajtova od kojih se kreira poruka za slanje. Tako je npr. kod poruke koja treba da izazove isključenje diode LED3 0x1322.

Tabela 4.3: Podaci od kojih se formira poruka koju ARM procesori šalju MicroBlaze procesoru

Oznaka	**Kod**	**Značenje**
LED0	0x10	Akcija se odnosi na LED0
LED1	0x11	Akcija se odnosi na LED1
LED2	0x12	Akcija se odnosi na LED2
LED3	0x13	Akcija se odnosi na LED3
ON	0x11	Akcija uključuje specificiranu diodu
OFF	0x22	Akcija isključuje specificiranu diodu
TOGGLE	0x33	Akcija menja stanje specificirane diode

Na kraju ciklusa, aplikacija očekuje potvrdu statusa od MicroBlaze procesora, pa u novoj *while* petlji, stalnim pozivanjem *ioctl* funkcije, čeka dok ne dobije status uspešnosti akcije u vidu statusne poruke DONE_ACK ili ERROR_ACK.

Za kompajliranje i linkovanje drajvera i aplikacije korišćen je *arm-none-linux-gnueabi* toolchain (gcc kompajler i linker).

Glava 5

Rezultati i diskusija

U ovoj glavi su opisani rezultati rada, tačnije data je analiza ispisa koje daju MicroBlaze procesor i Linux operativni sistem preko serijske veze sa računarom, jer je to jedini način praćenja statusa poslatih i primljenih poruka. Na PC računaru je pokrenuta Ubuntu 12.04 distribucija Linux-a, a za komunikaciju sa serijskim portom korišćen je program `minicom`.

Kako bi se omogućila komunikacija potrebno je prvo učitati drajver. Instaliranje drajvera automatski kreira device fajl i dozvoljava prekide na ulazima 90 i 91 prekidnog kontrolera. U listingu 5.1 prikazano je instaliranje drajvera iz komandne linije Linux operativnog sistema naredbom `insmod`, a zatim i provera da li je kreiran device fajl `mb_communication_driver` u `/dev` direktorijumu. Prvi znak u ispisku `ls -l` komande (znak `c`) označava da je device fajl kreiran kao *character device*. Zatim su prikazane dozvole čitanja i upisa za različite nivoe pristupa (`rw-rw--` znači da vlasnik (*owner*) i određena grupa korisnika (*group*) što je u ovom slučaju `root` nalog u Linux-u, imaju pravo čitanja i upisa u device fajl). Broj 247 u ispisu predstavlja glavni broj drajvera (*major number*). Ovaj broj identifikuje drajver koji upravlja uređajem. Jedan drajver može kreirati više character device fajlova preko kojih komunicira sa različitim uređajima. Svaki od tih uređaja je identifikovan od strane kernela korišćenjem drugog broja - *minor number*, koji je prikazan iza zareza posle *major* broja (u ovom slučaju *minor* broj je 1). Na kraju ispisa je prikazano ime preko koga se pristupa *character device* fajlu. [16]

Listing 5.1: Prikaz provere da li je kreiran device fajl

```
1 root@zynq:~# insmod /mb_communication_driver.ko
2 MicroBlaze communication driver
3 root@zynq:~# ls -l /dev/mb_communication_driver
4 crw-rw---- 1 root root 247, 1 Jan 1 00:02 /dev/mb_communication_driver
```

Prilikom instaliranja drajvera dozvoljavaju se prekidi, a svi dozvoljeni prekidi se mogu videti u `interrupts` fajlu koji se nalazi u `/proc` direktorijumu što je prikazano u listingu 5.2. Nisu prikazani svi prekidi koji su registrovani u `interrupts` fajlu, jer su prekidi od interesa na linijama 10 i 11 u listingu 5.2. Prva kolona predstavlja broj prekidnog zahteva (*IRQ number*). Druga kolona i treća kolona predstavljaju brojeve koji označavaju koliko puta se desio prekid i to prva kolona za procesor CPU0, a druga kolona za procesor CPU1. Usled pritiskanja tastera 3 puta, MicroBlaze je poslao 6 poruka, a ARM tri poruke. Prekid na liniji 10 je prekid koji MicroBlaze generiše kao acknowledge signal za prekid koji dobija od ARM procesora. Prekid na liniji 11 u listingu 5.2 generiše MicroBlaze kada god šalje poruku. U četvrtoj koloni je informacija o prekidnom kontroleru koji obrađuje prekide (*GIC - Generic Interrupt Controler* iz ZYNQ-7000 PS-a). [16]

Listing 5.2: Prikaz dozvoljenih prekida omogućen pozivanjem komande `cat /proc/interrupts`

```
root@zynq:~# insmod /mb_communication_driver.ko
MicroBlaze communication driver
------------------------------------------------------------------------
... pokretanje aplikacije, pritisak na taster 3 puta
... i zatvaranje aplikacije
------------------------------------------------------------------------
root@zynq:~# cat /proc/interrupts
...
 82:        128          0       GIC  82  xuartps
 90:          3          0       GIC  90  MB communication interrupt INTA
 91:          6          0       GIC  91  MB communication interrupt
IPI1:         0          0  Timer broadcast interrupts
...
```

Kada je instaliran drajver i pokrenuta aplikacija, omogućena je komunikacija između MicroBlaze procesora i Linux aplikacije. MicroBlaze preko *Uartlite* modula ispisuje poruke na terminal računara što je prikazano u listingu 5.3. Pritiskom na prvi taster šalje se poruka TAST1_PRESSED što je ispraćeno ispisom sa linije 15 u listingu 5.3. Zatim je primljena poruka koja je poslata od strane aplikacije (linija 16) i na kraju se šalje odgovor u vidu statusne poruke (linija 17). Slično je i za drugi i treći taster (linije 19 i 23 respektivno).

Listing 5.3: Ispis preko UART-a koji generiše MicroBlaze

```
1  ------------------------------------------------------------
2  Interprocessor communication project - MicroBlaze output
3  - Communication between Dual core ARM Cortex A-9 and MicroBlaze processors
4  - Platform: ZYNQ-7000 SoC zc706 Evaluation Board
5  ------------------------------------------------------
6  author: Vladimir Petrovic
7  - for the needs of diploma thesis
8  ------------------------------------------------------------
9
10 INIT...
11 Initialize the Communication BRAM...
12 Initialize the Communication BRAM DONE!
13 INIT finished.
14
15 Sending message to ZYNQ-7000 PS. Message code: 0x1111
16 Message arrived from ZYNQ-7000 PS. Message code: 0x1111
17 Sending message to ZYNQ-7000 PS. Message code: 0x5555
18
19 Sending message to ZYNQ-7000 PS. Message code: 0x2222
20 Message arrived from ZYNQ-7000 PS. Message code: 0x1233
21 Sending message to ZYNQ-7000 PS. Message code: 0x5555
22
23 Sending message to ZYNQ-7000 PS. Message code: 0x3333
24 Message arrived from ZYNQ-7000 PS. Message code: 0x1333
25 Sending message to ZYNQ-7000 PS. Message code: 0x5555
```

U listingu 5.4 prikazane su poruke koje prima i šalje Linux aplikacija. Nakon instaliranja drajvera (linija 7) i startovanja aplikacije (linija 9, odnosno 19), primljena je poruka od MicroBlaze procesora da je pritisnut prvi taster (linija 23). Aplikacija na osnovu te poruke, šalje poruku da se uključi LE dioda 1 (linija 24) i na kraju očekuje potvrdu uspešnosti koju dobija u vidu DONE_ACK poruke (linija 25). Slično je i za prijem poruka da su pritisnuti taster 2 (počev od linije 27) i taster 3 (počev od linije 31).

Listing 5.4: Učitavanje drajvera, pokretanje aplikacije u Linux-u i njen ispis

```
1  PetaLinux v2013.10 (Yocto 1.4) zynq ttyPS0
2
3  zynq login: root
4  Password:
5  login[743]: root login  on ttyPS0
6
7  root@zynq:~# insmod /mb_communication_driver.ko
8  MicroBlaze communication driver
9  root@zynq:~# ./mb-communication-app
10 ---------------------------------------------------
11 Interprocessor communication test application
12 - Communication between Dual core ARM Cortex A-9 and MicroBlaze processors
13 - Platform: ZYNQ-7000 SoC zc706 Evaluation Board
14 ---------------------------------------------------
15 author: Vladimir Petrovic
16 - for the needs of diploma thesis
17 ---------------------------------------------------
18 Press ENTER to start application...
19
20 Aplication started...
21 Please press one of GPIO pushbuttons on the board.
22
23 Received message from MicroBlaze. Message code: 0x1111  - pressed taster 1
24 Sending message to MicroBlaze. Turn on LED1  - Message code: 0x1111
25 Action succesfully done! DONE ackonowledge arrived from MicroBlaze.
26
27 Received message from MicroBlaze. Message code: 0x2222  - pressed taster 2
28 Sending message to MicroBlaze. Toggle LED2  - Message code: 0x1233
29 Action succesfully done! DONE ackonowledge arrived from MicroBlaze.
30
31 Received message from MicroBlaze. Message code: 0x3333  - pressed taster 3
32 Sending message to MicroBlaze. Toggle LED3  - Message code: 0x1333
33 Action succesfully done! DONE ackonowledge arrived from MicroBlaze.
```

Kako bi se jasno videli vremenski trenuci u kojima se šalju i primaju poruke, ispis MicroBlaze procesora je naknadno podešen da ide preko UART1 periferije iz ASIC dela čipa, tj. iste periferije preko koje se upravlja Linux-om. Na listingu 5.5 je prikazan ispis na terminal u ovom slučaju gde se jasno vidi redosled slanja, odnosno primanja poruka prilikom pritiska tastera 1, 2 i 3 respektivno. Ispis koji generiše MicroBlaze je označen `[MICROBLAZE]` prefiksom.

Listing 5.5: Ispis koji generišu Linux aplikacija i MicroBlaze kada je podešen UART1 za ispis sa MicroBlaze procesora

```
1  PetaLinux v2013.10 (Yocto 1.4) zynq ttyPS0
2
3  zynq login: root
4  Password:
5  login[743]: root login  on ttyPS0
6
7  root@zynq:~# insmod /mb_communication_driver.ko
8  MicroBlaze communication driver
9  root@zynq:~# ./mb-communication-app
10 ------------------------------------------------
11 Interprocessor communication test application
12 - Communication between Dual core ARM Cortex A-9 and MicroBlaze processors
13 - Platform: ZYNQ-7000 SoC zc706 Evaluation Board
14 ------------------------------------------------
15 author: Vladimir Petrovic
16 - for the needs of diploma thesis
17 ------------------------------------------------
18 Press ENTER to start application...
19
20 Aplication started...
21 Please press one of GPIO pushbuttons on the board.
22
23 [MICROBLAZE]: Sending message to ZYNQ-7000 PS. Message code: 0x1111
24 Received message from MicroBlaze. Message code: 0x1111  - pressed taster 1
25 Sending message to MicroBlaze. Turn on LED1  - Message code: 0x1111
26 [MICROBLAZE]: Message arrived from ZYNQ-7000 PS. Message code: 0x1111
27 [MICROBLAZE]: Sending message to ZYNQ-7000 PS. Message code: 0x5555
28 Action succesfully done! DONE ackonowledge arrived from MicroBlaze.
29
30
31 [MICROBLAZE]: Sending message to ZYNQ-7000 PS. Message code: 0x2222
32 Received message from MicroBlaze. Message code: 0x2222  - pressed taster 2
33 Sending message to MicroBlaze. Toggle LED2  - Message code: 0x1233
34 [MICROBLAZE]: Message arrived from ZYNQ-7000 PS. Message code: 0x1233
35 [MICROBLAZE]: Sending message to ZYNQ-7000 PS. Message code: 0x5555
36 Action succesfully done! DONE ackonowledge arrived from MicroBlaze.
37
38
39 [MICROBLAZE]: Sending message to ZYNQ-7000 PS. Message code: 0x3333
40 Received message from MicroBlaze. Message code: 0x3333  - pressed taster 3
41 Sending message to MicroBlaze. Toggle LED3  - Message code: 0x1333
42 [MICROBLAZE]: Message arrived from ZYNQ-7000 PS. Message code: 0x1333
43 [MICROBLAZE]: Sending message to ZYNQ-7000 PS. Message code: 0x5555
44 Action succesfully done! DONE ackonowledge arrived from MicroBlaze.
```

Ovim je prikazana uspešna realizacija međuprocesorske komunikacije. Međutim, za slanje i prijem poruka korišćena je samo po jedna memorijska lokacija što stvara određene nedostatke. Kada je baud rate *Uartlite* periferije, preko koje MicroBlaze vrši ispis na terminal PC računara, podešen na 9600b/s, primećeno je da kod dva uzastopna i brza pritiska na taster, drugi pritisak biva ignorisan. U tom slučaju još uvek nije detektovana komandna poruka koja treba da stigne od ARM procesora i nije poslata statusna poruka, pa je slanje nove poruke zabranjeno. Ovo se dešava jer se pre slanja statusne poruke na terminal ispisuje dugačka poruka (`Message arrived from ZYNQ-7000 PS. Message code: 0xYYYY`) koja notifikuje da je stigla poruka od ARM procesora. Za ispis je korišćena funkcija iz BSP-a za MicroBlaze podsistem koji je generisan u SDK. U ovoj funkciji je ispis realizovan tako što se za slanje svakog novog karaktera čeka na završetak slanja prethodnog. Ako je baud rate 9600b/s ispis od 57 karaktera koliko ima u poruci unosi kašnjenje od najmanje 60ms (prenosi se 10 bita po karakteru (1 start, 1 stop bit i 8-bitni podatak), ukupno 570 karaktera, što je samo za prenos, bez dohvatanja novog podatka i upisa u registar: $\frac{570\text{b}}{9600\text{b/s}} = 59,37\text{ms}$). Svakako, može se povećati baud rate UART-a ili napisati drugačija funkcija za ispis gde se novo slanje započinje u prekidnoj rutini UART-a i gde nije potrebno čekanje. Međutim, u kompleksnijim sistemima to ne rešava problem kada obrada poruka traje duže jer se i tada može desiti da se ne registruju sve poruke. Zbog toga bi trebalo realizovati bafer za poruke u koji bi se smeštale poruke iako prethodne nisu obrađene. Svakako, i ova realizacija ima ograničenje u vidu veličine bafera, ali se smanjuje verovatnoća gubljenja podataka.

Realizacija bafera za poruke može biti i softverska. U postojećem sistemu postoji dovoljno deljene memorije u koju se može smestiti veliki broj poruka. Svaki od dva podsistema može da zna koliko je poruka stiglo na osnovu broja prekidnih zahteva i na osnovu toga koliko je poruka obrađeno. Svaka nova poruka može da se upisuje na prvu sledeću memorijsku lokaciju, a kada se dođe do kraja bafera, upisuje se na početnu adresu bafera, što je zapravo emulacija FIFO bafera. Na osnovu informacija o tome koliko je poruka stiglo i koliko je od tih poruka obrađeno, prijemni podsistem može da odredi adresu sa koje čita podatak. Ovakva realizacija ne zahteva promene u hardverskom dizajnu i može biti tema nastavka ovog rada uz pokretanje ozbiljnijeg sistema u kome je potrebna brza međuprocesorska komunikacija.

Glava 6

Zaključak

Prikazana je jedna realizacija međuprocesorske komunikacije u heterogenom višeprocesorskom sistemu. Za potrebe ove realizacije, najpre je instanciran MicroBlaze procesor sa pratećim periferijama i IP blokovima potrebnim za međuprocesorsku komunikaciju u programabilnoj logici ZYNQ-7000 programabilnog sistema na čipu kompanije Xilinx. Komunikacija je ostvarena između MicroBlaze procesora i Dual core ARM Cortex-A9 procesora koji se nalazi u ASIC delu čipa (*ZYNQ Processing System*) i na kome se izvršava Linux operativni sistem. Međuprocesorska komunikacija je indirektna, a razmena poruka se ostvaruje upisom u deljenu blok RAM memoriju koja se nalazi u programabilnoj logici. Obaveštenje o poslatoj poruci svaki procesor daje drugom procesoru prekidnim zahtevom. Aplikacija za komunikaciju sa MicroBlaze procesorom se izvršava na Linux operativnom sistemu. Zato je sama komunikacija implementirana u drajveru kome se iz aplikacije može pristupiti `ioctl` sistemskim pozivom prosleđivanjem određenih komandi. Tako je od aplikacije razdvojen sam mehanizam slanja i prijema poruka, ali `ioctl` sistemskim pozivom aplikacija može da prosledi poruku za slanje ili pročita primljenu poruku.

Prikazani princip razmene poruka nije redak u višeprocesorskim sistemima. Čak za međuprocesorku komunikaciju ovog tipa postoje i posebne periferije - *mailbox* periferije koje automatski generišu prekidne zahteve i mogu biti napravljene za komunikaciju između nekoliko procesora. U Vivado Design Suite razvojnom okruženju u IP bibliotekama postoji jedna ovakva periferija namenjena komunikaciji između dva procesora, međutim ona nije korišćena što može biti nastavak ovog rada. Takođe, korisno bi bilo uraditi dizajn *mailbox* periferije u nekom od jezika za opis hardvera koja može da prosleđuje poruke između više procesora. Međutim, emulacija *mailbox* periferije opisana u ovom radu i dalje može poslužiti kao dovoljno efikasan način komunikacije za mnoge primene.

U diskusiji rezultata je pomenuto da se u trenutnoj realizaciji koristi samo jedna memorijska lokacija za slanje i jedna za prijem što je nedostatak ovog rada jer se može desiti da se, usled spore obrade primljenih poruka, neke od primljenih poruka ignorišu. Dalji rad na ovom projektu bi mogao da se zasniva na odstranjivanju ovog nedostatka kreiranjem bafera za poruke i primeni međuprocesorske komunikacije na nekom ozbiljnijem projektu gde MicroBlaze može da obavlja neke specifične poslove i da rezultate povremeno šalje ARM procesorima.

Zahvalnost

Zahvaljujem se svom mentoru prof. dr Lazaru Saranovcu sa Elektrotehničkog fakulteta u Beogradu na sugestijama i savetima tokom izrade rada. Posebnu zahvalnost dugujem i celom timu kompanije Aggios (Aggios Europe d.o.o. i Aggios Incorporated - *aggios.com*) u kojoj sam obavljao stručnu praksu tokom koje je i nastao ovaj rad.

Autor

Literatura

[1] John L. Hennessy, David A. Patterson, *Computer Architecture - A Quantitative Approach*, Elsevier, 2007.

[2] www.arm.com/products/processors/technologies/biglittleprocessing.php

[3] Mirela Simonović, Vojin Živojnović, Davorin Mista, Strahinja Janković, Lazar Saranovac, *Energy Proportional Management of Residential Gateways*, Telekomunikacioni forum - TELFOR, novembar 2013.

[4] www.xilinx.com/products/intellectual-property/mutex.htm

[5] www.quicklogic.com/technologies/connectivity/ipc/

[6] *OMAP5910 Dual-Core Processor Inter-Processor Communication Reference Guide*, Texas Instruments Incorporated, 2005. dostupno na www.ti.com/lit/ug/spru683a/spru683a.pdf

[7] *Zynq-7000 All Programmable SoC Technical Reference Manual*, Xilinx, septembar 2013.

[8] *ZC706 Evaluation Boardfor the Zynq-7000 XC7Z045 All Programmable SoC User Guide*, Xilinx, jul 2013.

[9] *MicroBlaze Processor Reference Guide*, Xilinx, april 2012.

[10] *AXI Reference Guide*, Xilinx, januar 2012.

[11] *AMBA® AXI Protocol v1.0*, ARM, 2004.

[12] *LogiCORE IP I/O Module v2.2 - Product Guide for Vivado Design Suite*, Xilinx, decembar 2013.

[13] *MicroBlaze Debug Module (MDM) v2.10.a - Product Guide*, Xilinx, jul 2012.

[14] *Data2MEM User Guide*, Xilix, jun 2009.

[15] www.wiki.xilinx.com/Zynq+Linux

[16] Jonathan Corbet, Alessandro Rubini, Greg Kroah-Hartman, *Linux Device Drivers*, O'RELLY, 2005.

Dodaci

Dodatak A

Programski kodovi

U ovom dodatku su prikazani programski kodovi aplikacija i drajvera uz kratka objašnjenja koje funkcionalnosti su realizovane u svakom od fajlova.

A.1 Kod MicroBlaze aplikacije

U `main` funkciji se vrši inicijalizacija sistema, a zatim se konstantno proverava da li je stigla poruka od ARM procesora i ako jeste, izvršava se određena akcija (uključenje, isključenje ili menjanje stanja specificirane LE diode) i na kraju šalje odgovarajući odgovor. U slučaju nepoznate primljene poruke, kao odgovor se šalje status greške.

main.c

```
1  /*
2   * main.c
3   *
4   *  Created on: Mar 23, 2014
5   *      Author: vladimir
6   */
7  #include "gpio.h"
8  #include "config_init.h"
9  #include "interrupts.h"
10 #include "communication.h"
11 #include "dbg_defines.h"
12
13 void user_test(void);
14
15 volatile unsigned int received_message = 0;
16 volatile unsigned int new_message = 0;
17 unsigned char action_string[20];
18
19
20 int main () {
21     xil_printf("-----\n\r");
22     xil_printf("Interprocessor communication project - MicroBlaze output\n\r\
23 - Communication between Dual core ARM Cortex A-9 and MicroBlaze processors\n\r\
24 - Platform: ZYNQ-7000 SoC zc706 Evaluation Board\n\r");
25     xil_printf("-----\n\r");
26     xil_printf("author: Vladimir Petrovic\n\r\
27 - for the needs of diploma thesis\n\r");
28     xil_printf("-----\n\r\n\r");
29     xil_printf("INIT...\n\r");
30
31     configure_and_init();
32
33     xil_printf("INIT finished.\n\r\n\r");
```

```
34
35      while (1) {
36          if (new_message)
37          {
38              new_message = 0;
39  #ifdef DIPLOMA_DEMO
40  #ifdef PS7_UART1
41              xil_printf("[MICROBLAZE]: Message arrived from ZYNQ-7000 PS. Message code: 0x%x\n\r", received_message);
42  #else
43              xil_printf("Message arrived from ZYNQ-7000 PS. Message code: 0x%x\n\r", received_message);
44  #endif
45  #endif
46              switch (received_message & LED_MASK) {
47              case LED0:
48              {
49                  switch (received_message & ACTION_MASK) {
50                  case TURN_ON: {unsigned int msg = DONE_ACK; turn_on_led(LED0); send_message_to_arm(&msg); break;}
51                  case TURN_OFF: {unsigned int msg = DONE_ACK; turn_off_led(LED0); send_message_to_arm(&msg); break;}
52                  case TOGGLE: {unsigned int msg = DONE_ACK; toggle_led(LED0); send_message_to_arm(&msg); break;}
53                  default: {unsigned int msg = ERROR_ACK; send_message_to_arm(&msg); break;}
54                  }
55                  break;
56              }
57              case LED1:
58              {
59                  switch (received_message & ACTION_MASK) {
60                  case TURN_ON: {unsigned int msg = DONE_ACK; turn_on_led(LED1); send_message_to_arm(&msg); break;}
61                  case TURN_OFF: {unsigned int msg = DONE_ACK; turn_off_led(LED1); send_message_to_arm(&msg); break;}
62                  case TOGGLE: {unsigned int msg = DONE_ACK; toggle_led(LED1); send_message_to_arm(&msg); break;}
63                  default: {unsigned int msg = ERROR_ACK; send_message_to_arm(&msg); break;}
64                  }
65                  break;
66              }
67              case LED2:
68              {
69                  switch (received_message & ACTION_MASK) {
70                  case TURN_ON: {unsigned int msg = DONE_ACK; turn_on_led(LED2); send_message_to_arm(&msg); break;}
71                  case TURN_OFF: {unsigned int msg = DONE_ACK; turn_off_led(LED2); send_message_to_arm(&msg); break;}
72                  case TOGGLE: {unsigned int msg = DONE_ACK; toggle_led(LED2); send_message_to_arm(&msg); break;}
73                  default: {unsigned int msg = ERROR_ACK; send_message_to_arm(&msg); break;}
74                  }
75                  break;
76              }
77              case LED3:
78              {
79                  switch (received_message & ACTION_MASK) {
80                  case TURN_ON: {unsigned int msg = DONE_ACK; turn_on_led(LED3); send_message_to_arm(&msg); break;}
81                  case TURN_OFF: {unsigned int msg = DONE_ACK; turn_off_led(LED3); send_message_to_arm(&msg); break;}
82                  case TOGGLE: {unsigned int msg = DONE_ACK; toggle_led(LED3); send_message_to_arm(&msg); break;}
83                  default: {unsigned int msg = ERROR_ACK; send_message_to_arm(&msg); break;}
84                  }
85                  break;
86              };
87              default:
88                  {unsigned int msg = ERROR_ACK; send_message_to_arm(&msg); break;}
89              }
90              can_send_message_to_arm = 1;
91
92          }
93      }
94      return 0;
95  }
```

U `config_init.h` i `config_init.c` fajlovima definisane su funkcije za inicijalizaciju periferija i odobravanje prekida. U U `config_init.h` fajlu deklarisane su strukture preko kojih se pristupa I/O modulu i BRAM memoriji.

config_init.h

```
#ifndef CONFIG_INIT_H_
#define CONFIG_INIT_H_

#include "xbram.h"
#include "xiomodule.h"

#define IOM_ID                  XPAR_IOMODULE_INTC_SINGLE_DEVICE_ID
extern XIOModule                IOModule;

extern XBram bram;
extern XBram_Config *bramcfg;

int configure_and_init(void);

#endif /* CONFIG_INIT_H_ */
```

config_init.c

```
#include "gpio.h"
#include "interrupts.h"
#include "config_init.h"
#include "xiomodule.h"
#include "xil_exception.h"

XBram bram;
XBram_Config *bramcfg;

static int bram_init()
{
    int status;
    xil_printf("Initialize the Communication BRAM...\n\r");
    bramcfg = XBram_LookupConfig(XPAR_AXI_BRAM_CTRL_0_DEVICE_ID);
    status = XBram_CfgInitialize(&bram, bramcfg, bramcfg->CtrlBaseAddress);
    if (status != XST_SUCCESS) {
        printf ("failed!\r\n");
        return status;
    }
    xil_printf("Initialize the Communication BRAM DONE!\n\r");
}


int configure_and_init(void)
{
    //init IO module
    u32 Status = XST_SUCCESS;
    Status = XIOModule_Initialize(&IOModule, IOM_ID);
    if (Status != XST_SUCCESS) {
        xil_printf("Could initialize IO Module\n\r");
        return -1;
    }

    //initialize communication BRAM
    bram_init();

    //set interrupts
    set_interrupt_gpi1();
    set_interrupt_arm();

    //Initialize the Microblaze exception table.
    Xil_ExceptionInit();

    //Register the IO module interrupt handler with the exception table.
    Xil_ExceptionRegisterHandler(XIL_EXCEPTION_ID_INT,
            (Xil_ExceptionHandler) XIOModule_DeviceInterruptHandler, (void*) 0);

    // Enable Microblaze exceptions
    Xil_ExceptionEnable();

    return Status;
}
```

U `communication.h` i `communication.c` fajlovima su opisane funkcije za slanje i prijem poruka i definisane sve poruke koje se koriste.

communication.h

```
#ifndef COMMUNICATION_H_
#define COMMUNICATION_H_

// communication with ARM locations
#define RX_LOCATION_OFFSET            0x00000008
#define TX_LOCATION_OFFSET            0x00000000

// messages to ARM
#define TAST1_PRESSED    0x1111
#define TAST2_PRESSED    0x2222
#define TAST3_PRESSED    0x3333
#define DONE_ACK         0x5555
#define ERROR_ACK        0xAAAA

//received message macros
/*
15          8 7          0
-------------------------
|    LEDx     |   ACTION   |
-------------------------
*/
#define LED_MASK     0xFF00
#define LED0         0x1000
#define LED1         0x1100
#define LED2         0x1200
#define LED3         0x1300

#define ACTION_MASK 0x00FF
#define TURN_ON      0x11
#define TURN_OFF     0x22
#define TOGGLE       0x33

extern volatile unsigned int received_message;

extern volatile unsigned int new_message;

extern void send_message_to_arm(unsigned int *data);

extern void receive_message_from_arm(unsigned int *data);

#endif /* COMMUNICATION_H_ */
```

communication.c

```
#include "communication.h"
#include "interrupts.h"
#include "config_init.h"
#include "dbg_defines.h"


void send_message_to_arm(unsigned int *data)
{
#ifdef DIPLOMA_DEMO
#ifdef PS7_UART1
    xil_printf("[MICROBLAZE]: Sending message to ZYNQ-7000 PS. Message code: 0x%x\n\r", *data↩
        );
#else
    xil_printf("Sending message to ZYNQ-7000 PS. Message code: 0x%x\n\r", *data);
#endif
#endif
    XBram_WriteReg(bramcfg->MemBaseAddress, TX_LOCATION_OFFSET, *data);
    trigger_arm_interrupt(ARM_IRQ_COMMUNICATION_MASK);
#ifdef DIPLOMA_DEMO
    if (*data == DONE_ACK || *data == ERROR_ACK)
        xil_printf("\n\r");
#endif
}

void receive_message_from_arm(unsigned int *data)
{
    *data = XBram_ReadReg(bramcfg->MemBaseAddress, RX_LOCATION_OFFSET);
}
```

U fajlu `dbg_defines.h` definišu se imena koja se koriste za uslovno prevođenje određenih sekcija koda koje se odnose na to da li će i kako će se generisati ispis.

dbg_defines.h

```
#ifndef DBG_DEFINES_H_
#define DBG_DEFINES_H_

//#define DBG_INTERRUPTS
//#define DBG_GPIO
#define DIPLOMA_DEMO
//#define PS7_UART1

#endif /* DBG_DEFINES_H_ */
```

U `gpio.h` i `gpio.c` fajlovima definisane su funkcije za dohvatanje i setovanje vrednosti GPIO portova iz I/O modula, kao i funkcije za uključenje, isključenje i menjanje stanje specificirane LE diode.

gpio.h

```
#ifndef GPIO_H_
#define GPIO_H_

#include "config_init.h"

//GPIO channel id
#define GPO_LED_CHAN_ID                 0x1
#define GPI_BUTTONS_CHAN_ID             0x1
#define GPO_INTR_CHAN_ID                0x2

#define GPO_LED_0_MASK                  0x1
#define GPO_LED_1_MASK                  0x2
#define GPO_LED_2_MASK                  0x4
#define GPO_LED_3_MASK                  0x8

unsigned char gpi_get_value(unsigned gpio_channel);
void gpio_set_mask(unsigned gpio_channel, unsigned mask);
void gpio_clear_mask(unsigned gpio_channel, unsigned mask);

void turn_on_led(unsigned int led);
void turn_off_led(unsigned int led);
void toggle_led(unsigned int led);

#endif /* GPIO_H_ */
```

gpio.c

```
#include "gpio.h"
#include "communication.h"

static unsigned char gpo_outs[2] = {0x00, 0x00};    //current GPO values

unsigned char gpi_get_value(unsigned gpio_channel)
{
    unsigned int gpi_offset = ((gpio_channel - 1) * XGPI_CHAN_OFFSET) + XGPI_DATA_OFFSET;
    return XIOModule_ReadReg(IOModule.BaseAddress, gpi_offset);
}

static unsigned char gpo_get_value(unsigned gpio_channel)
{
    unsigned int gpo_offset = ((gpio_channel - 1) * XGPO_CHAN_OFFSET) + XGPO_DATA_OFFSET;
    return XIOModule_ReadReg(IOModule.BaseAddress, gpo_offset);
}

void gpio_set_mask(unsigned gpio_channel, unsigned mask)
{
    unsigned int gpo_offset = ((gpio_channel - 1) * XGPO_CHAN_OFFSET) + XGPO_DATA_OFFSET;
#ifdef DBG_GPIO
    xil_printf("set mask, old reg value: %x new value %x\n\r", gpo_outs[gpio_channel - 1], ↩
        gpo_outs[gpio_channel - 1] | mask);
#endif
    XIOModule_WriteReg(IOModule.BaseAddress, gpo_offset, gpo_outs[gpio_channel - 1] | mask);
    gpo_outs[gpio_channel - 1] |= mask;
```

```
}

void gpio_clear_mask(unsigned gpio_channel, unsigned mask)
{
    unsigned int gpo_offset = ((gpio_channel - 1) * XGPO_CHAN_OFFSET) + XGPO_DATA_OFFSET;
#ifdef DBG_GPIO
    xil_printf("clear mask, old reg value: %x new value %x\n\r", gpo_outs[gpio_channel - 1], ↩
        gpo_outs[gpio_channel - 1] & ~mask);
#endif
    gpo_outs[gpio_channel - 1] &= ~mask;
    XIOModule_WriteReg(IOModule.BaseAddress, gpo_offset, gpo_outs[gpio_channel - 1] & ~mask);
}

void turn_on_led(unsigned int led) {
    switch (led) {
    case LED0: {gpio_set_mask(GPO_LED_CHAN_ID, GPO_LED_0_MASK);break;}
    case LED1: {gpio_set_mask(GPO_LED_CHAN_ID, GPO_LED_1_MASK);break;}
    case LED2: {gpio_set_mask(GPO_LED_CHAN_ID, GPO_LED_2_MASK);break;}
    case LED3: {gpio_set_mask(GPO_LED_CHAN_ID, GPO_LED_3_MASK);break;}
    }
}

void turn_off_led(unsigned int led) {
    switch (led) {
    case LED0: {gpio_clear_mask(GPO_LED_CHAN_ID, GPO_LED_0_MASK);break;}
    case LED1: {gpio_clear_mask(GPO_LED_CHAN_ID, GPO_LED_1_MASK);break;}
    case LED2: {gpio_clear_mask(GPO_LED_CHAN_ID, GPO_LED_2_MASK);break;}
    case LED3: {gpio_clear_mask(GPO_LED_CHAN_ID, GPO_LED_3_MASK);break;}
    }
}

void toggle_led(unsigned int led) {
    switch (led) {
    case LED0:
    {
        if (gpo_outs[0] & GPO_LED_0_MASK)
            gpio_clear_mask(GPO_LED_CHAN_ID, GPO_LED_0_MASK);
        else
            gpio_set_mask(GPO_LED_CHAN_ID, GPO_LED_0_MASK);
        break;
    }
    case LED1:
    {
        if (gpo_outs[0] & GPO_LED_1_MASK)
            gpio_clear_mask(GPO_LED_CHAN_ID, GPO_LED_1_MASK);
        else
            gpio_set_mask(GPO_LED_CHAN_ID, GPO_LED_1_MASK);
        break;
    }
    case LED2:
    {
        if (gpo_outs[0] & GPO_LED_2_MASK)
            gpio_clear_mask(GPO_LED_CHAN_ID, GPO_LED_2_MASK);
        else
            gpio_set_mask(GPO_LED_CHAN_ID, GPO_LED_2_MASK);
        break;
    }
    case LED3:
    {
        if (gpo_outs[0] & GPO_LED_3_MASK)
            gpio_clear_mask(GPO_LED_CHAN_ID, GPO_LED_3_MASK);
        else
            gpio_set_mask(GPO_LED_CHAN_ID, GPO_LED_3_MASK);
        break;
    }
    }
}
```

U `interrupts.h` i `interrupts.c` fajlovima su definisane sve funkcije za setovanje i obradu prekida na MicroBlaze podsistemu, kao i funkcija za slanje prekidnog zahteva ARM procesorima.

interrupts.h

```
#ifndef INTERRUPTS_H_
#define INTERRUPTS_H_

#include "xbasic_types.h"

extern unsigned char can_send_message_to_arm;   //this variable is set to 0 when MB sends a message
                                                //and command from arm hasn't yet arrived


//MicroBlaze interrupts
//currently we don't have MB interrupts, but functions are written for possible later use
void interrupt_Register(int intr_id, XInterruptHandler isr, void *args);
void interrupt_Enable(int intr_id);
void interrupt_Disable(int intr_id);
void interrupt_Clear(int intr_id);

//ARM interrupts
void trigger_arm_interrupt(int intr_id_mask);

#define ARM_WAKEUP_EVENT_MASK           0x01
#define ARM_IRQ_0_MASK                  0x02
#define ARM_IRQ_1_MASK                  0x04
#define ARM_IRQ_COMMUNICATION_INTA_MASK 0x08
#define ARM_IRQ_COMMUNICATION_MASK      0x10

//set interrupt functions
void set_interrupt_arm(void);
void set_interrupt_gpi1(void);

#endif /* INTERRUPTS_H_ */
```

interrupts.c

```
#include "interrupts.h"
#include "xil_exception.h"
#include "gpio.h"
#include "communication.h"
#include "dbg_defines.h"

XIOModule               IOModule;   // Instance of the IO Module
static unsigned int Intr_En_Shadow = 0;

unsigned char can_send_message_to_arm = 1;


// Functions responsible for enabling/disabling/clearing/registering the interrupt
void interrupt_Enable(int intr_id)
{
    Intr_En_Shadow |= 1 << intr_id;
    XIOModule_EnableIntr(IOModule.BaseAddress, (Intr_En_Shadow));
}

void interrupt_Disable(int intr_id)
{
    Intr_En_Shadow &= ~(1 << intr_id);
    XIOModule_EnableIntr(IOModule.BaseAddress, (Intr_En_Shadow));
}

void interrupt_Clear(int intr_id)
{
    XIOModule_AckIntr(IOModule.BaseAddress, (1 << intr_id));
}

void interrupt_Register(int intr_id, XInterruptHandler isr, void *args)
{
    XIOModule_RegisterHandler(IOModule.BaseAddress, intr_id, isr, args);
}


```

```
37 void trigger_arm_interrupt(int cpu_id) //triggers interrupt to ARM
38 {
39     int i, j;
40 #ifdef DBG_INTERRUPTS
41     xil_printf("Asserting the interrupt to arm\n\r");
42 #endif
43     gpio_set_mask(GPO_INTR_CHAN_ID, cpu_id);
44
45     // wait for appropriate time (todo: calibrate if needed)
46     for (i = 0; i < 0xff; i++)
47         for (j = 0; j < 0xff; j++);
48     gpio_clear_mask(GPO_INTR_CHAN_ID, cpu_id);
49 #ifdef DBG_INTERRUPTS
50     xil_printf("De-asserted the interrupt\n\r");
51 #endif
52 }
53
54 //Interrupt service routine for interrupt generated from ARM
55 void arm_isr(void *arg)
56 {
57     trigger_arm_interrupt(ARM_IRQ_COMMUNICATION_INTA_MASK);
58 #ifdef DBG_INTERRUPTS
59     xil_printf("ARM generated interrupt!\n\r");
60 #endif
61     receive_message_from_arm(&received_message);
62     new_message = 1;
63 }
64
65 //sets and enables interrupt generated from ARM
66 void set_interrupt_arm(void){
67     interrupt_Register(XIN_IOMODULE_EXTERNAL_INTERRUPT_INTR + ↩
        XPAR_IOMODULE_0_XLSLICE_1_DOUT_INTR, arm_isr, 0);
68     interrupt_Enable(XIN_IOMODULE_EXTERNAL_INTERRUPT_INTR + ↩
        XPAR_IOMODULE_0_XLSLICE_1_DOUT_INTR);
69 }
70
71 // GPI1 interrupt triggered whenever the GPI1 pin is asserted
72 //Interrupt service routine for GPI1 interrupt
73 void gpi1_isr(void *arg)
74 {
75 #ifdef DBG_GPIO
76     xil_printf("GPI1 ISR \n\r");
77 #endif
78     if (gpi_get_value(GPI_BUTTONS_CHAN_ID) & 0x01) {
79         unsigned int message = TAST1_PRESSED;
80         if (can_send_message_to_arm)
81         {
82             send_message_to_arm(&message);
83             can_send_message_to_arm = 0;
84         }
85     }
86     else
87     if (gpi_get_value(GPI_BUTTONS_CHAN_ID) & 0x02) {
88         unsigned int message = TAST2_PRESSED;
89         if (can_send_message_to_arm)
90         {
91             send_message_to_arm(&message);
92             can_send_message_to_arm = 0;
93         }
94     }
95     else
96     if (gpi_get_value(GPI_BUTTONS_CHAN_ID) & 0x04) {
97         unsigned int message = TAST3_PRESSED;
98         if (can_send_message_to_arm)
99         {
100            send_message_to_arm(&message);
101            can_send_message_to_arm = 0;
102        }
103    }
104 }
105
106 //sets and enables GPI1 interrupt
107 void set_interrupt_gpi1(){
108     interrupt_Register(XIN_IOMODULE_GPI_1_INTERRUPT_INTR, gpi1_isr, 0);
109     interrupt_Enable(XIN_IOMODULE_GPI_1_INTERRUPT_INTR);
110 }
```

A.2 Kod drajvera

Glavni deo drajvera je opisan u `module.c` fajlu. Tu su opisane funkcije za otvaranje (`mb_communication_driver_open()`) i zatvaranje (`mb_communication_driver_close()`) device fajla i *ioctl* funkcija (`mb_communication_driver_ioctl()`), kao i funkcije za instaliranje i uklanjanje drajvera. U `module.h` fajlu su definisane naredbe za *ioctl* funkciju i još neki makroi.

Za pristup memorijskim lokacijama u deljenoj memoriji i generisanje prekida koriste se funkcije iz `communication.h` i `communication.c` fajlova.

module.h

```
#ifndef _MODULE_H_
#define _MODULE_H_

// major number
#define MAJOR_NUMBER                247

#define DEVICE_FILE_NAME            "mb_communication_driver"

#define SUCCESS                     0

// IOCTL
#define SEND_COMMAND                _IOW(MAJOR_NUMBER, 1, unsigned int)
#define RECEIVE_STATUS          _IOR(MAJOR_NUMBER, 2, unsigned int)
#define RECEIVE_ACTION          _IOR(MAJOR_NUMBER, 3, unsigned int)

#define NO_NEW_MESSAGE          0x3

#endif // _MODULE_H_
```

module.c

```
#include <linux/module.h>
#include <linux/buffer_head.h>
#include <linux/device.h>
#include <linux/cdev.h>
#include <linux/interrupt.h>
#include <asm/io.h>                 // ioread, iowrite...
#include <linux/ioport.h>   //request_mem_region

#include "communication.h"
#include "module.h"

static int device_open = 0;
static int mb_communication_driver_count = 1;

static dev_t mb_communication_driver_dev;
static struct cdev mb_communication_driver_cdev;
static struct class *mb_communicationcl; // Global variable for the device class

unsigned int mb_message = 0;
unsigned int new_message_arrived = 0;

//open
static int mb_communication_driver_open(struct inode *inode, struct file *file)
{
    if (device_open)
        return -EBUSY;

    device_open++;
    try_module_get(THIS_MODULE);

    return SUCCESS;
}

//close
static int mb_communication_driver_release(struct inode *inode, struct file *file)
{
    device_open--;
    module_put(THIS_MODULE);

    return SUCCESS;
}

```

```
43  //ioctl
44  static long mb_communication_driver_ioctl(struct file *f, unsigned int cmd, unsigned long arg↩
        )
45  {
46      unsigned int data;
47      int retval = 0;
48      switch (cmd)
49      {
50          case SEND_COMMAND:
51              retval = get_user(data, (unsigned int __user *)arg);
52              if (retval != 0)
53              {
54                  printk("IOCTL Error\n");
55                  return -1;
56              }
57              send_message_to_microblaze(&data);
58              break;
59          case RECEIVE_STATUS:
60              if (new_message_arrived) {
61                  new_message_arrived = 0;
62                  retval = __put_user(mb_message, (unsigned int __user *)arg);
63                  if (retval != 0)
64                  {
65                      printk("IOCTL Error\n");
66                      return -1;
67                  }
68              }
69              else
70              {
71                  __put_user(NO_NEW_MESSAGE, (unsigned int __user *)arg);
72                  return -1;
73              }
74              break;
75          case RECEIVE_ACTION:
76              if (new_message_arrived) {
77                  new_message_arrived = 0;
78                  retval = __put_user(mb_message, (unsigned int __user *)arg);
79                  if (retval != 0)
80                  {
81                      printk("IOCTL Error\n");
82                      return -1;
83                  }
84              }
85              else
86              {
87                  __put_user(NO_NEW_MESSAGE, (unsigned int __user *)arg);
88                  return -1;
89              }
90              break;
91          default:
92              printk("ERROR: Unknown ioctl\n");
93              return -1;
94      }
95      return retval;
96  }
97
98  // fops
99  static struct file_operations mb_communication_driver_fops =
100 {
101     .owner = THIS_MODULE,
102     .open = mb_communication_driver_open,
103     .release = mb_communication_driver_release,
104     .read = NULL,
105     .write = NULL,
106     .unlocked_ioctl = mb_communication_driver_ioctl,
107 };
108 //microblaze communication handler
109 static irqreturn_t mb_communication_handler(int irq, void *dev_id)
110 {
111     read_message_from_microblaze(&mb_message);
112     new_message_arrived = 1;
113 #ifdef DBG_MB_COMMUNICATION_DRIVER
114     printk(KERN_INFO "*********Message from MicroBlaze********\n");
115     printk("Received ..... %x", mb_message);
116 #endif
117     return IRQ_HANDLED;
```

```
}
//INTA from microblaze handler
static irqreturn_t mb_communication_inta_handler(int irq, void *dev_id)
{
#ifdef DBG_MB_COMMUNICATION_DRIVER
    printk(KERN_INFO "*********MBINTA********\n");
#endif
    deassert_interrupt_to_microblaze();
    return IRQ_HANDLED;
}

//init
static int __init mb_communication_driver_init(void)
{
    int err;
    struct device* dev;

    if (alloc_chrdev_region(&mb_communication_driver_dev, 1, mb_communication_driver_count, DEVICE_FILE_NAME) < 0)
    {
        err = -ENODEV;
        goto err_dev;
    }

    if ((mb_communicationcl = class_create(THIS_MODULE, "mb_communicationdrv")) == NULL)
    {
        err = -ENODEV;
        goto err_dev_unreg;
    }

    if ((dev = device_create(mb_communicationcl, NULL, mb_communication_driver_dev, NULL, "mb_communication_driver")) == NULL)
    {
        err = -ENODEV;
        goto err_cl_dest;
    }

    cdev_init(&mb_communication_driver_cdev, &mb_communication_driver_fops);

    if (cdev_add(&mb_communication_driver_cdev, mb_communication_driver_dev, mb_communication_driver_count) == -1)
    {
        err = -ENODEV;
        goto err_dev_dest;
    }

//request_irq for interrupt that MicroBlaze generates as signal that message is ready
    if (request_irq(91, mb_communication_handler, IRQF_SHARED | IRQF_TRIGGER_RISING,
        "MB communication interrupt", (void *)mb_communication_handler) != 0)
    {
        printk("Error allocating interrupt\n");
        return 0;
    }

//request_irq for interrupt that MicroBlaze generates as interrupt acknowledge for interrupt
//that is generated by mb_communication driver
    if (request_irq(90, mb_communication_inta_handler, IRQF_SHARED | IRQF_TRIGGER_RISING,
        "MB communication interrupt INTA", (void *)mb_communication_inta_handler) != 0)
    {
        printk("Error allocating interrupt\n");
        return 0;
    }

    init_communication();
    printk("MicroBlaze communication driver\n");
    return 0;

err_dev_dest:
    cdev_del(&mb_communication_driver_cdev);
    device_destroy(mb_communicationcl, mb_communication_driver_dev);
err_cl_dest:
    class_destroy(mb_communicationcl);
err_dev_unreg:
    unregister_chrdev_region(mb_communication_driver_dev, mb_communication_driver_count);
err_dev:
    printk("Fail\n");
```

```
191     return err;
192 }
193 
194 //exit
195 static void __exit mb_communication_driver_exit(void)
196 {
197     end_communication();
198     cdev_del(&mb_communication_driver_cdev);
199     device_destroy(mb_communicationcl, mb_communication_driver_dev);
200     class_destroy(mb_communicationcl);
201     unregister_chrdev_region(mb_communication_driver_dev, mb_communication_driver_count);
202     printk("MicroBlaze communication driver exit\n");
203 }
204 
205 module_init(mb_communication_driver_init);
206 module_exit(mb_communication_driver_exit);
207 
208 MODULE_LICENSE("GPL");
```

communication.h

```
1  #ifndef _COMMUNICATION_H_
2  #define _COMMUNICATION_H_
3  
4  //communication adresses
5  #define AXI_BRAM_BASE_ADDRESS               0x40000000  //memory used for passing messages
6  #define AXI_MB_INTERRUPT_REQ_REG            0x41200000  //write 1 to this address triggers ↩
       interrupt in MicroBlaze
7  
8  //
9  #define AXI_BRAM_MEM_LENGTH              0x00001000
10 
11 // communication locations
12 #define TX_LOCATION_OFFSET             0x00000008
13 #define RX_LOCATION_OFFSET             0x00000000
14 
15 //received messages from MicroBlaze
16 #define DONE_ACK         0x5555
17 #define ERROR_ACK        0xAAAA
18 
19 //sending message macros
20 /*
21 15          8 7          0
22 ------------------------------
23 |    LEDx     |   ACTION    |
24 ------------------------------
25 */
26 #define LED_MASK     0xFF00
27 #define LED0         0x1000
28 #define LED1         0x1100
29 #define LED2         0x1200
30 #define LED3         0x1300
31 
32 #define ACTION_MASK 0x00FF
33 #define TURN_ON      0x11
34 #define TURN_OFF     0x22
35 #define TOGGLE       0x33
36 
37 extern void* bram_start_remapped;
38 extern void* mb_interrupt_start_remapped;
39 
40 extern int init_communication(void);
41 
42 extern int end_communication(void);
43 
44 extern void assert_interrupt_to_microblaze(void);
45 
46 extern void deassert_interrupt_to_microblaze(void);
47 
48 extern void send_message_to_microblaze(unsigned int *data);
49 
50 extern void read_message_from_microblaze(unsigned int *data);
51 
52 #endif // _COMMUNICATION_H_
```

communication.c

```c
#include <linux/module.h>
#include <asm/io.h>
#include <linux/ioport.h>        // request_mem_region
#include "communication.h"

void* bram_start_remapped;
void* mb_interrupt_start_remapped;

int init_communication(void)
{
    if (request_mem_region(AXI_BRAM_BASE_ADDRESS, AXI_BRAM_MEM_LENGTH, "bram") == NULL)
    {
        printk("Error while requesting Communication BRAM memory region\n");
        return -1;
    }

    bram_start_remapped = ioremap(AXI_BRAM_BASE_ADDRESS, AXI_BRAM_MEM_LENGTH);

    if (bram_start_remapped == NULL)
    {
        printk("Error while remapping Communication BRAM addresses\n");
        return -2;
    }

    if (request_mem_region(AXI_MB_INTERRUPT_REQ_REG, 16, "mb_interrupt") == NULL)
    {
        printk("Error while requesting Microblaze interrupt register memory region\n");
        return -1;
    }

    mb_interrupt_start_remapped = ioremap(AXI_MB_INTERRUPT_REQ_REG, 16);

    if (mb_interrupt_start_remapped == NULL)
    {
        printk("Error while remapping Microblaze interrupt register addresses\n");
        return -2;
    }
    return 1;
}

int end_communication(void)
{
    iounmap(mb_interrupt_start_remapped);
    release_mem_region (AXI_MB_INTERRUPT_REQ_REG, 16);
    iounmap(bram_start_remapped);
    release_mem_region (AXI_BRAM_BASE_ADDRESS, AXI_BRAM_MEM_LENGTH);
    return 1;
}

void assert_interrupt_to_microblaze()
{
    iowrite32(0x01,mb_interrupt_start_remapped);
}

void deassert_interrupt_to_microblaze()
{
    iowrite32(0x00,mb_interrupt_start_remapped);
}

void send_message_to_microblaze(unsigned int *data)
{
    iowrite32(*data, bram_start_remapped + TX_LOCATION_OFFSET);
    assert_interrupt_to_microblaze();
}

void read_message_from_microblaze(unsigned int *data)
{
    *data = ioread32(bram_start_remapped + RX_LOCATION_OFFSET);
}
```

A.3 Kod aplikacije za Linux

U `communication.h` i `communication.c` su definisane funkcije za komunikaciju sa drajverom, tj. za čitanje poruke, slanje poruke i čitanje statusa. U `communication.h` definisane su i poruke koje aplikacija obrađuje i šalje.

communication.h

```
1  #ifndef COMMUNICATION_H_
2  #define COMMUNICATION_H_
3
4  // major number of driver
5  #define MAJOR_NUMBER                247
6
7  #define DEVICE_FILE_NAME            "/dev/mb_communication_driver"
8
9  #define SUCCESS                     0
10
11 // IOCTL
12 #define SEND_COMMAND          _IOW(MAJOR_NUMBER, 1, unsigned int)
13 #define RECEIVE_STATUS        _IOR(MAJOR_NUMBER, 2, unsigned int)
14 #define RECEIVE_ACTION        _IOR(MAJOR_NUMBER, 3, unsigned int)
15
16 #define TAST1_PRESSED    0x1111
17 #define TAST2_PRESSED    0x2222
18 #define TAST3_PRESSED    0x3333
19 #define DONE_ACK         0x5555
20 #define ERROR_ACK        0xAAAA
21
22 //sending message macros
23 /*
24 15          8 7          0
25 ---------------------------
26 |    LEDx    |   ACTION   |
27 ---------------------------
28 */
29 #define LED_MASK     0xFF00
30 #define LED0         0x1000
31 #define LED1         0x1100
32 #define LED2         0x1200
33 #define LED3         0x1300
34
35 #define ACTION_MASK 0x00FF
36 #define TURN_ON      0x11
37 #define TURN_OFF     0x22
38 #define TOGGLE       0x33
39
40 #define IOCTL_ERROR          -1
41 #define NO_NEW_MESSAGE       0x3
42 #define WRONG_MESSAGE        -4
43
44 #endif // COMMUNICATION_H_
```

communication.c

```c
#include <stdio.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include "defines.h"
#include "communication.h"

int driver_receive_action()
{
    int fd = open(DEVICE_FILE_NAME, O_RDWR);
    unsigned int data = 0;
    int ret = 0;

    if (fd == -1) {
        printf("ERROR: Cannot open device file!\n");
        return -1;
    }
    ret = ioctl(fd, RECEIVE_ACTION, &data);
    if (ret == IOCTL_ERROR) {
        close (fd);
        if (data = NO_NEW_MESSAGE)
            return NO_NEW_MESSAGE;
        else
            return IOCTL_ERROR;
    }
    else if (data != TAST1_PRESSED && data != TAST2_PRESSED && data != TAST3_PRESSED) {
        close (fd);
        return WRONG_MESSAGE;
    }
    close (fd);
    return data;
}

int driver_receive_status()
{
    int fd = open(DEVICE_FILE_NAME, O_RDWR);
    unsigned int data = 0;
    int ret = 0;

    if (fd == -1) {
        printf("ERROR: Cannot open device file!\n");
        return -1;
    }
    ret = ioctl(fd, RECEIVE_ACTION, &data);
    if (ret == IOCTL_ERROR) {
        close (fd);
        if (data = NO_NEW_MESSAGE)
            return NO_NEW_MESSAGE;
        else
            return IOCTL_ERROR;
    }
    else if (data != DONE_ACK && data != ERROR_ACK) {
        close (fd);
        return WRONG_MESSAGE;
    }
    close (fd);
    return data;
}

int driver_send_message(unsigned int message)
{
    int fd = open(DEVICE_FILE_NAME, O_RDWR);
    int ret = 0;

    if (fd == -1) {
        printf("ERROR: Cannot open device file!\n");
        return -1;
    }
    ret = ioctl(fd, SEND_COMMAND, &message);
    if (ret == IOCTL_ERROR) {
        close (fd);
        return IOCTL_ERROR;
    }
    close (fd);
    return 0;
}
```

Glavni program aplikacije stalno pristupa drajveru i ispituje da li je stigla poruka od MicroBlaze podsistema. Ako jeste, onda se donosi odluka o tome koja komandna poruka se šalje kao odgovor i na kraju se čeka na statusnu poruku.

main.c

```
#include "communication.h"

int main() {
    char c;
    unsigned int received_message = 0;
    unsigned int received_status = 0;
    unsigned int retval = 0;
    printf("-----------------------------------------------------\n");
    printf("Interprocessor communication test application\n\
- Communication between Dual core ARM Cortex A-9 and MicroBlaze processors\n\
- Platform: ZYNQ-7000 SoC zc706 Evaluation Board\n");
    printf("-----------------------------------------------------\n");
    printf("author: Vladimir Petrovic\n\
- for the needs of diploma thesis\n");
    printf("-----------------------------------------------------\n");
    printf("Press ENTER to start application...\n");
    while (getchar() != '\n');
    printf("Aplication started...\n");

    printf("Please press one of GPIO pushbuttons on the board.\n\n");

    while(1) {  //wait for message
        received_message = driver_receive_action();
        if (received_message == IOCTL_ERROR) {
            printf("Application will end now...\n");
            break;
        }
        else if (received_message == WRONG_MESSAGE) {
            printf("Wrong message received from MicroBlaze!\n");
            printf("Application will end now...\n");
            break;
        }
        else if (received_message == NO_NEW_MESSAGE)
            continue;
        else        //send command message
        {
            if (received_message == TAST1_PRESSED) {
                printf("Received message from MicroBlaze. Message code: 0x%x  - pressed ↩
                    taster 1\n", received_message);
                retval = driver_send_message(LED1 | TURN_ON);
                printf("Sending message to MicroBlaze. Turn on LED1  - Message code: 0x%x\n",↩
                     LED1 | TURN_ON);
                if (retval == IOCTL_ERROR)
                {
                    printf("Application will end now...\n");
                    break;
                }
            } else if (received_message == TAST2_PRESSED) {
                printf("Received message from MicroBlaze. Message code: 0x%x  - pressed ↩
                    taster 2\n", received_message);
                retval = driver_send_message(LED2 | TOGGLE);
                printf("Sending message to MicroBlaze. Toggle LED2  - Message code: 0x%x\n", ↩
                    LED2 | TOGGLE);
                if (retval == IOCTL_ERROR)
                {
                    printf("Application will end now...\n");
                    break;
                }
            } else if (received_message == TAST3_PRESSED) {
                printf("Received message from MicroBlaze. Message code: 0x%x  - pressed ↩
                    taster 3\n", received_message);
                retval = driver_send_message(LED3 | TOGGLE);
                printf("Sending message to MicroBlaze. Toggle LED3  - Message code: 0x%x\n", ↩
                    LED3 | TOGGLE);
                if (retval == IOCTL_ERROR)
                {
                    printf("Application will end now...\n");
                    break;
```

```
63                  }
64              }
65              while (1) { //wait for status
66                  received_status = driver_receive_status();
67                  if (received_status == IOCTL_ERROR)
68                  {
69                      printf("Application will end now...\n");
70                      break;
71                  }
72                  else if (received_status == WRONG_MESSAGE)
73                  {
74                      printf("Unknown status received from MicroBlaze!\n");
75                      printf("Application will end now...\n");
76                      break;
77                  }
78                  else if (received_message == NO_NEW_MESSAGE)
79                  {
80                      break;
81                  } else if (received_status == DONE_ACK)
82                  {
83                      printf("Action succesfully done! DONE ackonowledge arrived from ↩
                          MicroBlaze. \n\n");
84                      break;
85                  } else if (received_status == ERROR_ACK)
86                  {
87                      printf("Error happened during the action at MicroBlaze. ERROR ↩
                          ackonowledge arrived from MicroBlaze.\n");
88                      break;
89                  }
90              }
91              if (received_status == IOCTL_ERROR || received_status == WRONG_MESSAGE) break;
92          }
93      }
94  }
```

Dodatak B

Uputstvo za korišćenje Xilinx-ovih alata za dizajn hardvera i softvera, instaliranje Linux-a i pokretanje sistema

U ovom dodatku je dato uputstvo za dizajn hardvera uz pomoć IP blokova u programskom paketu Vivado Design Suite, eksportovanje tog dizajna u određeni format na osnovu koga se generišu Board Support Package fajlovi za potrebe pisanja sofvera. Dato je i uputstvo za korišćenje Xilinx Software Development Kit programskog paketa za kreiranje aplikacija, boot loader-a, generisanje *device tree source* fajla (.dts) kao i postupak kompajliranja u-boot-a, Linux kernela i pokretanja celog sistema. Sav razvojni softver je instaliran na Ubuntu 12.04 distribuciji Linux-a.

B.1 Uputstvo za korišćenje Xilinx Vivado Design Suite okruženja

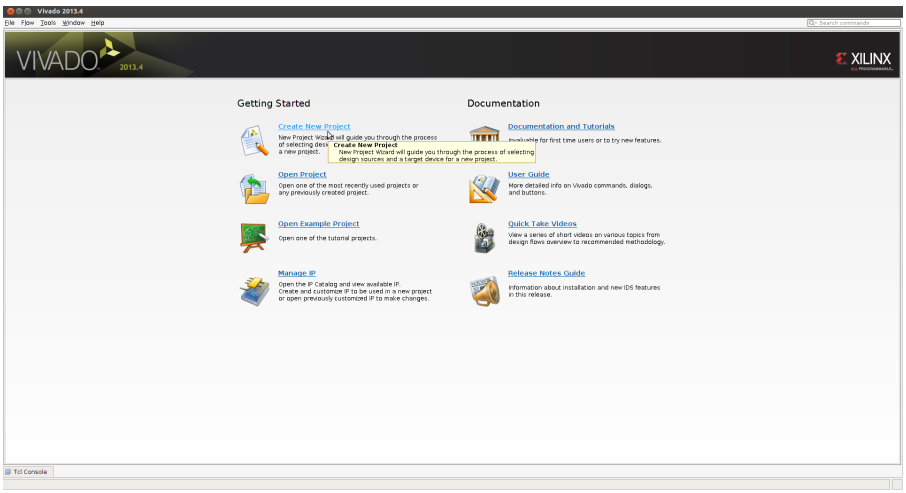
Vivado Design Suite razvojno okruženje omogućava korišćenje IP blokova u dizajnu sistema. U ovom radu su korišćeni konfigurabilni IP blokovi iz IP kataloga. U ovom uputstvu će korak po korak biti objašnjen postupak projektovanja hardvera korišćenjem IP blokova, počev od učitavanja, podešavanja i povezivanja IP blokova do generisanja bitstream fajla za programiranje FPGA. Uputstvo je bazirano na Vivado Design Suite verziji 2013.4 i razvojnoj ploči opisanoj u glavi 2 ovog rada.

Pokretanjem Vivado Design Suite grafičkog korisničkog interfejsa otvara se prozor kao na slici B.1. Klikom na **Create New Project** otvara se prozor za kreiranje novog projekta.

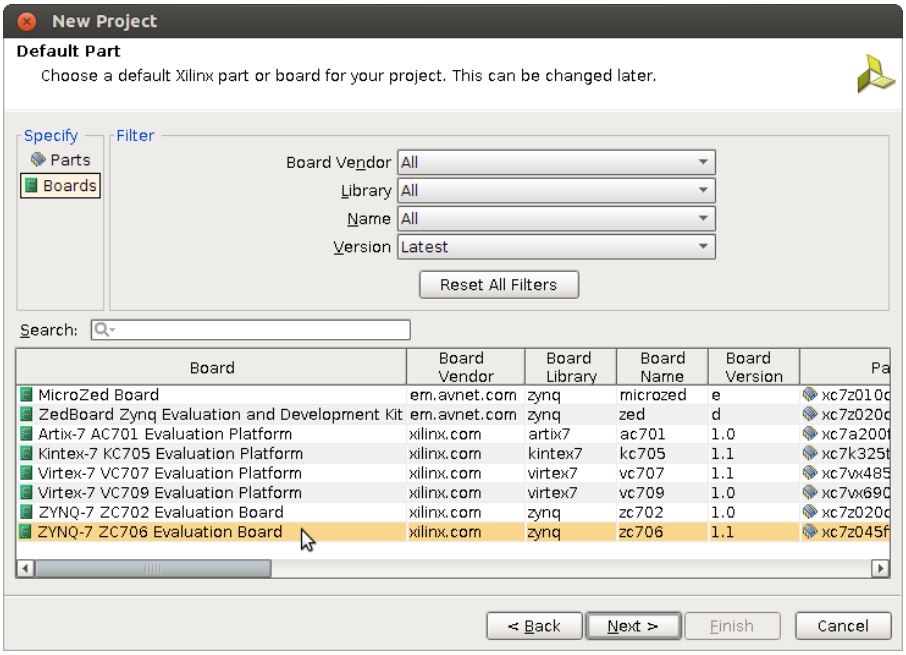
Odabirom imena projekta i lokacije projekta i klikom na **Next** prelazi se na odabir tipa projekta. Ovde treba odabrati RTL projekat sa selektovanom opcijom **Do not specify sources at this time**. Klikom na **Next** otvara se prozor za podešavanje platforme kao na slici B.2. U odeljku **Boards** bira se razvojna ploča čime je i podešen čip. Klikom na **Next** pa **Finish** kreira se novi projekat.

Novi dizajn se kreira klikom na **Create New Block Design** kao što je prikazano na slici B.3.

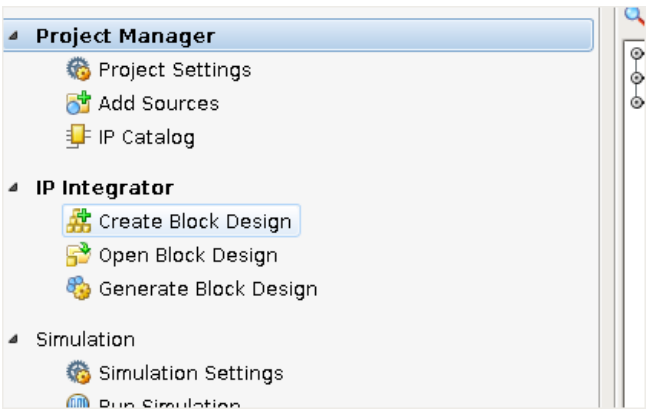
Kreirani dizajn je prazan kao što je prikazano na slici B.4.



Slika B.1: Vivado početni prozor

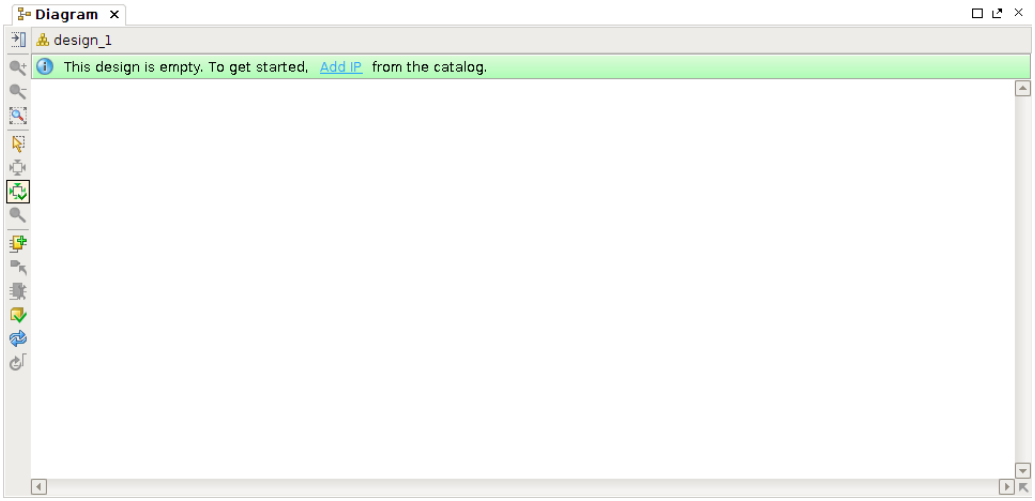


Slika B.2: Odabir hardverske platforme



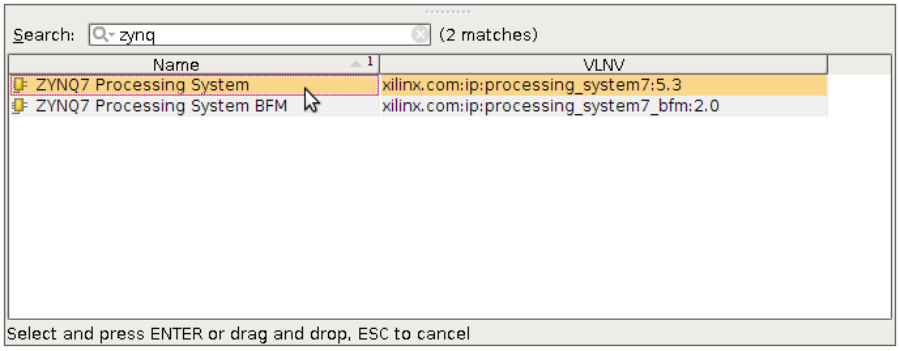
Slika B.3: Kreirani projekat, kreiranje novog blok dizajna

Prvi IP blok koji ćemo dodati je ZYNQ PS koji predstavlja ASIC deo čipa sa svim interfejsima i podešavanjima. Klikom na *Add IP* u okviru blok dizajna otvara se prozor u kome se

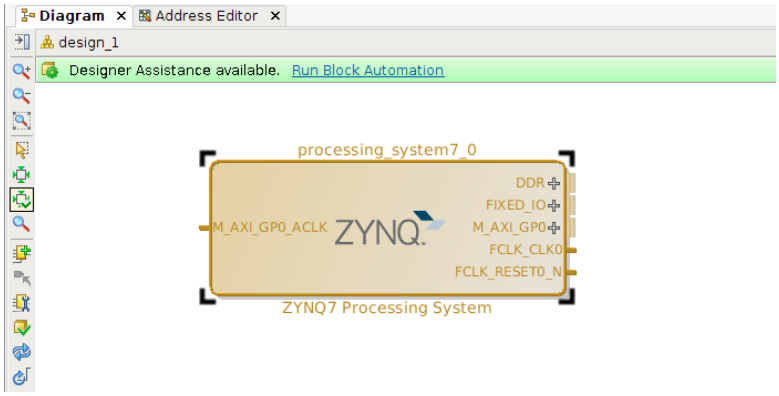


Slika B.4: Kreirani blok dizajn

pretragom može naći željeni IP blok (slike B.5 i B.6).

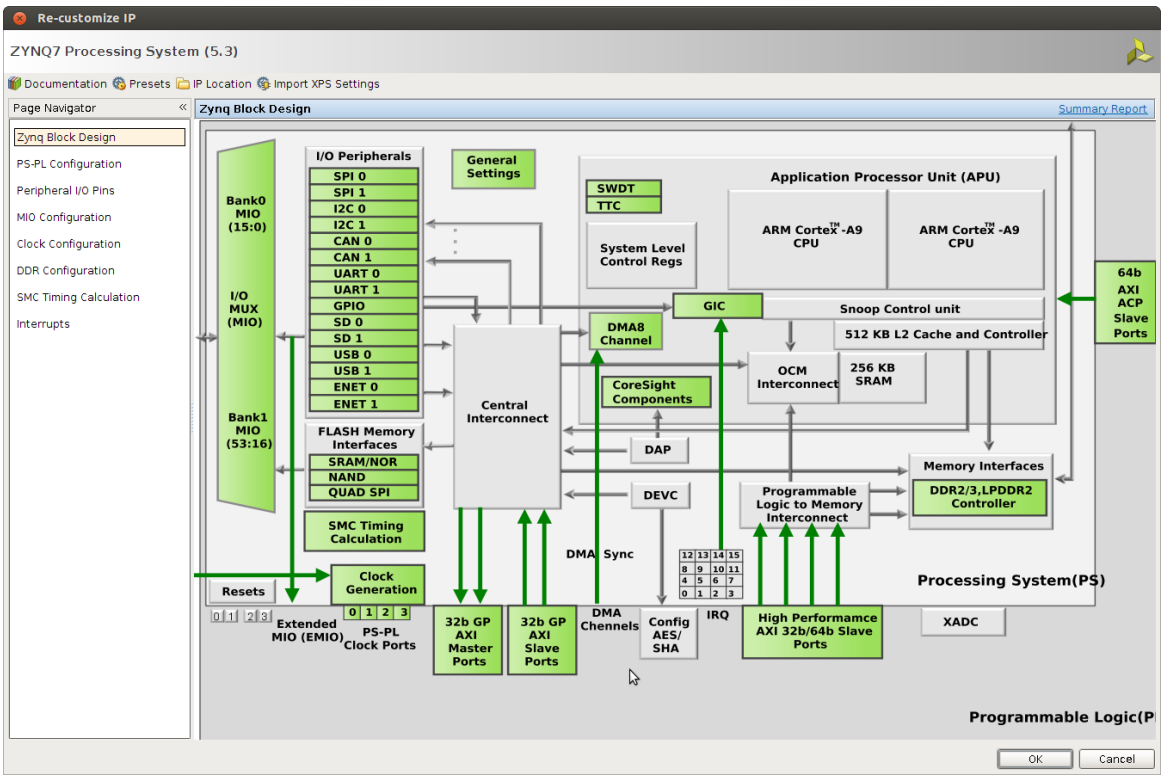


Slika B.5: Dodavanje IP bloka u sistem

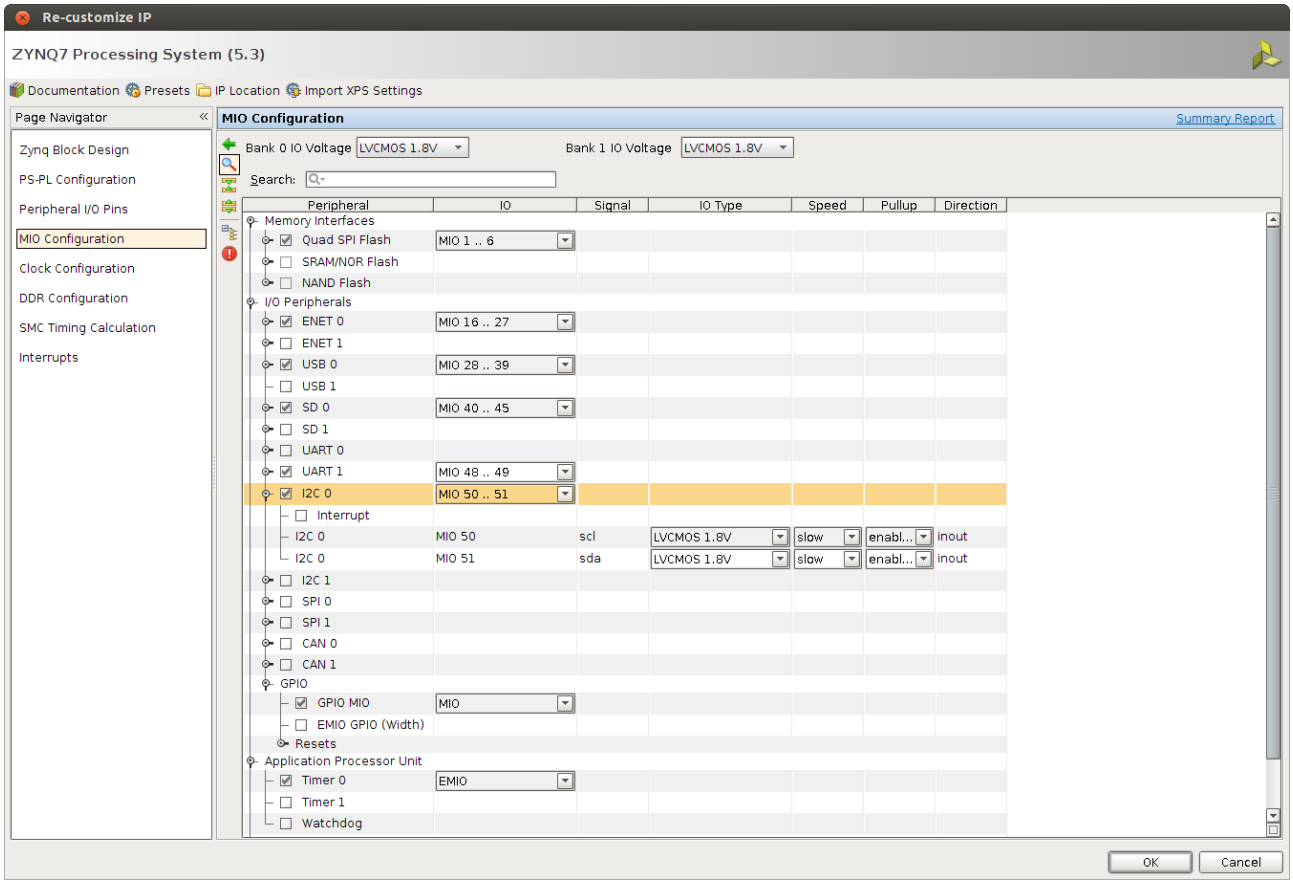


Slika B.6: Dodati IP blok

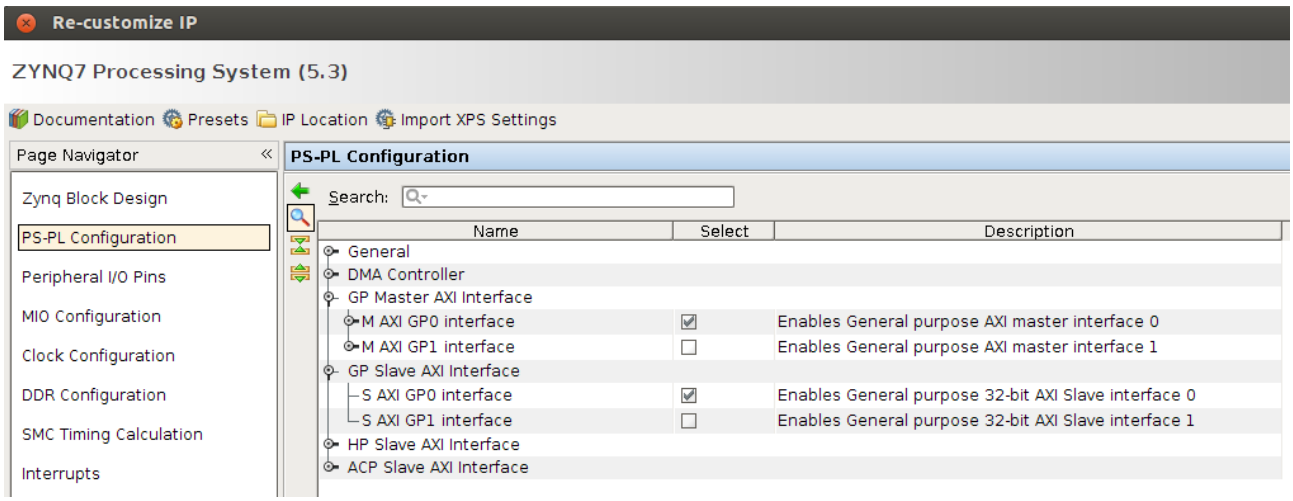
Podešavanja IP bloka se otvaraju duplim klikom na konkretan IP blok. Podešavanja su za svaki IP blok različita i ovde će se naglasiti samo neka od važnih podešavanja. Na slici B.7 prikazan je prozor podešavanja ZYNQ-7000 PS-a. Duplim klikom na neku od periferija, otvaraju se podešavanja sa slike B.8. Na slikama B.9 i B.10 prikazana su podešavanja AXI interfejsa i interfejsa za prekide od programabilne logike.



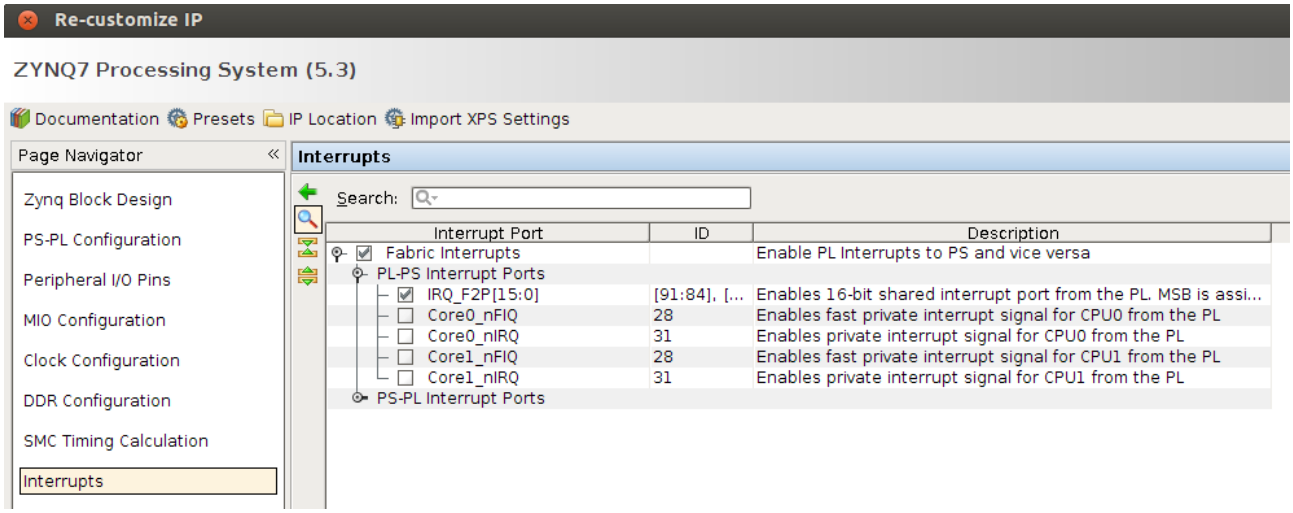
Slika B.7: Prozor podešavanja ZYNQ PS-a 1



Slika B.8: Prozor podešavanja ZYNQ PS-a 2

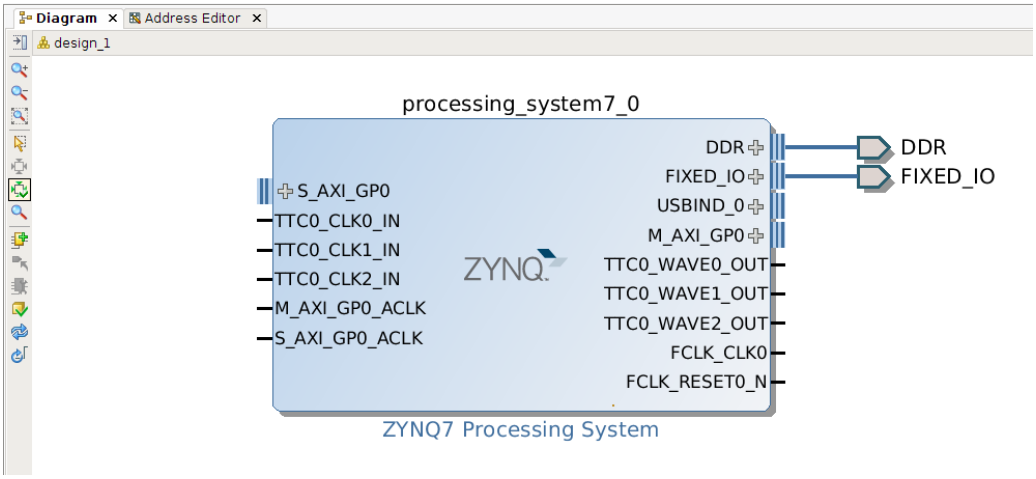


Slika B.9: Prozor podešavanja ZYNQ PS-a 3



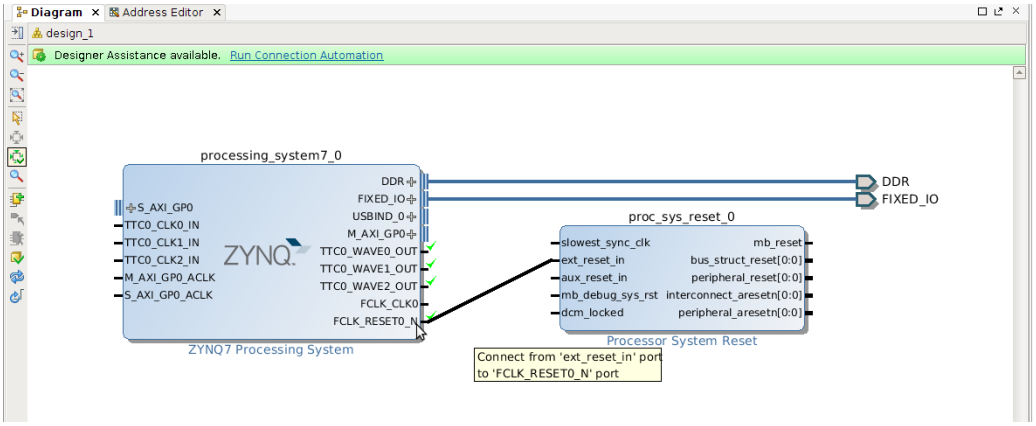
Slika B.10: Prozor podešavanja ZYNQ PS-a 4

Klikom na **Run Block Automation** se automatski generiše interfejs ka DDR memoriji i interfejs za sve fiksne pinove kao na slici B.11.



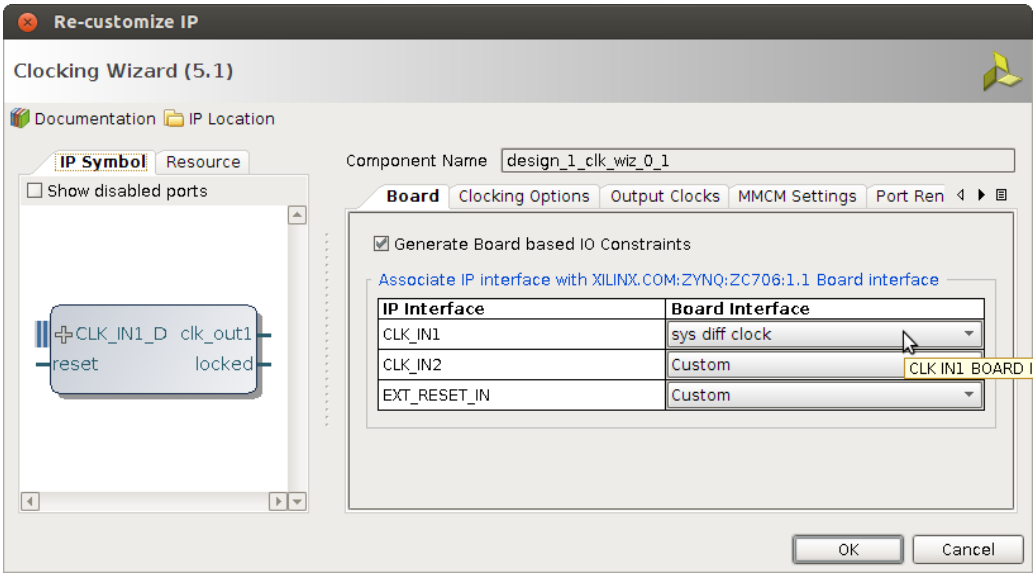
Slika B.11: ZYNQ PS posle podešavanja

Za resetne signale koristi se IP blok *Processor System Reset* čiji ulaz *ext_reset_in* se dovodi na izlaz *FCLK_RESET0_N* ZYNQ PS-a, a izlazni signali se vode kao resetni signali za MicroBlaze procesor, magistrale i druge periferije (slika B.12). Signal *peripheral_reset* se vodi na sve periferije koje ne specificiraju drugačiji resetni signal. Npr. glavni reset *AXI Interconnect* blokova se vezuje na *bus_struct_reset* signal. *mb_reset* je resetni signal za MicroBlaze procesor, dok je *mb_debug_sys_reset* resetni signal za MicroBlaze debug modul.



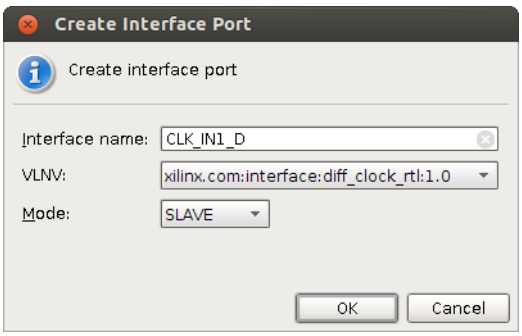
Slika B.12: Povezivanje IP bloka za kontrolu reseta

Signal takta se dovodi iz IP bloka *Clocking Wizard* koja se povezuje na eksterni izvor signala takta. Na slici B.13 je prikazano podešavanje ulaznog taktnog signala.



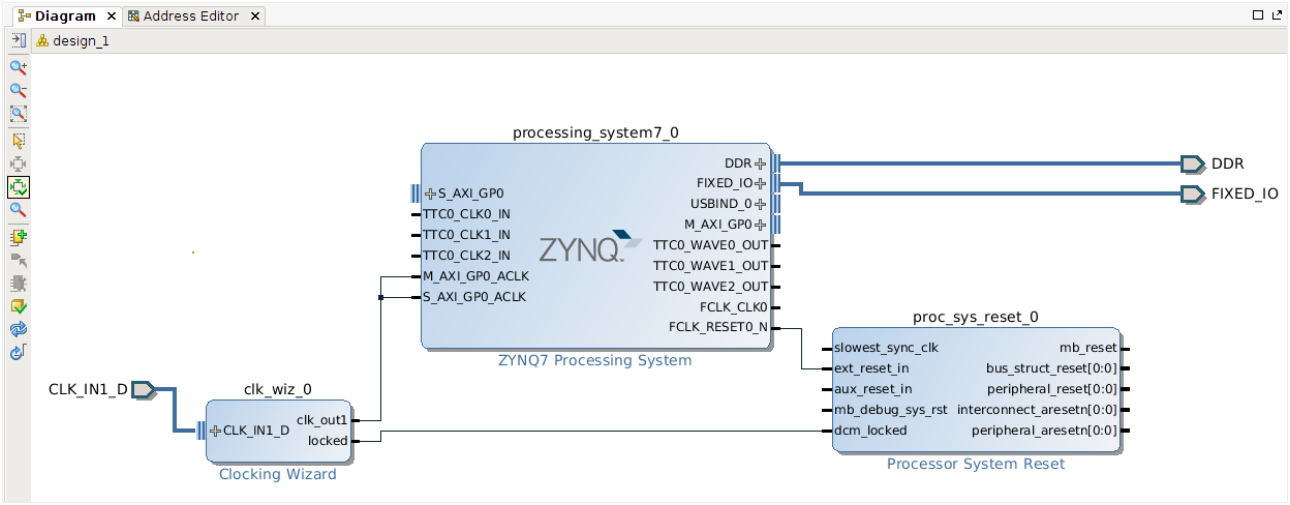
Slika B.13: Podešavanje *Clock Wizard* IP bloka

Pošto se signal takta dovodi spolja potrebno je kreirati port i kasnije nakon implementacije taj signal povezati na odgovarajući pin na čipu. Za signal takta treba kreirati interfejs što se postiže desnim klikom na block dizajn i klikom na **Create Interface Port**. Otvara se prozor sa slike B.14. Na slici su prikazana podešavanja koja je potrebno izvršiti da bi se ispravno kreirao interfejs za signal takta.



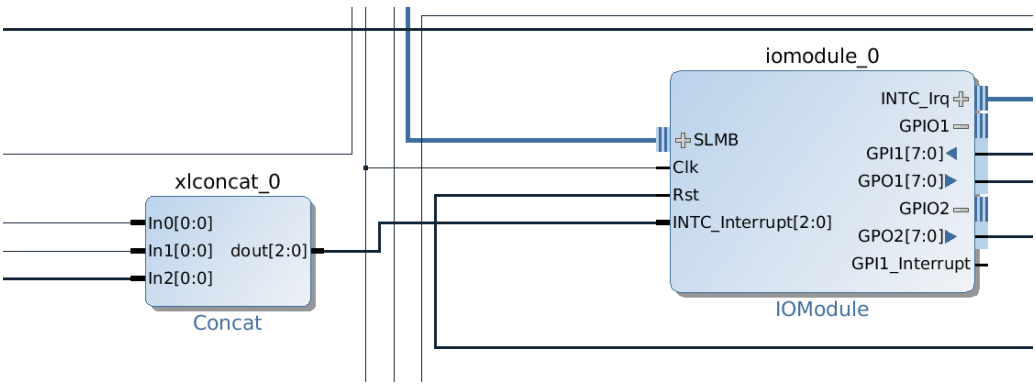
Slika B.14: Podešavanje interfejsa za signal takta

Blok dizajn nakon ovih podešavanja izgleda kao na slici B.15.



Slika B.15: Blok dizajn nakon inicijalnih podešavanja

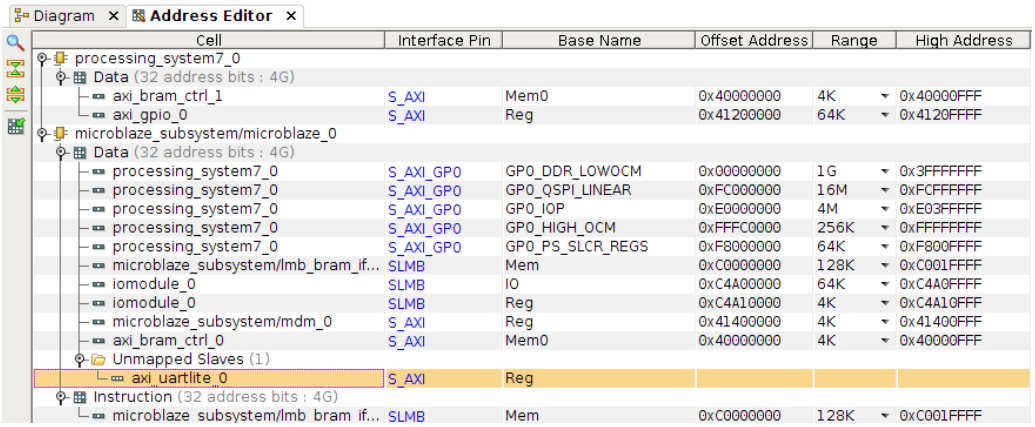
Port za GPIO izlaze se dodaje desnim klikom na block dizajn i klikom na **Create Port** i može se podešavati broj bita. Za povezivanje više signala na ulaze IP blokova koji su obeleženi kao vektori, a koji nisu standardni interfejsi (AXI, LMB i sl.) potrebno je prvo te signale spojiti u vektor IP blokom *Concat*, a zatim izlazni signal povezati na vektorski ulaz drugog IP bloka. Na slici B.16 je prikazan primer povezivanja tri signala u jedan vektor koji se vodi na ulaz za prekidne zahteve IP bloka *I/O Module*. Za rastavljanje vektora na više manjih vektora ili signala koristi se IP blok *Slice*.



Slika B.16: Povezivanje više signala u jedan vektor

Kao što je opisano u glavi 3, AXI interkonekcijski blokovi se koriste za realizaciju AXI interfejsa. Svakom portu bilo da je to SLAVE ili MASTER potrebno je dovesti signal takta i resetni signal. Resetni signal je *peripheral_ aresetn* iz *Processor System Reset* IP bloka.

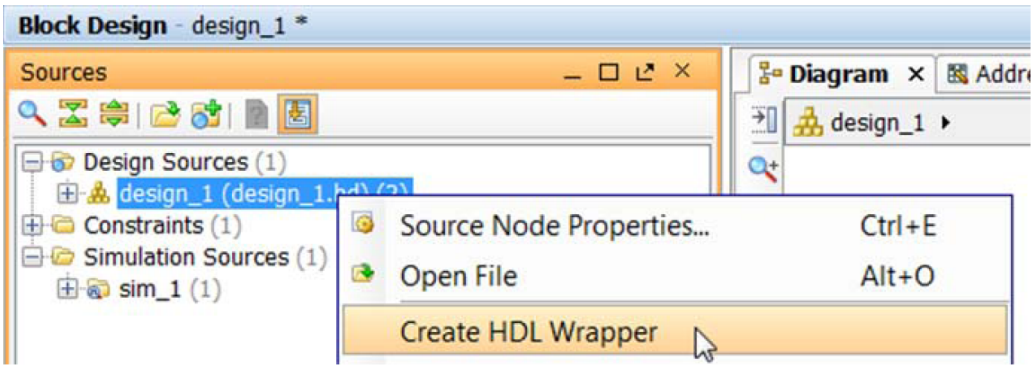
Nakon što je završen dizajn hardvera uz pomoć IP blokova potrebno je podesiti adrese periferija koje vide procesori u sistemu. Klikom na polje **Address Editor** padajućeg menija **Window** otvara se adresni editor gde se mogu podesiti memorijske mape za procesore (slika B.17). Ako postoje periferije kojima nisu dodeljene adrese (*Unmapped slaves*) potrebno im je dodeliti adrese što se može uraditi desnim klikom na komponentu i odabirom opcije **Assign address**. Može se odabrati i opcija **Auto Assign Address** čime se automatski dodeljuju slobodne adrese svim nemapiranim periferijama.



Slika B.17: Podešavanje adresa

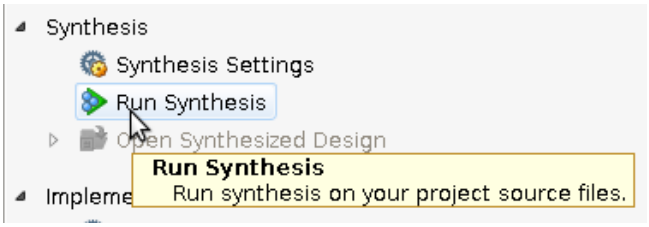
Pre sinteze i implementacije potrebno je proveriti da li je dizajn spreman za sintezu. To se postiže desnim klikom na blok i dizajn i odabirom opcije **Validate Design**. Ukoliko postoji neka greška, pojaviće se prozor sa obaveštenjem koje greške su u pitanju.

Ostaje još da se kreira HDL wrapper i nakon toga se prelazi na sintezu. Kreiranje HDL wrapper-a prikazano je na slici B.19.



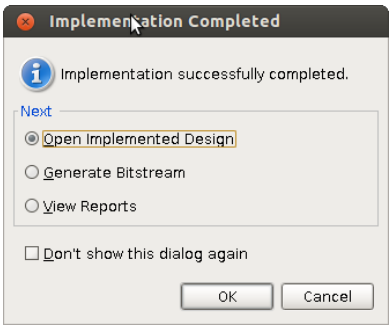
Slika B.18: Kreiranje HDL wrapper-a

Za pokretanje sinteze treba kliknuti na Run Synthesis u Flow Navigator prozoru, a nakon toga i Run Implementation za pokretanje implementacije.



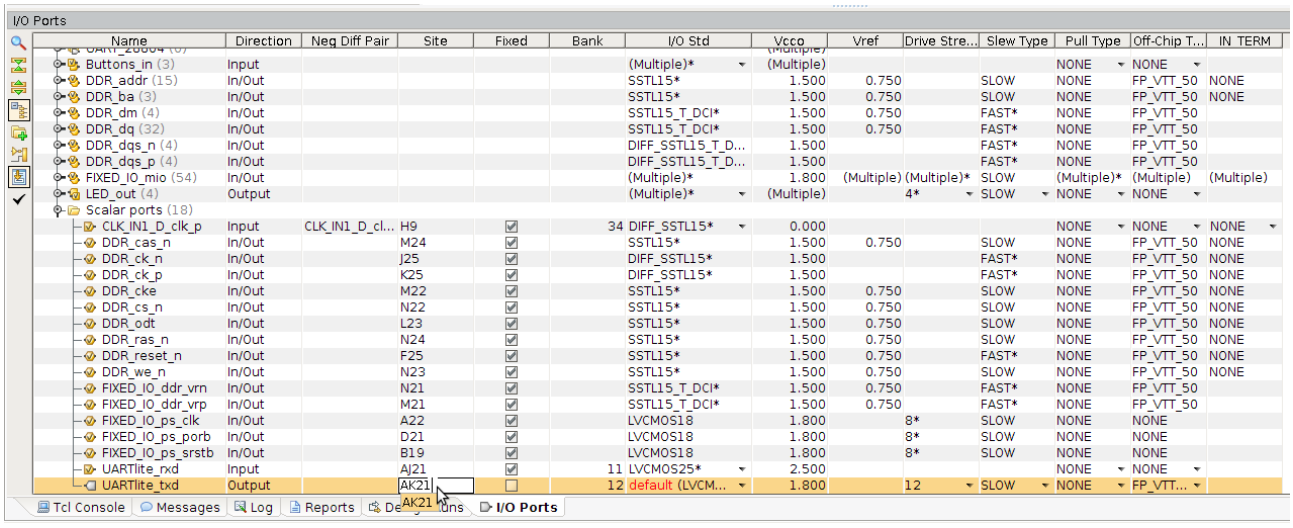
Slika B.19: Pokretanje sinteze

Nakon završene implementacije potrebno je povezati ulazne i izlazne signale na odgovarajuće portove, ako nisu već opisani u *contraint* .xdc fajlu. Nakon implementacije treba prvo otvoriti implementirani dizajn (slika B.20), a zatim otvoriti prozor namenjen automatskom generisanju constraint fajla klikom na **I/O Ports** polje iz **Window** padajućeg menija (slika B.21).



Slika B.20: Završetak implementacije

Na slici se vidi da portovi za UART nisu podešeni, pa se odabirom odgovarujeg pina i I/O standarda (LVCMOS25) podešava da *Uartlite* periferija bude izvedena na pinove PMOD interfejsa kao što je to ranije u radu opisano.

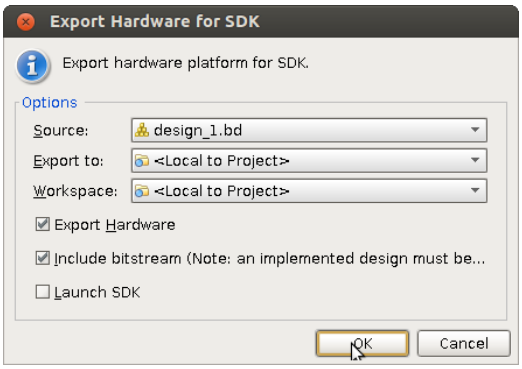


Slika B.21: Podešavanje ulaznih i izlaznih pinova

Ovakvo podešavanje će generisati sledeće linije u *constraint* .xdc fajlu:

```
set_property PACKAGE_PIN AJ21 [get_ports UARTlite_rxd]
set_property IOSTANDARD LVCMOS25 [get_ports UARTlite_rxd]
set_property IOSTANDARD LVCMOS25 [get_ports UARTlite_txd]
set_property PACKAGE_PIN AK21 [get_ports UARTlite_txd]
```

Nakon ovog podešavanja, klikom na **Generate Bitsream**, generiše se bitsream fajl za programiranje FPGA. Postoji mogućnost da će Vivado dati upozorenje da je potrebno uraditi ponovo implementaciju. Nakon uspešnog generisanja bitstream fajla, potrebno je eksportovati hardver dizajn za dizajn softvera. Na osnovu fajlova koji se ovde generišu, Xilinx SDK generiše Board Support Package za konkretne aplikacije. Klikom na **File->Export->Export Hardware for SDK** otvara se prozor kao na slici B.22.



Eksportovanje potrebnih fajlova sa dizajn softvera

Ovim je završena sinteza hardvera i može se preći na dizajn softvera u Xilinx SDK razvojnom okruženju.

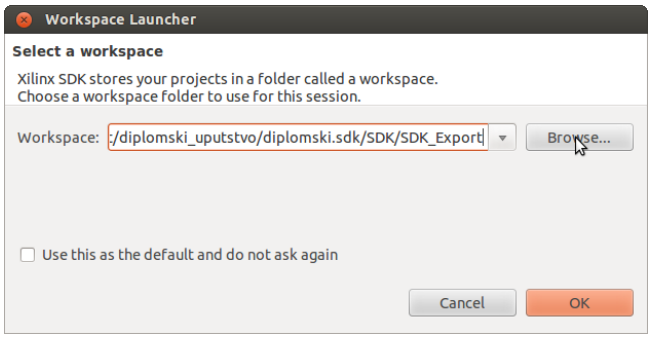
Postoji veliki broj uputstava za Vivado razvojno okruženje i sva su dostupna. Najlakše se pretražuju pod skraćenim identifikatorima, a neka od njih su: ug895, ug989, ug904, ug911, ug939, ug940 u kojima se može naći većina opisanih funkcionalnosti. Xilinx pruža i video tutorijale za rad u svim alatima koji se mogu naći na Xilinx-ovom sajtu.

B.2 Kreiranje BSP-a za dizajnirani hardver, FSBL-a, devicetree fajla, kompajliranje u-boot-a, Linux-a i priprema SD kartice za podizanje sistema

Nakon eksportovanja hardvera opisanog u prethodnoj glavi u direktorijumu projekta kreiranog u Vivado Design Suite okruženju pojaviće se direktorijum pod nazivom `ime_projekta.sdk`, a u njemu direktorijumi `SDK/SDK_Export/hw` i u `hw` direktorijumu se nalazi nekoliko fajlova. Jedan od fajlova je bitstream fajl za programiranje hardvera (.bit). Drugi je Block RAM Memory Map (.bmm) fajl koji Data2MEM program koristi za inicijalizaciju programske blok RAM memorije. Ostali fajlovi su potrebni za kreiranje FSBL-a i projekta hardverske specifikacije koji je neophodan za sve ostale projekte.

B.2.1 Kreiranje FSBL-a i aplikacije za MicroBlaze

Pokretanjem Xilinx SDK okruženja otvoriće se prozor u kome treba uneti putanju do workspace direktorijuma. Važno je napomenuti da putanja ne sme da sadrži razmake i u našem slučaju je: `direktorijum_projekta/ime_projekta.sdk/SDK/SDK_Export`.



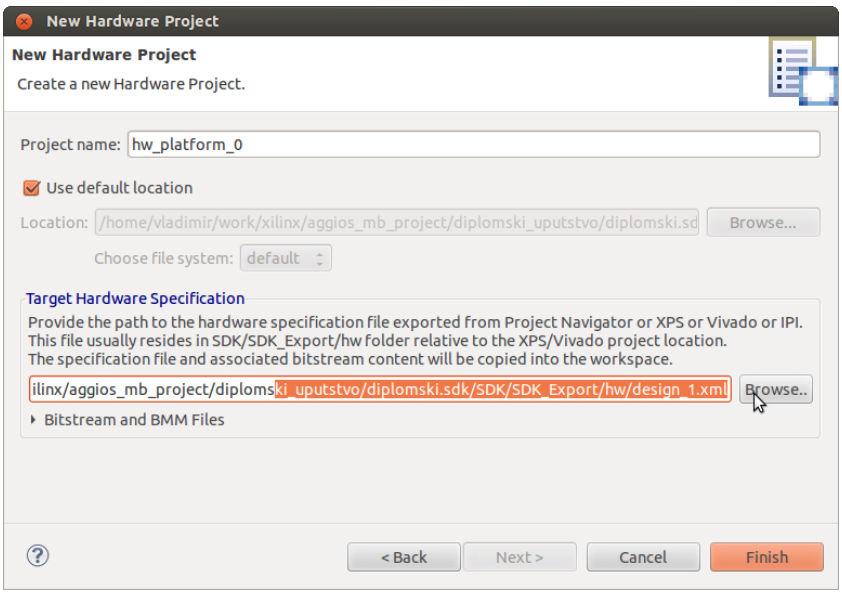
Slika B.23: Podešavanje workspace putanje

Klikom na **OK** kreira se novi workspace u SDK. Najpre treba kreirati projekat hardverske specifikacije i to otvaranjem prozora za novi projekat sa **File->New->Project**. Odabrati **Xilinx** pa **Hardware Platform Specification** i kliknuti na **Next**. Otvara se prozor kao sa slike B.24. Za *Target Hardware Specification* treba odabrati `design_1.xml` (ili kako je već ime dizajna iz Vivado okruženja) iz već pomenutog `hw` direktorijuma. Kliknuti na **Finish**.

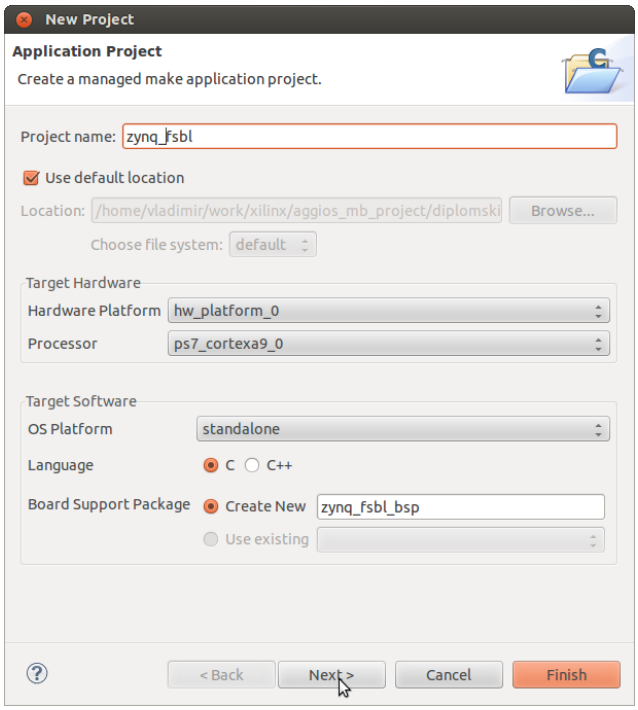
Zatim se kreira *First Stage Boot Loader (FSBL)*. **File->New->Application Project** otvara prozor za kreiranje nove aplikacije. Treba izvršiti podešavanja kao na slici B.25, Za *hardware platform* treba selektovati platformu koja je kreirana u prethodnom koraku. Klikom na **Next** i odabirom **Zynq FSBL** templejta kreira se FSBL projekat za specificiranu hardversku platformu.

Nakon FSBL-a kreira se BSP projekat za MicroBlaze podsistem. **File->New->Board Support Package** otvara se prozor za kreiranje BSP-a kao na slici B.26.

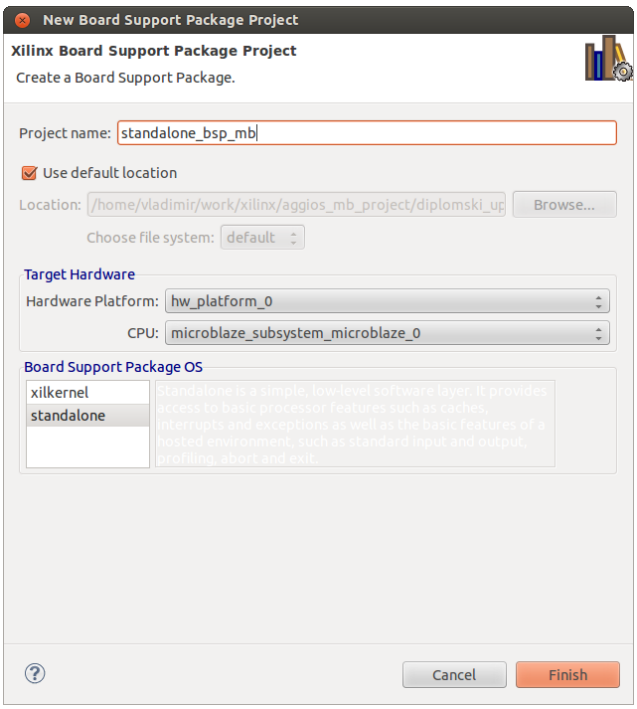
Klikom na **Finish** automatski se otvara prozor za podešavanje kreiranog BSP-a. Za stdin i stdout treba odabrati željenu periferiju, tj. UART koji će se koristiti za ispis. Ponuđene su sve UART periferije koje MicroBlaze vidi u svom memorijskom prostoru. Mi ćemo odabrati `axi_uartlite_0` kao što je prikazano na slici B.27.



Slika B.24: Odabir .xml fajla



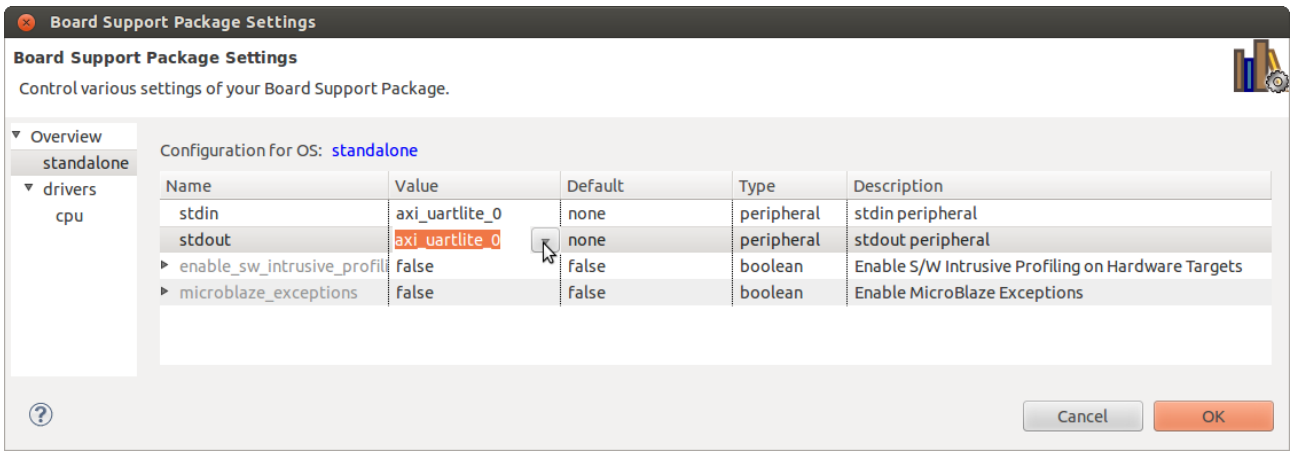
Slika B.25: Kreiranje FSBL-a



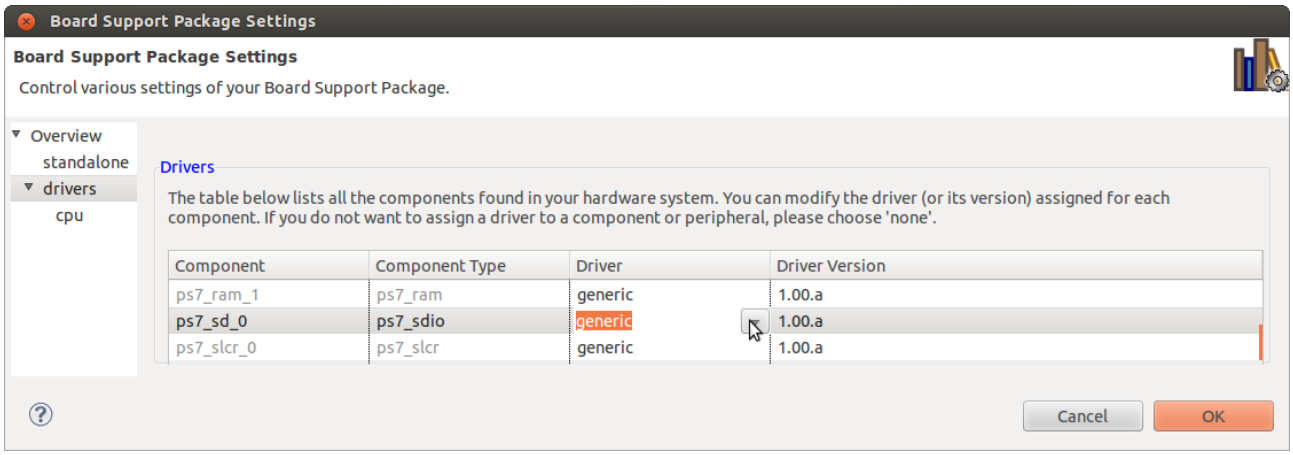
Slika B.26: Kreiranje BSP-a

Za slučaj da MicroBlaze treba da pristupa SD kontroleru, treba promeniti drajver za SD kontroler na `generic` da bi se BSP uspešno kompajlirao kao na slici B.28.

Sada je sve spremno za kreiranje aplikacije za MicroBlaze. Nova aplikacija se kreira sa **File->New->Application Project** pri čemu treba selektovati odgovarajuću hardversku platformu i procesor, a može se ubaciti i već gotova aplikacija sa **File->Import...**, a zatim **General->Existing Projects into Workspace**. Odabirom lokacije projekta i selektovanjem opcije **Copy projects into workspace** kopira se ceo projekat u trenutni workspace.



Slika B.27: Podešavanje BSP-a - podešavanje periferije za serijsku komunikaciju



Slika B.28: Podešavanje BSP-a - podešavanje drajvera za SD kontroler

Kada se završi pisanje aplikacije za MicroBlaze, treba importovati izvršni fajl u bitstream kako bi se izvršilo regularno programiranje programabilne logike. Ovo ćemo uraditi uz pomoć programa data2MEM. Iz komandne linije treba otići u `hw_platform_0` direktorijum. Ime direktorijuma zavisi od toga kako je dato ime inicijalnom *hardware specification* projektu. Iz tog direktorijuma treba pozvati program data2MEM sa sledećim argumentima:

```
data2mem -bm design_1_wrapper_bd.bmm -bt design_1_wrapper.bit \
-bd ../mb_application/Debug/mb_application.elf -o b download.bit
```

`mb_application` je ime aplikacije koje je dato projektu prilikom kreiranja. Ovim se u direktorijumu `hw_platform_0` kreira `download.bit` fajl koji predstavlja inicijalizovani bitstream fajl kojim se programira FPGA.

B.2.2 Kompajliranje U-boot-a, Linux kernela i bootimage fajla

Za pokretanje Linuxa, kao što je već napomenuto u radu, potreban je boot loader u-boot. Ovde će biti opisan postupak njegove kompilacije.

Najpre iz komandne linije podesiti cross kompajler da bude `arm-xilinx-linux-gnueabi-`. Iz direktorijuma u kome će biti u-boot fajlovi eksportovati `CROSS_COMPILE` varijablu na sledeći način:

```
export CROSS_COMPILE=arm-xilinx-linux-gnueabi-
```

Zatim podesiti PATH varijablu da sadrži putanju do cross kompajlera:

```
export PATH=xilinx instalacioni direktorijum/SDK/2013.4/gnu/arm/lin/bin/:$PATH
```

Zatim treba klonirati U-boot `git` repository sa Xilinx-ovog `git` servera:

```
git clone git://github.com/Xilinx/u-boot-xlnx.git
```

Podesiti U-boot za ZYNQ-7000 platformu:

```
cd u-boot-xlnx
make ARCH=arm zynq\_zc70x\_config
```

I na kraju kompajlirati U-boot:

```
make
```

Za generisanje BOOT.bin fajla koji je neophodan za podizanje sistema treba preimenovati `u-boot` izvršni fajl u `u-boot.elf`

```
mv u-boot u-boot.elf
```

Ovim je kompilacija U-boot-a završena.

Za kompilaciju Linux kernela je takođe potrebno podesiti kompajler kao za U-boot, pa nećemo ponavljati te korake. Klonirati Zynq Linux kernel `git` repository sa Xilinx-ovog `git` servera:

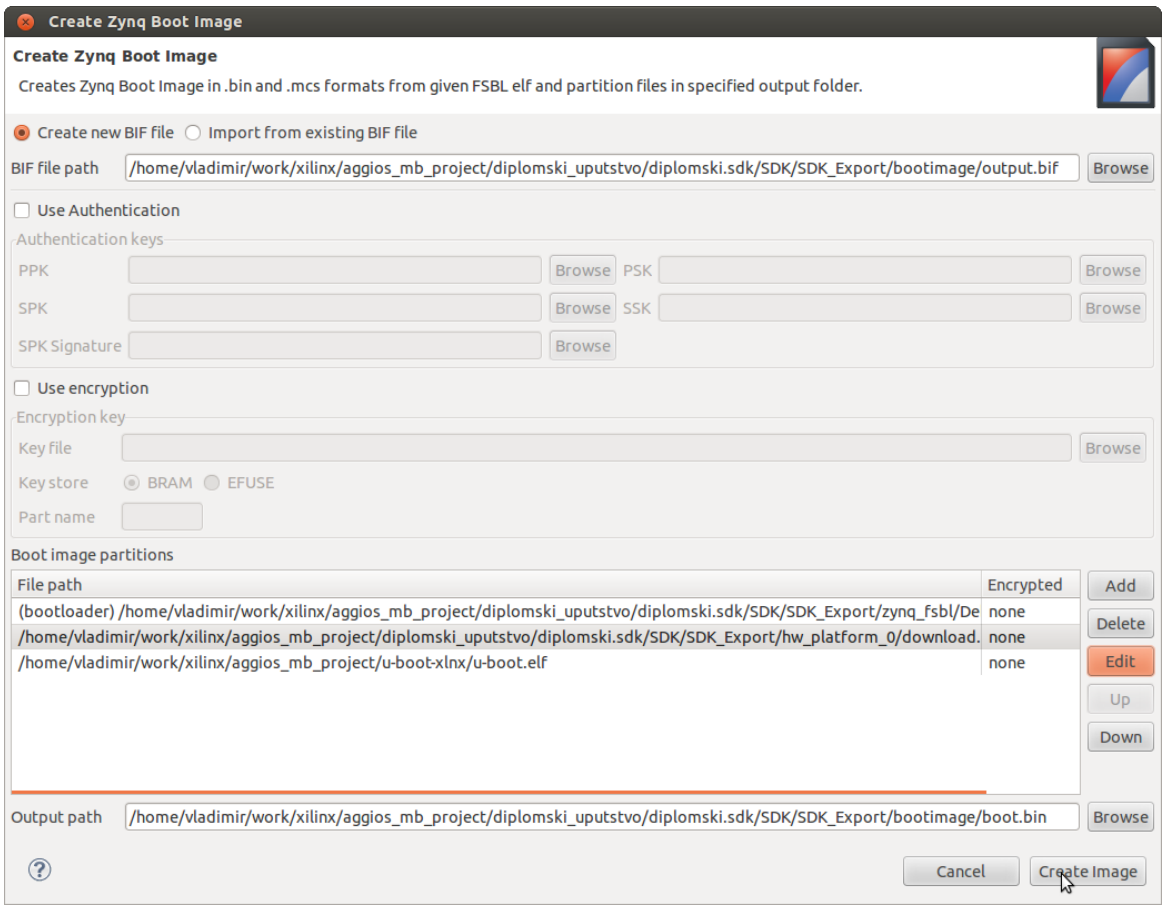
```
git clone git://github.com/Xilinx/linux-xlnx.git
```

Kompajlirati uImage:

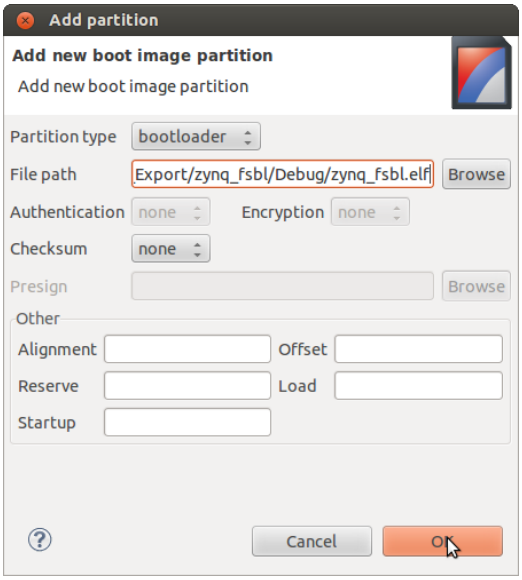
```
cd linux-xlnx
make ARCH=arm uImage UIMAGE\_LOADADDR=0x8000
```

Generisani uImage fajl se nalazi u direktorijumu: `linux-xlnx/arch/arm/boot/uImage` počev od trenutnog direktorijuma.

Preostalo je da kreiramo bootimage fajl koji je binarni fajl u kome se nalaze FSBL, bitstream za programiranje FPGA i u-boot. **Xilinx Tools->Create Zynq Boot Image** otvara prozor za kreiranje ovog fajla (slika B.29). Odabrati opciju **Create new BIF file** i podesiti putanju za taj fajl na `.../SDK/SDK_Export/bootimage`. Klikom na ADD prvo dodati FSBL, tj. `zynq_fsbl.elf` iz FSBL projekta kao bootloader (slika B.30), zatim download.bit fajl kreiran korišćenjem data2MEM programa kao datafile i na kraju `u-boot.elf` fajl, takođe kao datafile. Izlazni direktorijum je prozivoljan, ali zbog lepe organizacije projekta zgodno je da to bude `.../SDK/SDK_Export/bootimage`, a izlazni fajl treba da ima ime `boot.bin`.



Dodavanje devicetree repostory-ja u SDK workspace



Dodavanje devicetree repostory-ja u SDK workspace

B.2.3 Kreiranje devicetree.dtb fajla

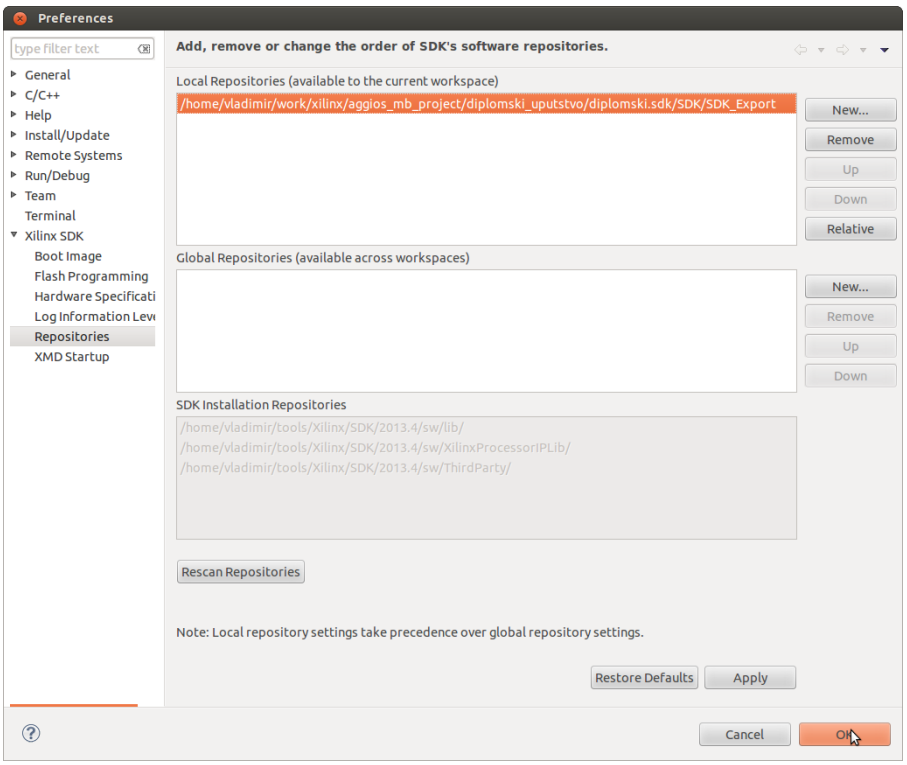
Za kreiranje devicetree fajla prvo je potrebno iz komandne linije otići u direktorijum SDK workspace-a u kome su kreirani svi ranije pomenuti projekti (`.../SDK/SDK_Export`). SDK poseduje devicetree generator koji može da generiše device tree source fajl na osnovu hardverske specifikacije. Da bi se na taj način generisao devicetree source fajl, prvo treba klonirati devicetree repository na trenutni workspace.

```
git clone git://github.com/Xilinx/device-tree.git bsp/device-tree_v0_00_x
```

i vratiti se na verziju podešenu za 2013.4 verziju SDK okruženja, kao i kreirati novi branch `nas_projekat` za sve potencijalne izmene:

```
cd bsp/device-tree_v0_00_x
git checkout -b nas_projekat xilinx-v2013.4
```

Zatim u SDK dodati klonirani repository sa **Xilinx Tools->Repositories->New**, odabrati workspace direktorijum (`.../SDK/SDK_Export`) i kliknuti na OK.

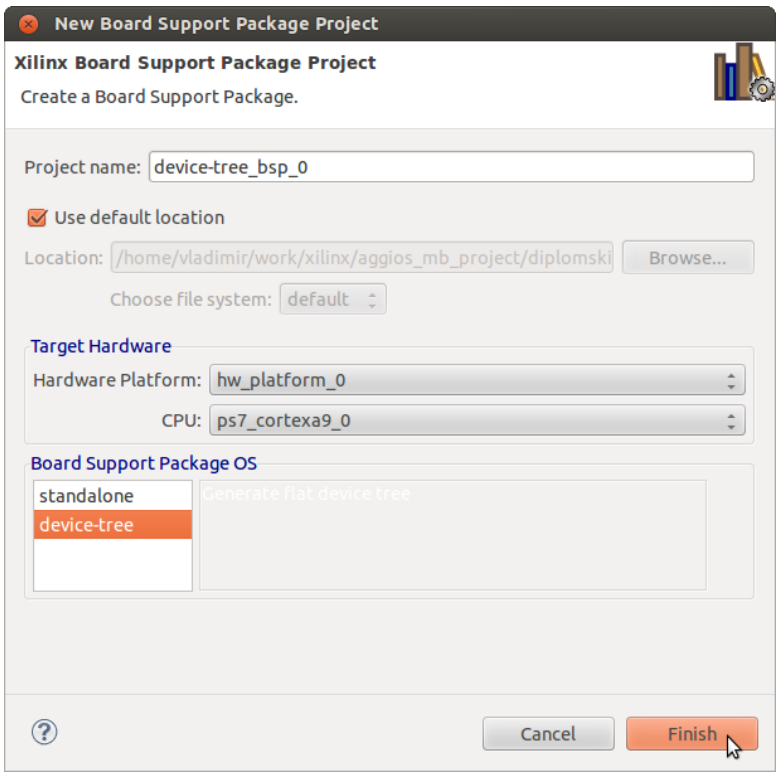


Slika B.31: Dodavanje devicetree repostory-ja u SDK workspace

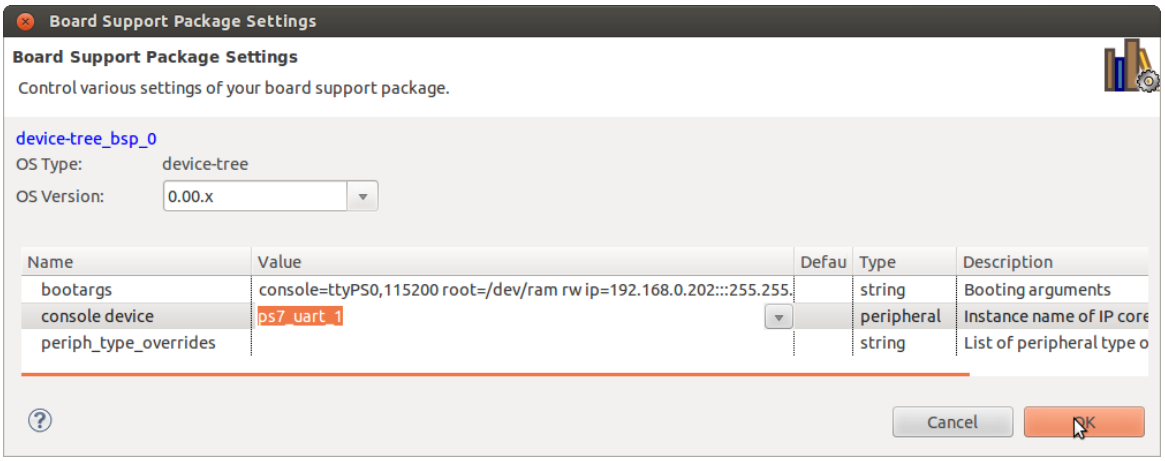
File->New->Board Support Package otvara prozor za novi BSP koji treba podesiti kao na slici B.32. Klikom na **Finish** otvara se prozor za podešavanje devicetree BSP-a. Za console-device treba odabrati `ps7_uart_1` što je UART1 iz ZYNQ PS-a, a za boot argumente treba staviti:

```
console=ttyPS0,115200 root=/dev/ram rw ip=192.168.0.202:::255.255.255.0:ZC706:eth0
earlyprintk
```

Naravno za IP adresu ploče treba staviti onu adresu koja će biti korišćena prilikom kopiranja podataka na ploču. Klikom na OK, kreiran je devicetree BSP projekat koji se automatski kompajlira.



Slika B.32: Dodavanje devicetree BSP-a



Slika B.33: Podešavanja devicetree BSP-a

Kreirani device tree source fajl (`xilinx.dts`) se nalazi u sledećem direktorijumu: `.../SDK_Export/device-tree_bsp_0/ps7_cortexa9_0/libsrc/device-tree_v0_00_x`

Ovaj fajl je potrebno preimenovati pošto će njegovo ime biti korišćeno prilikom kompajliranja .dtb (device tree blob) fajla. Kopirati preimenovani fajl u sledeći direktorijum u okviru prethodno kompajliranog Linux kernela: `linux-xlnx/arch/arm/boot/dts`. Iz `linux-xlinx` direktorijuma pozvati sledeću komandu:

```
make ARCH=arm ime_dts_fajla.dtb
```

U `linux-xlnx/arch/arm/boot/dts` se sada može naći kreirani .dtb fajl sa ispim imenom koje je dato .dts fajlu. Ovaj .dtb fajl treba preimenovati u `devicetree.dtb` i koristiti kao devicetree fajl za pokretanje Linuxa.

B.2.4 Priprema SD kartice za uspešno pokretanje sistema

Za uspešno pokretanje sistema, na kraju formatiranja SD kartice, biće nam potrebni kompajlirani Linux kernel *uImage*, `boot.bin` kreiran iz SDK i `devicetree.dtb`.

Ubaciti čitač kartice sa SD karticom u računar. Komanda `dmesg` bi trebalo da ispiše neke podatke vezane za SD katicu. U uglastim zagradama tog ispisa se nalazi `[sdX]` gde je X identifikator sd kartice, npr. 1, 2, b i sl. U narednim linijama svuda umesto sdX treba staviti ime kartice koja je učitana. Za većinu narednih komandi su potrebne administratorske dozovole, tj. `sudo` ispred svake komande.

`dd if=/dev/zero of=/dev/sdX bs=1024 count=1` briše prvi sektor SD kartice kada već postoje particije na kartici, pa je ovo neophodno da se uradi.

`fdisk -l /dev/sdX` izbacuje sledeći izlaz:

```
Disk /dev/sdX: 8068 MB, 8068792320 bytes
249 heads,62 sectors/track,1020 cylinders, total 15759360 ↩
   sectors
Units = sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disk identifier: 0x00000000

Disk /dev/sdX doesn't contain a valid partition table
```

Ovaj izlaz ćemo iskoristiti za izračunavanje broja cilindara. Broj cilindara se računa kao veličina kartice u bajtovima podeljena sa 8225280, što je za ovaj primer: 8068792320 / 8225280 = 980. Ovaj broj će biti potreban kasnije, pa ga treba zapamtiti i uneti kada fdisk program to bude zahtevao.

Sada ćemo napraviti particije na SD kartici:

`fdisk /dev/sdX` daje interfejs u kome treba kucati komande na sledeći način:

```
Command (m for help): x
Expert command (m for help): h
Number of heads (1-256, default 30): 255
Expert command (m for help): s
Number of sectors (1-63, default 29): 63
Expert command (m for help): c
Number of cylinders (1-1048576, default 2286): <broj cilindara>
Expert command (m for help): r
```

Ovim su konfigurisani sektori, glave i cilindri. Dalje se kreiraju particije (neuneta vrednost daje podrazumevanu vrednost):

```
Command (m for help): n
Partition type:
 p primary (0 primary, 0 extended, 4 free)
 e extended
Select (default p): p
Partition number (1-4, default 1): 1
First sector (2048-15759359, default 2048):
Using default value 2048
Last sector, +sectors or +size{K,M,G} (2048-15759359, default ↩
   15759359): +200M
```

```
Command (m for help): n
Partition type:
 p primary (1 primary, 0 extended, 3 free)
 e extended
Select (default p): p
Partition number (1-4, default 2): 2
First sector (411648-15759359, default 411648):
Using default value 411648
Last sector, +sectors or +size{K,M,G} (411648-15759359, default↩
    15759359):
Using default value 15759359
```

Zatim se setuju boot flegovi i ID-jevi particija:

```
Command (m for help): a
Partition number (1-4): 1

Command (m for help): t
Partition number (1-4): 1
Hex code (type L to list codes): c
Changed system type of partition 1 to c (W95 FAT32 (LBA))

Command (m for help): t
Partition number (1-4): 2
Hex code (type L to list codes): 83
```

Proveriti tabelu particija i na kraju upisati izmene na kartici.

```
Command (m for help): p}

Disk /dev/sdb: 8068 MB, 8068792320 bytes
249 heads, 62 sectors/track, 1020 cylinders, total 15759360 ↩
   sectors
Units = sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disk identifier: 0x920c958b}

Device Boot Start End Blocks Id System
/dev/sdb1 * 2048 411647 204800 c W95 FAT32 (LBA)
/dev/sdb2 411648 15759359 7673856 83 Linux}

Command (m for help): w
The partition table has been altered!}

Calling ioctl() to re-read partition table.

WARNING: If you have created or modified any DOS 6.x
partitions, please see the fdisk manual page for additional
information.
Syncing disks.
```

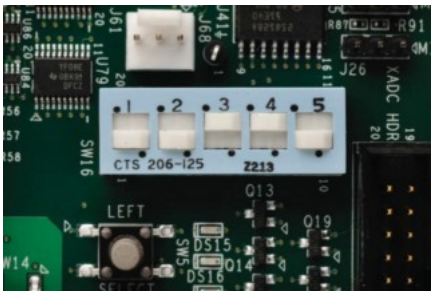
Kreiranje fajlsistema na particijama:

```
mkfs.vfat -F 32 -n boot /dev/sdX1
mkfs.ext4 -L root /dev/sdX2
```

Na sledeće komande mount-uju boot particiju u koju treba prekopirati gorenavedene fajlove.

```
mkdir -p /mnt/boot
mount /dev/sdX1 /mnt/boot}
```

Kada se završi kopiranje ubaciti karticu u SD slot na ploči podesiti prekidače za butovanje kao na slici B.34, uključiti napajanje i sačekati da se Linux pokrene.



Slika B.34: Podešavanje boot moda za boot sa SD kartice

Login podaci su:

username: `root`

password: `root`

Nakon što je Linux startovan, potrebno je kopirati drajver i aplikaciju na ploču. Ovo je najlakše postići *secure copy* (`scp`) naredbom kada su računar i ploča povezani preko LAN mreže. U nastavku je primer shell skripte koja kopira aplikaciju i drajver. Prva linija predstavlja brisanje `known_hosts` fajla što će sprečiti prijavu *RSA Host Key Change* grešeke svaki put kad se restartuje ploča, dok su druge dve linije kopiranje fajlova. Umesto 192.168.0.202 potrebno je staviti IP adresu ploče koja se može pročitati nakon `ifconfig` komande iz komandne linije Linuxa.

```
rm ~/.ssh/known_hosts
scp putanja_do_drajvera/mb_communication_driver.ko root@192↩
   .168.0.202:/
scp putanja_do_aplikacije/mb-communication-app root@192↩
   .168.0.202:/home/root
```

Da bi se pokrenula skripta potrebno joj je dodeliti dozvolu za izvršavanje

```
chmod +x ime_skripte.sh
```

Nakon kopiranja sve je spremno za učitavanje drajvera i startovanje aplikacije. Kao što je već objašnjeno ranije, drajver se učitava naredbom `insmod` i nakon toga se može pokrenuti aplikacija.

Dodatak B: Literatura

[1] *Vivado Design Suite User Guide Embedded Processor Hardware Design*, Xilinx, oktobar 2013.

[2] *Zynq-7000 All Programmable SoC: Concepts, Tools, and Techniques (CTT) A Hands-On Guide to Effective Embedded System Design (ug873)*, Xilinx, jun 2013.

[3] www.wiki.xilinx.com/Build+U-Boot

[4] www.wiki.xilinx.com/Build+kernel

[5] www.wiki.xilinx.com/Build+Device+Tree+Blob

[6] www.wiki.xilinx.com/Prepare+Boot+Image