

DIPLOMSKI RAD

**Komunikacija između Linux operativnog sistema i
hardvera realizovanog u FPGA**

Dejan Lukić 82 / 2014

Predmetni profesor:
Lazar Saranovac

Elektrotehnički fakultet
Univerzitet u Beogradu

Sadržaj

1 Uvod	3
1.1 Sistemi na čipu sa FPGA	3
1.2 Opis DE1-SoC	3
1.3 Opis Altera Cyclone 5	4
1.3.1 Konfigurisanje FPGA i pokretanje HPS	4
1.3.2 HPS-FPGA interfejsi	4
1.3.3 Proces pokretanja HPS (boot)	5
1.4 Alati	5
2 Opis projektovanog sistema	7
2.1 Memorijski mapiran interfejs ka LE diodama i tasterima u FPGA	7
2.2 Preloader	8
2.3 Bootloader	9
2.3.1 Skripta za U-Boot	9
2.4 Device Tree	10
2.4.1 Struktura Device Tree	10
2.5 Linuks kernel	13
2.6 Root fajl sistem	14
2.7 Drajver	14
2.7.1 Linux Device Model	14
2.7.2 Implementacija drajvera	15
3 Zaključak	18
4 Dodatak	20
4.1 Uputstvo za repliciranje rezultata	20
4.1.1 Projektovanje u Quartus-u	20
4.1.2 Podešavanja okruženja i preuzimanje kompjajlera	26
4.1.3 Generisanje RBF fajla	26
4.1.4 Generisanje i kompjajliranje Preloader-a	26
4.1.5 Generisanje Device Tree	27
4.1.6 U-Boot	27
4.1.7 Linuks kernel	29
4.1.8 Kompjajliranje drajvera	30
4.1.9 Generisanje Root File System	30
4.1.10 Priprema SD kartice	31
4.1.11 Testiranje sistema	32
4.2 Izvorni kodovi	33
4.2.1 Drajver	33
4.2.2 Skripta za U-Boot	38
4.2.3 Device Tree	39

1 Uvod

U ovom radu je na DE1-SoC razvojnom sistemu implementiran jednostavan hardver u FPGA, portovan je Linuks operativni sistem i napisan je drajver za pristup registrima i prihvatanje prekida iz FPGA. [1]

1.1 Sistemi na čipu sa FPGA

Sa sve većim mogućnostima namenskih sistema došlo je do popularizacije sistema na čipovima (SoC - *System on Chip*) koji integrišu mikroprocesore sa više jezgara, memorije na čipu, mnogobrojne periferije i transivere, kao i FPGA (*Field Programmable Gate Array*).

Ova tehnologija daje dizajneru sistema veliku slobodu i mogućnosti, a zadršava se klasičan postupak projektovanja namenskih sistema. Uz to se ostvaruje veća integracija, manja potrošnja, manja površina štampane ploče (PCB - *Printed Circuit Board*) i veći protok podataka između procesora i FPGA dela.

Uobičajena primena ovih sistema je implementacija specifičnih akceleratora koji ubrzavaju izvršavanje algoritama i implementacija specifičnih programabilnih interfejsa ka spoljnom svetu. Sve zrelije tehnologije kao što su OpenCL, Vivado HLS, Matlab HDL Coder omogućavaju kompatibilnost dizajna softvera na visokom nivou i implementiranog hardvera na niskom nivou.

SoC FPGA sistemi najčešće sadrže ARM mikroprocesor. Aplikacije na mikroprocesoru bez operativnog sistema (*baremetal application*) nude jednostavno pisanje koda i uštedu na resursima. Za kompleksnije aplikacije koriste se operativni sistemi (OS) i time se olakšava integrisanje mrežnih protokola, rad sa multimedijalnim sadržajima, kriptografskim bibliotekama kao i mnoge druge mogućnosti koje su dostupne kao *open-source* softver. Kada je potrebno garantovati reakciju u određenom vremenu na neki spoljni događaj veliki operativni sistemi nisu dobro rešenje i koriste operativni sistemi u realnom vremenu (RTOS - *Real time operating system*).

Hardver u FPGA se projektuje upotrebom nekog od dva popularna jezika za opis hardvera - Verilog i VHDL (*Very High Speed Integrated Circuit Hardwer Description Language*). Pored toga neophodni su softverski alati za specifični uređaj, koje obezbeđuje sam proizvođač uređaja. Dodatno ovi alati olakšavaju dizajn upotrebom IP(*Intellectual Property*) blokova, generisanjem raznih izlaznih fajlova koji opisuju projektovani hardver na standardni način i koriste se prilikom razvoja softvera.

1.2 Opis DE1-SoC

U ovom radu korišćen je DE1-SoC razvojni sistem koji se vrlo često upotrebljava u edukativne svrhe. Razvojni sistem je zasnovan na čipu iz familije Cyclone V kompanije Intel (ranije Altera).

U nastavku su navedene samo osobine razvojnog sistema koje se tiču ovog rada, a detaljniji opis se može pronaći u dokumentu zvaničnoj dokumentaciji proizvođača [] dodati referencu:

- Sistem na čipu Cyclone V 5CSEMA5F31
- Memorija 1GB (2x256Mx16) DDR3 SDRAM povezana na HPS
- Slot za Micro SD karticu povezan na HPS
- UART na USB (USB Mini-B konektor)
- 5 debaunsiranih tastera (FPGA x4, HPS x1)
- 11 LE dioda (FPGA x10, HPS x 1)

- 12V DC napajanje

1.3 Opis Altera Cyclone 5

Altera Cyclone V je SoC FPGA koji se sastoji od dva dela(slika): procesorski deo (HPS - *Hard processor System*) i programabilni FGPA deo. HPS se sastoji od MPU (*Microprocessor unit*) sa ARM Cortex-A9 MPCore sa dva jezgra i sledećih modula: kontroleri memorije, memorije, periferije, sistem interkonekcije, debug moduli, PLL moduli. FPGA deo se sastoji od sledećih delova: FPGA programabilna logika (*look-up* tabele, RAM memorije, množaci i rutiranje), kontrolni blok, PLL, kontroler memorije.

Svaki pin kućista je povezan na samo jedan od ova dva dela sistema, tako da HPS deo i FPGA deo ne mogu međusobno razmenjivati pinove.

1.3.1 Konfigurisanje FPGA i pokretanje HPS

Pri pokretanju HPS (boot) može da učita program iz FPGA dela, iz eksterne *flash* memorije ili preko JTAG. FPGA ima mogućnost da se konfiguriše softverski iz HPS korišćenjem periferije FPGA Manager ili spoljnim programatorom. Kombinacije ovih mogućnosti daju nekoliko scenarija:

- nezavisno konfigurisanje FPGA i pokretanje HPS
- konfigurisanje FPGA, zatim pokretanje HPS iz memorije koja se nalazi u FPGA
- pokretanje HPS, zatim konfigurisanje FPGA iz HPS

DE1-SoC razvojni sistem dolazi sa integriranim programatorom kojem se pristupa preko USB porta. Moguće je podešiti konfigurisanje FPGA spolja ili iz HPS upotrebom prekidača MSEL, dok se HPS uvek pokreće iz *flash* memorije SD kartice. (dodati tableau 3-2 iz de1soc user guide)

1.3.2 HPS-FPGA interfejsi

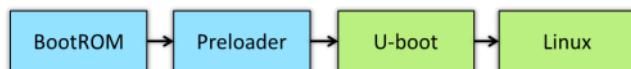
HPS-FPGA interfejsi su komunikacioni kanali između HPS i FPGA dela. U nastavku su nabrojani i opisani HPS-FPGA interfejsi:

- FPGA-to-HPS bridge - magistrala visokih performansi konfigurabilne sirine od 32,64 ili 128 bita. Na ovoj magistrali je FPGA master. Ovaj interfejs otkriva FPGA masterima ceo adresni prostor HPS dela.
- HPS-to-FPGA bridge - magistrala visokih performansi konfigurabilne sirine od 32,64 ili 128 bita. Na ovoj magistrali je HPS master a u FPGA se nalazi slave.
- Lightweight HPS-to-FPGA - magistrala sirine 32 bita. HPS je master na ovoj magistrali. Ovaj interfejs manjeg protoka je namenjen za pristup statusnim i kontrolnim registrima periferijama implementiranim u FPGA delu.
- FPGA manager - HPS periferija koja komunicira sa FPGA delom prilikom konfiguracije ili pokretanja (boot)
- Prekidi - mogućnost povezivanja prekida iz FPGA na HPS kontroler prekida
- HPS debug interfejs - omogućava da se debug mogućnosti prošire i na FPGA deo

Interfejsi koji su produžetak AXI magistrale na FPGA deo su FPGA-to-HPS bridge, HPS-to-FPGA bridge i Lightweight HPS-to-FPGA. Za povezivanje na ovu magistralu sa strane FPGA koristi se Avalon magistrala, stoga je neophodan AXI-Avalon bridge.

1.3.3 Proces pokretanja HPS (boot)

Pokretanje HPS je proces koji se obavlja u više koraka. Nakon izvršavanja svakog koraka se učitava i pokreće sledeći. Ovo je proces je sličan kod svih ARM procesora, a u nastavku je ukratko opisan za konkretnu platformu.



Slika 1: Tok pokretana sistema

Pri izlazu iz reset stanja procesor počinje izvrsavanje sa reset vektora iz memorije na čipu. Na adresi reset vektora je upisan Boot ROM program. Ovo je prvi korak u pokretanju HPS. *Boot ROM* izvršava osnovna podešavanja procesora i dohvata *preloader* iz NOR *flash* memorije, NAND *flash* memorije ili SD/MMC *flash* memorije. Očitavaju se BSEL pinovi na osnovu kojih se određuje gde je smešten *preloader*, zatim se inicijalizuje taj interfejs i učitava i pokreće *preloader*. *Boot ROM* softver proizvođača i ne može se menjati.

Preloader je prvi korak u pokretanju koji može da se konfiguriše. *Preloader* obično izvršava inicijalizaciju SDRAM, dodatna podešavanja sistema, inicijalizaciju *flash* kontrolera koji sadrži sledeći program (NAND, SD/MMC, QSPI) i zatim učitavanje programa u RAM memoriju i pokretanje.

Softver koji sledi nakon *preloader*-a može biti *baremetal* aplikacija ili *bootloader*. *Preloader* i svi prethodni programi se izvršavaju na prvom jezgru procesora dok je drugo u reset stanju. Naredni koraci mogu inicijalizovati drugo jezgro.

Bootloader ima zadatak da podesi promenljive okruženja operativnog sistema, dohvati fajlove za pokretanje operativnog sistema (sa *flash* memorije, putem *Etherneta* preko TFTP protokola ili USB), konfigurise FPGA pruži konzolu za korisničke operacije. Neki od populatnih *open-source bootloader*-a su U-Boot i Barebox.

1.4 Alati

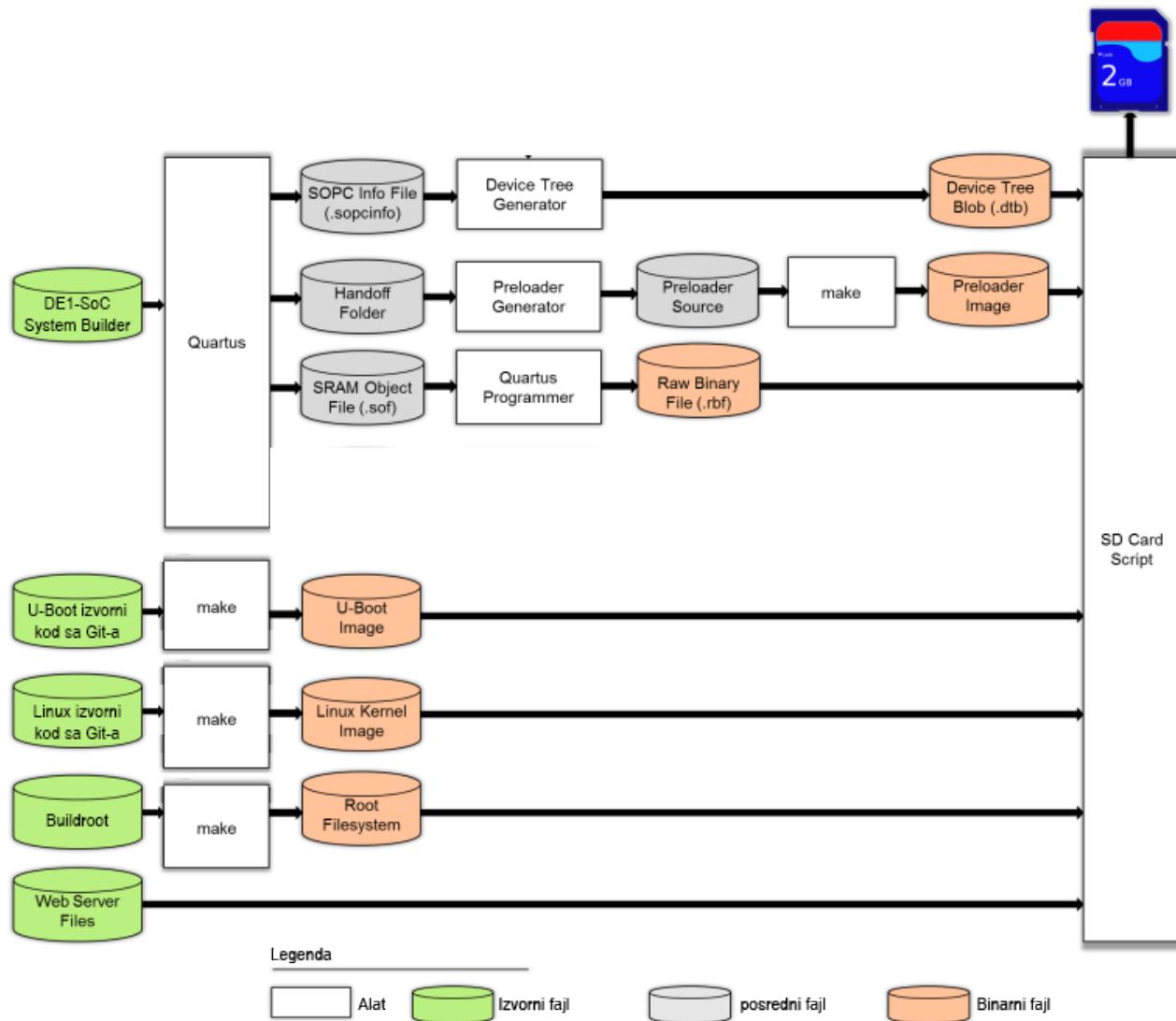
U nastavku će ukratko biti opisani korišćeni alati sa izdvojenim najvažnijim mogućnostima:

- Quartus Prime 18.0 - alat za razvoj hardvera na FPGA. Deo paketa je Platform Designer (ranije Qsys) koji u dizajn uključuje HPS, IP blokove i definiše povezanost ovih delova
- *Preloader Generator* (`bsp-editor` alat iz SoC EDS) - Generiše izvorni kod *preloader*-a na osnovi izlaznih fajlova koji opisuju hardver
- *Device Tree Generator* (`sopc2dts` alat iz SoC EDS) - Generiše *Device Tree* opis hardvera na osnovi izlaznih fajlova koji opisuju hardver
- *DE1-SoC Builder* - Generise prazan *Quartus* projekat za DE1-SoC razvojni sistem
- *Linaro Toolchain* - koristi se za kompajliranje softvera

Na slici 2 je grafički prikazan tok projektovanja jednog SoCFPGA sistema.

U nastavku su objašnjeni fajlovi koji se koriste pri projektovanju:

- `.qpf` - projektni fajl za Quartus. Ovaj fajl generiše DE1-SoC Builder
- `.qsf` - skripta za podešavanje pinova. Ovaj fajl generiše DE1-SoC Builder
- `.sdc` - skripta za podešavanje takta. Ovaj fajl generiše DE1-SoC Builder

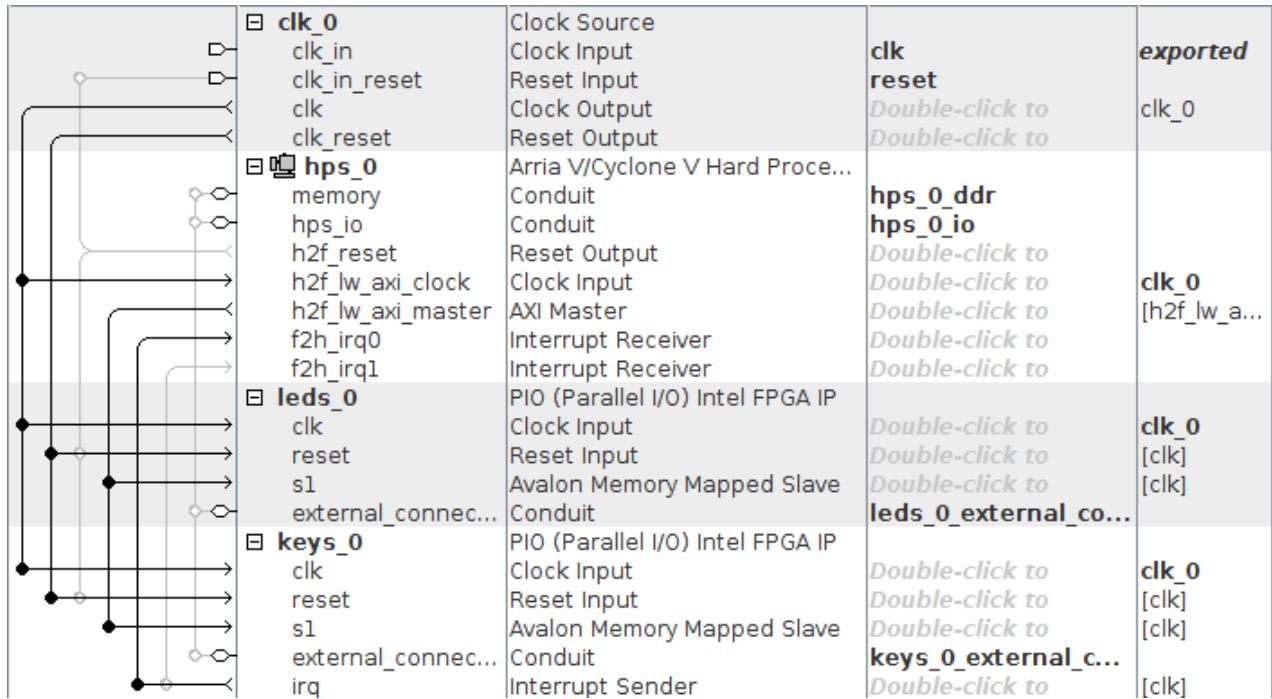


Slika 2: Tok projektovanja

- **.v** - Verilog HDL izvorni kod
- **.vhd** - VHDL izvorni kod
- **.sof** - SDRAM Object File - fajl za programiranje FPGA. Ovaj fajl generiše Quartus pri kompajlirajući dizajna
- **.rbf** - *Raw Binary File* - fajl za programiranje FPGA. Ovaj fajl se dobija konverzijom **.sof** alatom **quartus_cpf**
- **.dts** - *Device Tree Source* - opis hardvera za Linuks kernel
- **.dtb** - *Device Tree Blob* - binarni fajl, kompajlirani opis hardvera za Linuks kernel
- **.sopcinfo** - sadrži opis hardvera na osnovu kog se generišu drugi fajlovi. Ovaj fajl generiše *Platform Designer*
- **.c** - izvorni kod u jeziku C
- **Makefile** - sadrži set direktiva za **make build system**

2 Opis projektovanog sistema

U ovom radu je implementiran jednostavan sistem koji demonstrira osnovne mogućnosti u dizajniranju sistema na SoC FPGA. Na slici 12 je prikazan realizovani sistem (dodati novu sliku?)



Slika 3: Blok šema sistema

2.1 Memorijski mapiran interfejs ka LE diodama i tasterima u FPGA

Za projektovanje hardvera u FPGA koristi se PIO (*Parallel Input/Output*) Intel FPGA IP blok. Ovo je jedan od mnogih dostupnih Intelovih IP blokova sa standardizovanim *Avalon Memory Mapped Slave* interfejsom. HPS sistem pristupa registrima periferija implementiranim u FPGA (*soft cores*) preko standardne AXI magistrale. Između Avalon i AXI magistrale nalazi se automatski generisani *bridge*.

PIO Intel FPGA IP blok ima mogućnost da se konfiguriše kao ulazni, izlazni ili bidirekcionni. Takođe postoji mogućnost generisanja prekida na ulaznu, silazni ili obe ivice ulaznog signala kao i mogućnost generisanja prekida na osnovu nivoa ulaznog signala. Dostupna su podešavanja za širinu paralelnog porta.

U tabeli ref su opisani registri PIO Intel FPGA IP bloka.

PIO Intel FPGA IP je u ovom radu iskorišćen za jednostavnu demonstraciju tako što je povezan na pinove Cyclone V sistema na čipu koji su na DE1-SoC razvojnom sistemu povezani na LE diode i tastere. U slučaju konkretne primene na rešavanje nekog problema, PIO Intel FPGA IP pruža jednostavan memorijski mapiran interfejs i mogućnost slanja prekida iz FPGA dela. Ova mogućnost korisna je u slučaju kada je za rešavanje datog problema pogodno implementirati prizvoljno rešenje u programabilnoj logici. Ovaj pristup pruža slobodu projektovanja sistema u programabilnoj logici, a sa druge strane se zadržava standardizovani interfejs prema kompleksnom sistemu procesora. Na primer, izlazi PIO Intel FPGA IP registara mogu se koristiti kao konfiguracioni registri koji će upravljati korisničkom logikom, kao način da se programabilnoj logici dostave podaci koji će zatim biti obrađeni i vraćeni nazad. Mogućnost slanja prekida koristna je kada se PIO Intel FPGA IP koristi za posmartenje toka izvršavanja

algoritma u programabilnoj logici - na promenu nekog signala se generiše prekid i opcionalno zatim očitava iz statusnog regisra potrebna informacija koja oslikava stanje sistema. Takođe, prekid se može koristiti za sinhronizaciju toka podataka pri obradi podataka u FPGA tj. kao vid da se mikroprocesoru javi da su podaci spremni za čitanje. Rezultati rešavanja problema u FPGA se mogu učiniti dostupnim spoljnom svetu preko nekog od mnogih standardnih interfejsa (USB, Ethernet, UART, ...) i to vrlo jednostavnim postupcima u okviru operativnog sistema na HPS delu.

U ovom projektu u FPGA delu su postavljena dva PIO (*Parallel Input/Output*) Intel FPGA IP bloka. PIO IP blok `leds_0` je izlazni i koristi se za kontrolisanje LE dioda. PIO IP blok `keys_0` je ulazni i koristi se za očitavanje tastera. PIO IP blok `keys_0` takođe šalje prekidni zahtev HPS-u na svaku ulaznu i silaznu ivicu.

2.2 Preloader

Pri pokretanju HPS sistema koristi se *preloader* generisan Alterinim alatima. Za generisanje *preloader-a* neophodni su fajlovi za opis sistema koje generiše Platform Designer. Program za generisanje *Preloader Generator* je deo Alterinog SOC EDS(*Embedded Development Suite*) paketa alata. U grafičkom meniju se odabira folder u kojem se nalazi Platform Designer projekat sistema za koji se generiše *preloader*. Ovako generisani *preloader* je zasnovan na SPL (*Secondary Program Loader*) framework-u koji je deo U-Boot projekta. Ovo ima pozitivnu posledicu da *preloader* i U-Boot dele dosta izvornog koda, kao što je mnoštvo pouzdanih drajvera.

Standardna uloga *preloader-a* za Cyclone V sistem na čipu je da:

- inicijalizacije SDRAM interfejsa uključujući kalibraciju SDRAM PLL modula
- dohvata *bootloader* binarnog fajla sa *flash* memorije (NAND, SD/MMC, NOR)
- smešta binarni fajl *bootloader-a* u SDRAM i prepušta tok izvršavanja
- konfiguriše multipleksiranje pinova(podešavanja za konfiguraciju su dostupna u Platform Designeru)
- konfiguriše PLL na osnovu korisničkih podešavanja dostupnih u *preloader generator-u*
- otpušta određene periferije iz stanja reseta (izbor periferija se konfiguriše u Platform Designer-u)
- inicijalizuje *flash* kontroler (bilo NAND, SD/MMC ili QSPI) na osnovu *boot* prekidača

U ovom radu se *preloader* koristi za učitavanje i pokretanje U-boot *bootloader-a*. Izvršni fajl *bootloader-a* se nalazi na SD kartici. DE1-SoC razvojni sistem je već podešen za pokretanje sistema sa SD kartice, pa je preostalo u *preloader generator-u* uključiti podršku za FAT fajl sistem i definisati ime binarnog fajla *bootloader-a*. Nakon konfigurisanja *preloader-a* potrebno je kompajlirati izvršni fajl koji će se prebaciti na razvojni sistem. Kompajliranje se vrši jednostavnim pozivom *make* naredbe koja pokreće skriptu za *make build system* koja kao rezultat daje izvršni fajl.

Ovaj izvršni fajl je potrebno prebaciti na SD karticu na posebnu particiju. Prilikom particionisanja SD kartice neophodno je predvideti ovu particiju i podesiti njen tip na posebnu vrednost `a2`.

2.3 Bootloader

Open-source projekat U-Boot (*Universal Boot Loader*) je uobičajeni izbor *bootloader-a* za namenske sisteme zasnovane na ARM, PowerPC, MIPS procesorima. U-Boot se u ovom radu koristi za programiranje FPGA i učitavanje operativnog sistema. Ovo se postiže zahvaljujući tome što U-Boot nudi korisniku komandnu liniju koja se jednostavno može koristiti za pisanje skripti za željeno ponašanje *bootloader-a*.

U-Boot se preuzima u obliku izvornog koda sa git-a[ref na git] <https://github.com/u-boot/u-boot> Čas U-BootSource Tree Kako U-Boot podržava različite arhitekture potrebno je izvršiti konfiguraciju softvera. Konfiguracija je dostupna kao gotova za mnoge razvojne sisteme. Izvorni kod se konfiguriše kroz Kbuild infrastrukturu koja je se takođe koristi za konfiguraciju Linuksa. Konfiguracija se pokreće pozivom `make menuconfig` iz komandne linije i pruža vrlo jednostavan grafički interfejs i veliku moć podešavanja.

Konfiguracija za DE1-SoC razvojni sistem je dostupna kao gotova, što znači da se lako dobija funkcionalno konfigurisan izvorni kod. Osim toga, ovom konfiguracijom dostupne su komande programiranja FPGA dela preko *FPGA Manager* periferije.

Nakon osnovnog konfigurisanja softvera potrebno je napisati skriptu za željeno ponašanje i konfiguriranje U-Boot da pri pokretanju izvrši tu skriptu. Kroz Kbuild sistem se menja vrednost promenljive *bootcmd* koja sadrži komandu koja će biti izvršena odmah po pokretanju sistema ukoliko korisnik ne prekine ovaj proces. Promenljiva *bootcmd* je podešena tako da se izvrši kratka skripta za učitavanje naredne korisničke skripte (podešena je komanda *run callscript*). Dalje, u izvornom kodu je na mestu predviđenom za korisničke definicije definisana jednostavna skripta *callscript* koja samo učitava narednu korisničku skriptu (nazvanu *u-boot.scr*). Na ovaj način je ostavljena sloboda za izmenu skripte i time ponašanja *bootloader-a* bez potrebe da se ponovo konfiguriše i kompajlira U-Boot.

Ovim je konfiguracija U-Boot *bootloader-a* završena i zatim se vrši kompajliranje kako bi se dobio izvršni fajl. Za kompajliranje koristi se *Linaro toolchain*.

2.3.1 Skripta za U-Boot

Skripta za U-Boot piše se kao tekstualni i svodi se na pozive U-Boot komandi i definisanje vrednosti promenljivih. U-Boot komande pružaju velike mogućnosti i razlišite načine pokretanja operativnog sistema. Uobičajene primene ovih U-boot su:

- podešavanje promenljivih okruženja operativnog sistema
- dohvatanje binarnih fajlova za pokretanje operativnog sistema iz *flash* memorije ili preko *ethernet*-a pri čemu je omogućen pristup standardnim fajl sistemima kao i prenos preko mreže preko TFTP(*Trivial File Transfer Protocol*)
- smeštanje binarnih fajlova za pokretanje sistema u SDRAM i prepuštanje toka izvršavanja
- podešavanje *boot* argumenata koja se koriste za podešavanje kernela operativnog sistema

Nakon pisanja skripte potrebno je *mkimage* alatom dodati odgovarajući U-Boot heder. Fajl koji se dobija je potrebno prebaciti na SD karicu na FAT particiju. U-Boot je već konfigurisan tako da učita i izvrši ovu skriptu, kao što je opisano ranije.

U ovom radu skripta je napisana tako da programira FPGA i pokrene Linuks operativni sistem. Fajlovi koje U-Boot čita pri izvršavanju ove skripte su:

- **socfpga.rbf** - binarni za konfiguraciju FPGA
- **socfpga.dtb** - binarni fajl koji opisuje hardversku platofrmę i proslešuje se Linuks kernelu

- `zImage` - kompajlirani kernel

U skripti se podešava vrednost promenljive `bootargs` koja definiše argumente koji će biti prosleđeni kernelu pri pokretanju operativnog sistema. Ova promenljiva podešena je tako da definiše veličinu RAM memorije, parametre konzole za komunikaciju sa sistemom i određuje `ext3` particiju SD kartice kao *root file system* Linuksa.

Nakon učitavanja ovih fajlova izvršavaju se sledeće važne komande:

- `fpgaload` - za programiranje FPGA
- `bridge enable` - za inicijalizaciju AXI magistrale između FPGA dela i HPS dela
- `bootz` - komanda za pokretanje pokretanje operativnog sistema pri čemu se kernelu prosleđuje *Device Tree Blob* i argumenti iz promenljive `bootargs`

2.4 Device Tree

Device Tree ili skraćeno DT je struktura podataka koja opisuje hardver. Koristi se za identifikaciju platforme i podešavanja u toku izvršavanja kao što je vrednost argumenata pri pokretanju kernela (promenljiva `bootargs` koja je pomenuta ranije). Izdvajanje detaljnih informacija o sistemu u poseban fajl je uvedeno iz potrebe da se standardizuje način pokretanja kernela i interfejsa između *bootloader-a* i kernela i omogući jednostavno dodavanje novih plato-firmi. Tako na primer jednom kompajlirani kernel može da se koristi na različitim platformama. *Device Tree Source* koji opisuje detalje hardvera se kompajlira alatom *Device Tree Compiler* u binarni fajl *Device Tree Blob*. *Device Tree Blob* učitava *bootloader* i prosleđuje kernelu.

2.4.1 Struktura Device Tree

Device Tree struktura podataka je organizovana kao drvo tako da se sastoji od čvorova (*node*). Svaki čvor u ovoj strukturi mora imati čvor koji se nalazi iznad njega u hijerarhiji (*parent node*) osim korena stabla (*root node*). Svaki čvor predstavlja jedan drajver ili modul i sadrži opis osobina. Ukoliko postoje pravila za organizaciju informacija unutar čvora ona su opisana u *bindings* delu u Linuks dokumentaciji [<https://github.com/torvalds/linux/tree/master/Documentation/device-tree/bindings.rst>]. *Device Tree* ne mora da opiše svaki deo hardvera, a čvorovi koji su uobičajeni opisuju:

- procesori sistema (ovaj čvor je neophodan po specifikaciji)
- memorije sistema (ovaj čvor je neophodan po specifikaciji)
- specijalni konfiguracioni čvor za podešavanje argumemnata pri pokretanju kernela, podrazumevanih ulazno/izlaznih uređaja (ovaj čvor je neophodan po specifikaciji)
- čvor koji opisuje generalne osobine sistem na čipu (ovaj čvor je neophodan ukoliko je procesor u sistemu na čipu)
- čvorovi koji opisuju magistrale sistema (ovaj čvor nije neophodan ali se preporučuje)

Neke od vaznih osobina u čvoru su:

- `compatible` - koristi se da označi kompatibilnost uređaja sa porodicom sličnih uređaja i za uparivanje drajvera i uređaja
- `reg` - opisuje deo adresnog prostora u kom se nalazi uređaj kao par: početna adresa i dužina adresnog opsega

- **#address-cells** - određuje koliko 32-bitnih celija se koristi za opis početne adrese u **reg**
- **#size-cells** - određuje koliko 32-bitnih celija se koristi za opis dužine adresnog opsega u **reg** polju
- **interrupts** - koristi se za uređaje koji generišu prekid i opisuje najčešće broj prekida i tip (nivo/ivica)
- **interrupt-parent** - koristi se za uređaje koji generišu prekid i pokazuje na kontroler prekida kojem se šalje prekidni zahtev
- **interrupt-controller** - boolean vrednost koja označava da li je uređaj kontroler prekida
- **#interrupt-cells** - određuje koliko 32-bitnih celija se koristi za opis **interrupts** polja u uređajima koji koriste kontroler prekida

U nastavku(premestiti u dodatak?) je na primeru dat pojednostavljeni *Device Tree Source* koji opisuje DE1-SoC razvojni sistem. Na osnovu prethodnog teksta se lako može razumeti organizacija i sastav sistema. Dati primer je pojednostavljen tek toliko što su izostavljeni uređaji sistema na čipu čiji rad se ne izučava u ovom radu. Ceo *Device Tree* je podeljen u nekoliko fajlova i može se pronaći u dodatku ovog rada i literaturi [socfpga_cyclone5_socdk.dts socfpga_cyclone5.dtsi socfpga.dtsi] Iz priloženog koda se vidi da se sistem sastoji od:

- dva procesorska jezgra Cortex-A9
- standardni ulazno/izlazni uređaj je serijski uređaj **serial0**
- RAM memorija je kapaciteta 1GB
- koristi se kontroler prekida ARM *Generic Interrupt Controller*
- na memorijskoj adresi 0xff200000 koja odgovara početku adresnog prostora AXI magistrale koji izlazi prema FPGA se nalazi korisnički hardver koji se implementira u ovom radu i zauzima adresni opseg dužine 32 memorijске lokacije, prekidni zahtev je pod brojem 40 i ostljiv je na uzlaznu i silaznu ivicu
- **serial0** je periferija **uart0** na naznačenoj adresi i sa naznačenim brojem prekida

```
/ {
    #address-cells = <1>;
    #size-cells = <1>;

    cpus {
        #address-cells = <1>;
        #size-cells = <0>;

        cpu0: cpu@0 {
            compatible = "arm,cortex-a9";
            device_type = "cpu";
            reg = <0>;
        };
        cpu1: cpu@1 {
            compatible = "arm,cortex-a9";
            device_type = "cpu";
            reg = <1>;
        };
    };
}
```

```

};

chosen {
    bootargs = "earlyprintk";
    stdout-path = "serial0:115200n8";
};

memory@0 {
    name = "memory";
    device_type = "memory";
    reg = <0x0 0x40000000>; /* 1GB */
};

intc: intc@ffffed000 {
    compatible = "arm,cortex-a9-gic";
    #interrupt-cells = <3>;
    interrupt-controller;
    reg = <0xffffed000 0x1000>,
          <0xffffec100 0x100>;
};
soc {
    #address-cells = <1>;
    #size-cells = <1>;
    compatible = "simple-bus";
    device_type = "soc";
    interrupt-parent = <&intc>;
    ranges;

    lddipl0: lddipl@0xff200000 {
        compatible = "ld,dipl";
        reg = <0xff200000 0x00000020>;
        interrupts = <0 40 1>;
    };

    uart0: serial0@ffc02000 {
        compatible = "snps,dw-apb-uart";
        reg = <0ffc02000 0x1000>;
        interrupts = <0 162 4>;
    };
}

```

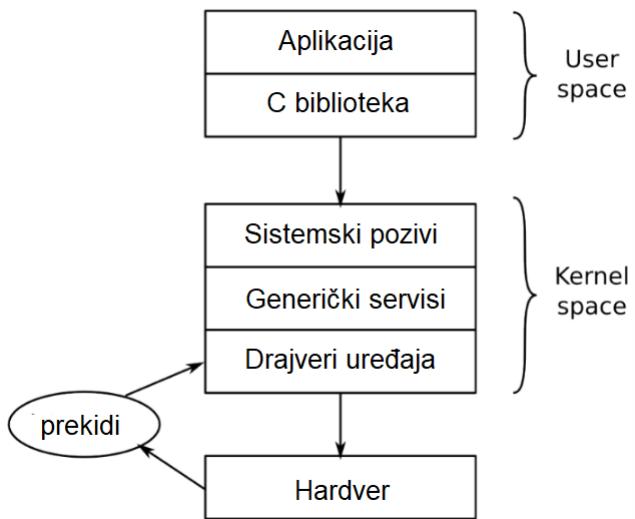
Kada je *Device Tree Source* napisan potrebno je smestiti .dts fajl u folder `arch/arm/boot/dts/`. Kompajliranje izovrnog opisa u binarni fajl *Device Tree Blob* se ivršava komandom `make *.dtb` gde je potrebno upisati naziv fajla koji se kompajlira.

Potrebno je napomenuti da je u ovom radu modifikovan *Device Tree* za razvojni sistem *Cyclone V Developoment Kit* koji je veoma sličan DE1-SoC. Uvedene modifikacije se svode na isključivanje CAN uređaja koji Cyclone V *Developoment Kit* ima a DE1-SoC nema. Zatim je dodat opis implementiranih PIO Intel FPGA IP u FPGA. Drugi način dobijanja *Device Tree* koji nije zahtevao ručnu modifikaciju je korišćenje *Device Tree Generator*-a koji je dostupan u okviru *Altera Embedded Developoment Suite* paketa softverskih alata. Za ovaj način je testiranjem utvrđeno da postoji sledeći problem: drajver ne vidi uređaje koji pripadaju čvoru `bridge` koji opisuje adresni prostor između FPGA i HPS i samim tim i generisani hardver u FPGA. Najverovatniji razlog ovom problemu je različita verzija kernela od one za koju je alat namenjen, ali je u skladu sa edukativnim ciljem ovog rada izabran put ručnog pisanja *Device Tree* umesto korišćenja generisanog.

2.5 Linuks kernel

Linuks kernel centralni deo operativnog sistema koji ima zadatak da upravlja resursima sistema i pruži interfejs prema hardveru. Linuks kernel je *open-source* projekat koji održava i razvija veliki broj programera i kompanija. Izvorni kod je otvoren što omogućava prilagođavanje za različite platforme kao i privlači razvoj raznih kompatibilnih projekata. Ova osobina čini Linuks veoma popularnim u namenskim sistemima. Prednosti su portabilnost i hardverska porška koju pruža zajednica i proizvođači. Kernel je skalabilan pa se može konfigurisati za izvršavanje na super-računarima kao i na malim uređajima sa čak 4MB RAM memorije. Takođe, Linuks nudi gotove implementacije raznih delova sistema koji su potrebni za povezivanje i bezbednost.

Glavne uloge kernela su da upravlja harverskim resursima procesorom, memorijom i ulazno/izlaznim uređajima. Iz ugla razvoja softvera kernel nudi standardizovani interfejs (API Application Programming Interface) koji je portabilan i nezavistan od arhitekture i specifičnog hardvera. Ovaj interfejs nudi korisne apstrakcije koje su prikazane na slici ?? Korisničke apli-



Slika 4: Tok pokretana sistema

kacije se izvršavaju u neprivilegovanom režimu i uglavnom pozivaju standardnu C biblioteku. C biblioteka se obraća kernelu preko sistemskih poziva, a implementacija sistemskih poziva je različita u zavisnosti od arhitekture sistema. Drajveri uređaja pružaju interfejs prema hardveru i biće detaljnije objašnjeni kasnije.

Jedna od osnovnih odlika Linuks kernela je podela sistema na privilegovani *kernel-space* deo i na neprivilegovani *user-space* deo. U privilegovanim režimima kernel ima pristup zaštićenom delu memorije i slobodan pristup hardveru. *User-space* je namenjen za aplikacije kojima je dozvoljen ograničen pristup određenim resursima. Međutim, jedna od korisnih mogućnosti za namenske sisteme koja se nudi u *user-space*-u je pristup memorijskom prostoru u kojem se nalaze periferije. Ovo se postiže mapiranjem memorijskog prostora u virtualnu adresu. Ovaj način pristupa registrima je zgodan jer je brz i jednostavan, a u nekim slučajevima je dovoljan za rešavanje datog problema. Veliki nedostatak ovog pristupa je što ne postoji način da se registruju prekidne rutine.

U ovom radu korišćena je verzija 4.6 Linuks kernela dostupna na git repozitorijumu kompanije Altera [<https://github.com/altera-opensource/linux-socfpga>]. Verzije kernela koje obezbeđuju proizvođači su modifikovane za podržane platforme tako što su dodate konfiguracije i drajveri. Veći proizvođači vremenom unete izmene unesu i u glavnu verziju kernela. U ovom radu je korišćena Alterina verzija jer se preporučuje prema zvaničnim uputstvima.

Po preuzimanju izvornog koda vrši se konfiguracija upotrebom podrazumevanog konfiguracijskog fajla za Alterine sisteme na čipu. Konfiguracija se izvršava komandom `make socfpga_defconfig`. Nakon toga se pokreće kompajliranje kernela komandom `make zImage`. U folder `arch/arm/boot` se nalazi izvršni fajl `zImage` koji je spreman za prebacivanje na SD karticu.

2.6 Root fajl sistem

Root fajl sistema je neophodan deo operativnog sistema. *Root* označaca koren fajl sistema i psotavlja se na nekom fajl sistemu koji se nalazi na trajnoj memoriji. Struktura *root* fajl sistema je definisana standardnom [https://wiki.linuxfoundation.org/lsb/fhs] koji definiše strukturu i nazive fodlera kao i njihovu namenu. Neophodne je da sa na definisanim mestima u *root* fajl sistemu nalaze sva potrebna podešavanja za ispravno pokretanje kernela, kao i korisničke aplikacije koje nude od osnovne funkcionalnosti sistema do kompleksnih i velikih aplikacija.

U nekim delovima *root* fajl sistema se nalaze virtuelni fajl sistemi koji nude organizovanu reprezentaciju kernela. Neki od ovih virtuelnih fajl sistema nude korisne mogućnosti kao što je modifikacija parametara kernela. Za ovaj rad je važno pomenuti folder `dev` u kojem se nalaze svi uređaji na sistemu predstavljeni kao fajlovi i folder `sys` koji će detaljnije biti opisan kasije.

Root fajl sistema može da se nabavi kao gotov ili da se pravi ručno. Prednosti gotovog *root* fajl sistema je lako korišćenje ali dolazi sa manama jer se teško prilagođavaju konkretnoj primeni i optimizuju, a često su veoma veliki. Ručno pravljenje omogućava potpunu kontrolu i dobru priliku da se nauči dosta o samom radu operativnog sistema, ali to može postati kompleksan zadatak ukoliko je potrebno brzo i pouzdano pokrenuti sistem.

Open-source zajednica nudi različita rešenja za automatizaciju pravljenja *root* fajl sistema. Dva najveća projekta su Yocto/OpenEmbedded i Buildroot. Yocto/OpenEmbedded je veliki projekat koji unosi apstrakcije i generalizacije sa ciljem da generiše kompletну distribuciju Linuks operativnog sistema i lako podrži različite platforme. Za jednostavnije sisteme za koje je potrebno generisati samo jednostavan *root* fajl sistem se koristi Buildroot.

Buildroot takođe nudi generisanje izvršnih fajlova kernela i *bootloader-a*. Buildroot koristi poznati `make` sistem za konfiguraciju, samim tim je jednostavan za korišćenje i modifikovanje prema nameni. Ovo čini Bulidroot veoma popularnim u namenskim sistemima.

U ovom radu je demonstrirana primena Buildroot-a za generisanje *root* fajl sistema dok su Linuks kernel i *bootloader* ručno preuzeti i kompajlirani. Buildroot se konfiguriše unosom informacija o sistemu: podešeni su detalji arhitekture u skladu sa razvojnim sistemom i podešena je putanja ka eksternom Linaro *toolchain-u*. Nakon toga se naredbom `make` za kratko vreme dobija arhiviarni fajl koji je spreman za otpakivanje na SD karticu.

2.7 Drajver

Drajveri imaju specijalnu ulogu u Linuks kernelu. Drajver je deo sistema koji obezbeđuje da deo hardvera radi na očekivani način preko jednostavnog definisanog programerskog interfejsa, pri čemu se krije kompleksnost detalja rada hardvera. Korisničke aktivnosti se svode na jednostavne pozive nezavisno od vrste drajvera, a ti pozivi se preslikavaju na operacije specifične za svaki drajver koji deluju na pravom hardveru. Drajveri su organizovani kao moduli kernela tako da se razvijaju odvojeno i mogu se uključiti u kernel po potrebi. Moduli po učitavanju postaju deo kernela tako da imaju pristup `kernel space`-u a komuniciraju sa `user space`-om preko određenog interfejsa.

2.7.1 Linux Device Model

Linux Device Model je novi standard koji uvodi apstrakcije o sastavu sistema. Pre ovoga nije postojala jedna struktura koja opisuje način na koji je sistem sastavljen. Ove apstrakcije

izdvajaju sličnosti uređaja i grupišu ih nezavisno od specifične implementacije. Linux Device Model nudi podršku za različite zadatke kao što su: upravljanje potrošnjom, komunikacija sa `user space`-om, priključivanje uređaja tokom rada (*hotplug, plug and play*), podela uređaja u klase. Bitne komponente Linux Device Model-a su `udev`, `sysfs`, `kobject` i klase uređaja.

`sysfs` je virtuelni fajl sistem koji pruža interfejs ka strukturama kornela. Ovaj fajl sistem prikazuje strukturu Linux Device Model-a i se obično nalazi u folderu `sys`. Fajlovi u `sysfs` pružaju informacije o uređajima, modulima kornela, fajl sistemima i drugim komponentama kornela. Većina fajlova može samo da se čita kako bi se dobile informacije o kernelu, dok su neki namanjeni i za upis što dozvoljava izmene u kernelu. Kako bi se izbegla redundantnost dosta se koristi simboličko linkovanje tako da se pravi fajl postoji samo na jednom mestu, a ako je potrebno prikazati ga na drugim mestima stvara se simbolička veza ka originalnom fajlu.

Svaki fajl u `/sys` folderu je zasnovan na strukturi `kobject`. Ovaj objekat koja nudi neke opšte osobine koje se koriste kod svih struktura kao što je brojač referenci, pripadnost skupu i tip strukture. `kobject` nije namenjen da se koristi sam po sebi već je obično deo neke veće strukture i samim tim se i operacije sa `kobject`-om kriju iza programerskog interfejsa `framework-a`.

Klase uređaja grupišu uređaje po funkcionalnosti a ne prema načinu povezivanja i implementacije. Unutar foldera `sys/class` se nalaze klase koje grupišu uređaje u terminale, mrežne uređaje, grafičke uređaje, zvučne urešaje itd. U ovim folderima se nalaze simboličke veze ka unosima u folderu `sys/devices` u kojem se nalazi *Device Tree* opis kornela.

U folderu `dev` su postoji jedan fajl za svaki uređaj. `udev` ima zadatak da dinamički dodaje uređaje u ovaj folder. `udev` se oslanja na informacije iz `sysfs`.

Linux Device Model je organizovan oko tri glavne strukture podataka:

- `struct bus_type` - predstavlja jedan tip magistrale (USB, PCI, I2C)
- `struct device_driver` - predstavlja jedan drajver koji komunicira sa uređajima na određenoj magistrali
- `struct device` - predstavlja jedan uređaj prikačen na magistralu

U kernelu se koristi nasleđivanje ovih struktura kako bi se stvorile specijalizovane verzije drajvera za svaku magistralu i uređaj.

2.7.2 Implementacija drajvera

Drajver je implementiran kao modul kornela. Drajver pripada *platform* magistrali. Ovo je pseudo magistrala minimalne infrastrukture koja se koristi za memoriski mapirane uređaje kojima se može pristupiti direktno preko magistrale procesora. Ova magistrala je uobičajeni izbor za namenske sisteme, pogotovo kada se implementira drajver za neku posebnu namenu kao što je FPGA periferija u slučaju ovog rada. Implementirani *platform* drajver se koristi za dohvatanje informacija o uređaju iz *Device Tree* strukture, za registrovanje prekidne rutine i za stvaranje `sysfs` fajlova koji korisniku pružaju jednosavan interfejs prema hardveru.

Modul kornela

Važni delovi svakog modula kornela su:

- makro definicije za osnovne informacije o modulu kao što su ime autora, opis, licenca, verzija modula
- `init` funkcija koja se izvršava pri učitavanju modula u kernel
- `exit` funkcija koja se izvršava pri uklanjanju modula iz kornela

U funkcijama `init` i `exit` se pozivaju funkcije sa registrovanje *platform* drajvera.

Platform drajver

Platform drajver se definiše jednostavnom strukturu čija su polja:

- **probe** - pokazivač na funkciju koja će biti pozvana pri uspešnom registrovanju drajvera
- **remove** - pokazivač na funkciju koja će biti pozvana pri uklanjanju drajvera
- **driver** - sadrži informacije o imenu drajvera i pokazivač na **of_device_id** koja identificuje uređaj u *Device Tree* na osnovu **compatible** polja (neophodno je da se vrednosti definisane u drajveru i u *Device Tree* poklope kako bi se uspešno registrovao drajver)

Funkcije **probe** i **remove** se koriste za dohvatanje informacija iz *Device Tree* i potrebna podešavanja:

- dohvataju se adresa početka i dužina memorijskog opsega periferije pozivom funkcije **platform_get_resource** i zatim se taj opseg mapira pozivom funkcije **ioremap**
- dohvata se broj prekida pozivom funkcije **get_platform_irq** i zatim se registruje prekidna rutina pozivom funkcije **requset_irq**

Dalje se prave **sysfs** fajlovi.

sysfs fajlovi

sysfs se koristi kao jednostavan intefrejs prema **user space-u**. Stvaranje fajlova se u drajveru svodi na jednostavno pisaje funkcija **.show** za čitanje i **.store** za upis. Ovim izborom se postiže jednostavno pisanje drajvera sa prostim programerskim intefrejsom, dok su alternativni načini bili pisanje **char** drajvera, stvaranja klase uređaja ili upotreba nekog standardnog **framework-a** za ovu konkretnu primenu kao što je GPIO , LED.

Stvoreni fajlovi se pojavljuju u folderi koji je se nalazi u **/sys/bus/platform/driver/lddipldrv**. Sistemski fajlovi koje pravi drajver i njihov opis:

- **leds** - broj upisan u ovaj fajl se prikazuje na LE diodama u binarnoj predstavi
- **keys** - čitanje iz ovog fajla vraća binarnu predstavu stanja tastera
- **irq_flag** - pristup registru za flegove prekida (1 na n-tom bitu označava pristigli prekid na n-tom tasteru, upis 1 na n-ti bit čisti n-ti fleg)
- **irq_mask** - čitanje i upis u registar za maksiranje prekida (upis 1 na n-ti bit omogućava prekid na n-tom tasteru)

Ovi fajlovi se mapiraju u pozive funkcija koje ivršavaju jednostavne upise i čitanje u registre. Za detalje o radu drajvera pogledati ceo kod drajvera koji je dostupan u dodatku.

Testiranje sistema

Za testiranje sistema je neophodan DE1-SoC razvojni sistem sa kablom za napajanje i USB kabl i PC računar za komunikaciju preko terminala.

Pre testiranja potrebno je spakovati sve sistemske fajlove na određeni način. Za particionisanje SD kartice koristi se alat **fdisk**. SD karticu treba prvo formatirati i zatim particionisati tako da sadrži tri particije:

1. particija na kojoj se nalazi FAT fajl sistem

2. particija na kojoj se nalazi *ext3*

3. specijalna particija tipa a2

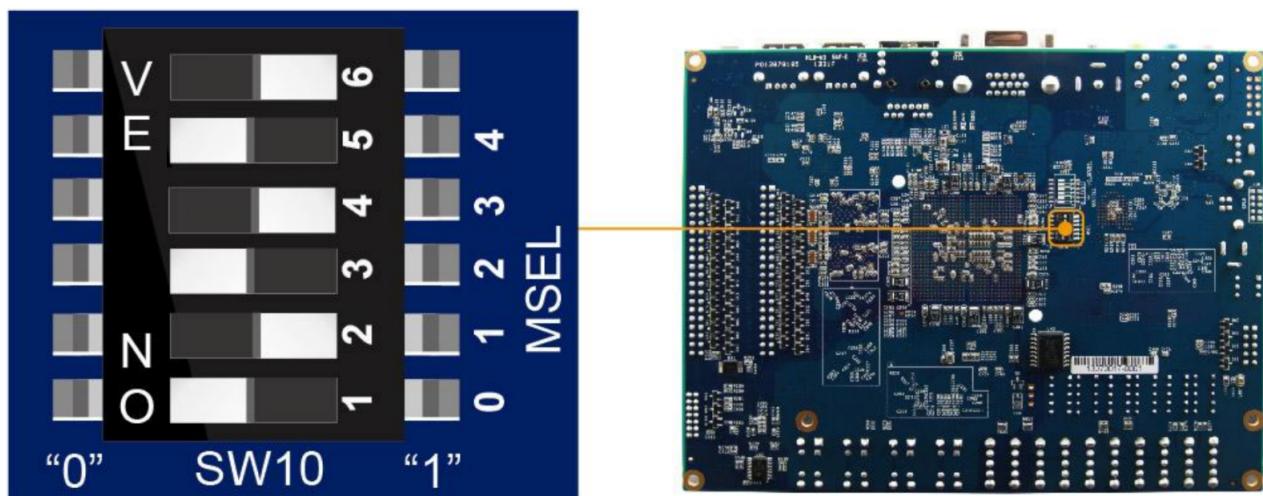
Na prvoj particiji se nalazi svi fajlovi neophodni za pokretanje sistema. Podrška za FAT fajl sistem je uključena u U-Boot što omogućava učitavanje fajlova i pokretanje sistema. Na ovoj particiji se nalazi skripta za U-Boot, binarni fajl za programiranje FPGA, binari opis hardvera *Device Tree Blob* i binarni fajl kernela.

Na drugoj particiji je potrebno otpakovati sadržaj arhive koja je generisana Buildroot-om. Ova particija je podešena kao *root* fajl sistem. Nakon toga na ovu particiju se kopira izvršni fajl drajvera u folder */home/root*.

Na trećoj particiji ne postoji fajl sistem. Na ovu particiju se direktno smešta izvršni fajl *preloader-a*. Razvojni sistem je konfigurisan da traži *preloader* na SD kartici a Boot ROM očekuje ovaj program na particiji tipa a2.

Nakon prebacivanja potrebnih fajlova na SD karticu, potrebno je ubaciti SD karticu u slot u razvojnog sistemu.

Na ploči razvojnog sistema je potrebno podešiti prekidače koji određuju da se FPGA programira softverski. Podešavaju prekidača **MSEL** na vrednost 01010 je prikazana na slici 7 ??



Slika 5: Podešavanje prekidača **MSEL** za softversko konfigurisanje FPGA, pozicija prekidača na ploči razvojnog sistema

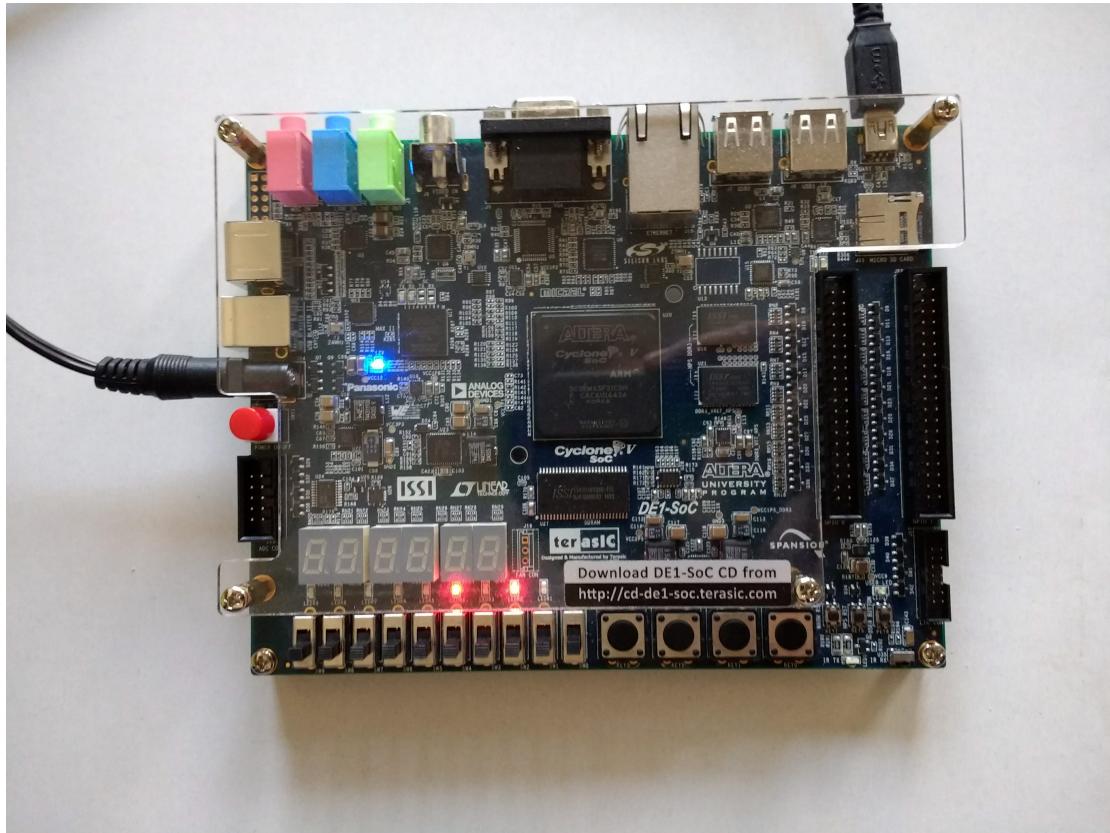
Sistem povezati na PC računar USB kablom i otvoriti terminal za komunikaciju.

Nakon uključivanja sistema u terminalu će početi ispisivanje sistemskih obaveštenja koja su uobičajena pri pokretanju sistema. Nakon toga biće zatraženo ime korisnika i lozinka. Po unosu ovih informacija dobija se pristup linuks sistemu. Za učitavanje drajvera koristi se komanda `insmod`. Nakon učitavanja u folderu `sys/bus/platform/driver/lddipldrv` se pojavljuju sistemski fajlovi za pristup hardveru. Korišćenjem komande za upis u fajl `leds` se može upisati vrednost koja će se pojaviti na LE diodama. U primeru na slici ?? je upisan broj 10 (binarno 1010) i to je ispisano na LE diodama ???. ?? Za demonstraciju pristizanja prekidne rutine je potrebno dozvoliti prekida na tasteru 1 upisom u sistemski fajl `irq_mask`. Nakon pritiskanja tastera na razvojnom sistemu u konzoli se pojavlju obaveštenje koje ispisuje prekidna rutina drajvera 8

Ovim postupcima je pokazan ispravan rad projektovanog sistema.

```
# ls
bind          irq_mask      module
ff200000.lddipl keys        uevent
irq_flag      leds         unbind
# echo 10 > leds
```

Slika 6: Korišćenje drajvera za uključivanje LE dioda



Slika 7: Razvojni sistem DE1-SoC povezan za testiranje

```
# pwd
/sys/bus/platform/drivers/lddipldrv
# ls
bind          irq_mask      module
ff200000.lddipl keys        uevent
irq_flag      leds         unbind
# echo 1 > irq_mask
# [ 2654.886937] irq received: 1
```

Slika 8: Dozvola prekida i pristizanje prekidne rutine

3 Zaključak

Izrada ovog rada je motivisana željom da se upoznaju konkretni alati i postupak projektovanja sa SoC FPGA sistemima. Izveštaj o radu je napisan tako da pruži kratak pregled bitnih poj-

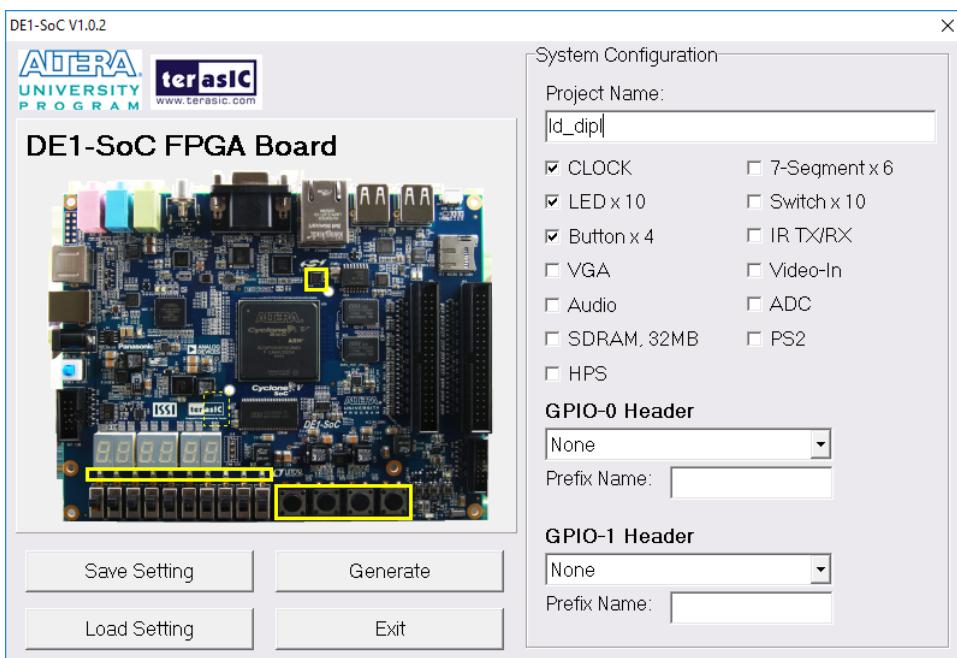
mova i detaljna uputsva za reproduciju rezultata sa osvrtom na usputne probleme. Stečeno znanje može olakšati slušanje predmeta na master studijama, ili biti osnova za rešavanje konkretnog problema (npr. ubrzavanje algoritama kompresije i obrade slike), a autoru je olakšalo snalaženje na novom radnom mestu. Autor izražava veliku zahvalnost profesoru Lazaru Saranovcu i asistentu Strahinji Jankovicu za podršku i savete prilikom izrade diplomsog rada.

4 Dodatak

4.1 Uputstvo za repliciranje rezultata

4.1.1 Projektovanje u Quartus-u

1. U Microsoft Windows operativnom sistemu pokrenuti DE1SoC_SystemBuilder.exe (dostupan na [])
2. Izabrati konfiguraciju kao na slici 9 i izabrati Generate



Slika 9: Podešavanja DE1_Soc Bulder-a

3. Izbristati `ld_dipl.v` fajl (ovaj fajl se ne koristi, kasnije će biti napravljen `ld_dipl.vhd` fajl za VHDL)
4. Kopirati generisane fajlove u Ubuntu u radni folder (u ovom radu je to `~/ld_dipl/hw/`)
5. Pokrenuti Quartus (Quartus Prime 18.0) Lite Edition
6. Otvoriti projekat komandom `File > Open Project...` i izabrati `~/ld_dipl/hw/ld_dipl.qpf`
7. U prozoru Tasks izabrati `Edit Settings`, u novom prozoru pod `Timing Analyser` ispod teksta `SDC files to include in the project` klikom na dugme '...' izabrati fajl `ld_dipl.sdc`
8. Pokrenuti Platform Designer klikom na `Tools > Platform Designer`
9. Iz prozora IP Catalog izabrati `Processors and Peripherals > Hard Processor Systems > ArriaV/Cyclone V Hard Processor System`. Ovim se otvara meni za podešavanje HPS modula
10. Pod tabom `FPGA interfaces` izvršiti sledeće izmene:
 - U opštim podešavanjima isključiti opciju `Enable MPU standby and event signals`

- U podešavanjima AXI Bridges podesiti FPGA-to-HPS interface i HPS-to-FPGA interface na unused, a Lightweight HPS-to-FPGA na 32-bit
- U podešavanjima FPGA-to-HPS SDRAM Interface izabratи f2h_sdram0 i zatim isključiti pritiskom na dugme '-'
- U podešavanjima Interrupts uključiti opciju Enable FPGA-to-HPS interrupts

11. Pod tabom **Peripheral Pins** izvršiti sledeće izmene

- U podešavanjima SD/MMC Controller postaviti SDIO pin na HPS I/O Set 0 i SDIO mode na 4-bit Data
- U podešavanjima UART Controllers postaviti UART0 pin na HPS I/O Set 0 i UART mode na No Flow Control

U ovom tabu je za potrebe nekog drugog projekta moguce uključiti ostale periferije: CAN Controller, Ethernet Media Access Controller, I2C Controller, SPI Controller, QSPI Flash Controller, NAND Flash Controller, Trace Port Intefrace Unit, GPIO za podešavanja pogledati []

12. Pod tabom **HPS Clocks** ostaviti podešavanja na podrazumevanim vrednostima

13. Pod tabom **SDRAM** podesiti:

- SDRAM Protocol: DDR3
- PHY Settings:
 - Clocks:
 - * Memory clock frequency: 400.0 MHz
 - * PLL reference clock frequency: 25.0 MHz
 - Advanced PHY Settings:
 - * Supply Voltage: 1.5V DDR3
- Memory Parameters:
 - Memory vendor: Other
 - Memory device speed grade: 800.0 MHz
 - Total interface width: 32
 - Number of chip select/depth expansion: 1
 - Number of clocks: 1
 - Row address width: 15
 - Column address width: 10
 - Bank-address width: 3
 - Uključiti DM pins
 - Uključiti DQS#
 - Memory Initialization Options:
 - * Mirror Addressing: 1 per chip select: 0
 - * Burst Length: Burst chop 4 or 8 (on the fly)
 - * Read Burst Type: Sequential
 - * DLL precharge power down: DLL off
 - * Memory CAS latency setting: 11

- * Output drive strength setting: RZQ/7
- * ODT Rtt nominal value: RZQ/4
- * Auto selfrefresh method: Manual
- * Selfrefresh temperature: Normal
- * Memory write CAS latency setting: 8
- * Dynamic ODT (Rtt_WR) value: RZQ/4

- Memory Timing:

- tIS (base): 180 ps
- tIH (base): 140 ps
- tDS (base): 30 ps
- tDH (base): 65 ps
- tDQSQ: 125 ps
- tQH: 0.38 cycles
- tDQSCK: 255 ps
- tDQSS: 0.25 cycles
- tQSH: 0.4 cycles
- tDSH: 0.2 cycles
- tDSS: 0.2 cycles
- tINIT: 500 us
- tMRD: 4 cycles
- tRAS: 35.0 ns
- tRCD: 13.75 ns
- tRP: 13.75 ns
- tREFI: 7.8 us
- tRFC: 260.0 ns
- tWR: 15.0 ns
- tWTR: 4 cycles
- tFAW: 30.0 ns
- tRRD: 7.5 ns
- tRTP: 7.5 ns

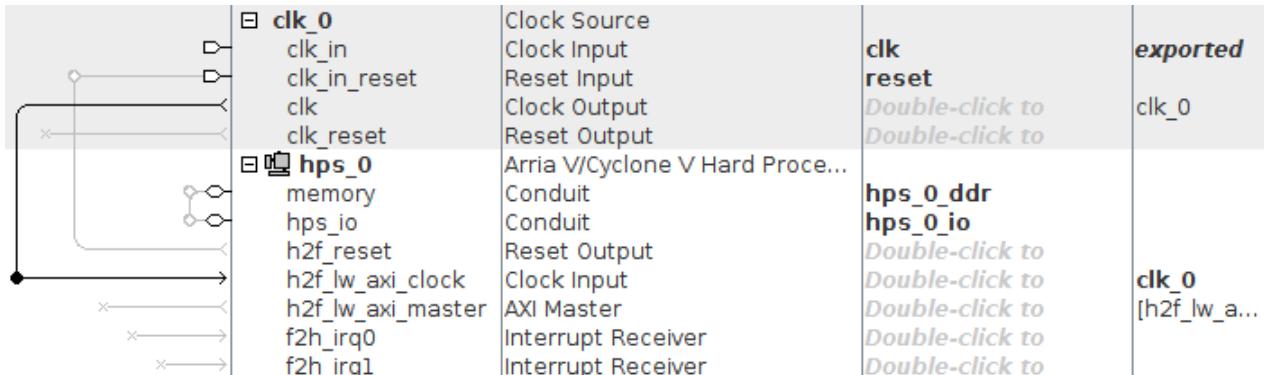
- Board Settings:

- Setup and Hold Derating:
 - * Use Altera's default settings
- Channel Signal Integrity:
 - * Use Altera's default settings
- Board Skews:
 - * Maximum CK delay to DIMM/device: 0.03 ns
 - * Maximum DQS delay to DIMM/device: 0.02 ns
 - * Minimum delay difference between CK and DQS: 0.06 ns
 - * Maximum delay difference between CK and DQS: 0.12 ns
 - * Maximum skew within DQS group: 0.01 ns
 - * Maximum skew between DQS groups: 0.06 ns
 - * Average delay difference between DQ and DQS: 0.05 ns

- * Maximum skew within address and command bus: 0.02 ns
- * Average delay difference between address and command and CK: 0.01 ns

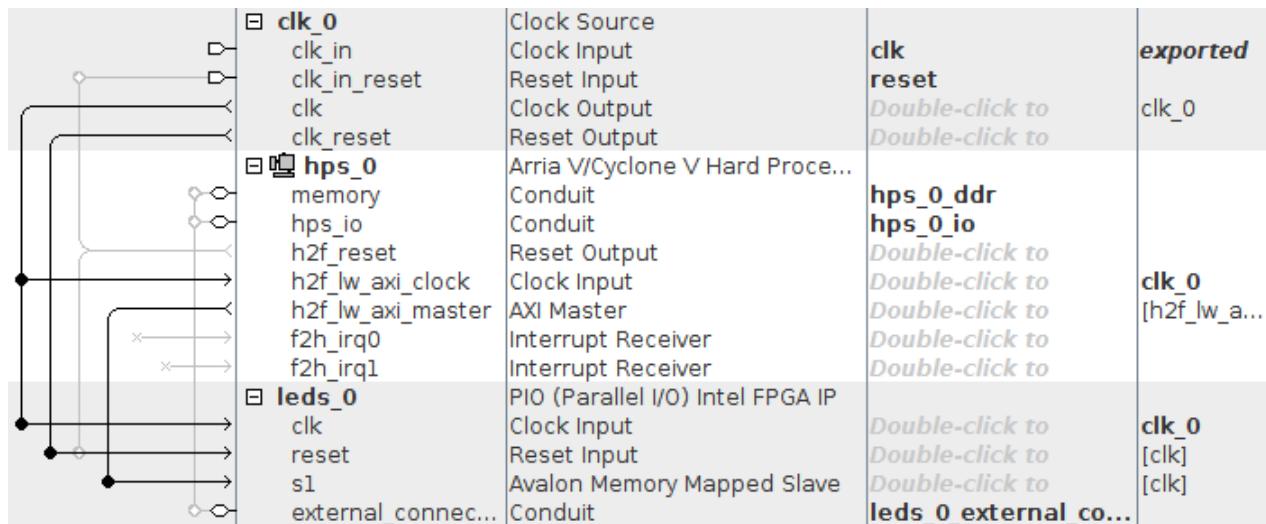
Ovim su podešavanja HPS modula završena, izabrati Finish.

- Duplim klikom u Export koloni eksportovati signale `memory` pod imenom `hps_0_ddr` i signale `hps_io` pod imenom `hps_0_io`.
- Povezati HPS sa izvorom takta kao što je prikazano na slici 10

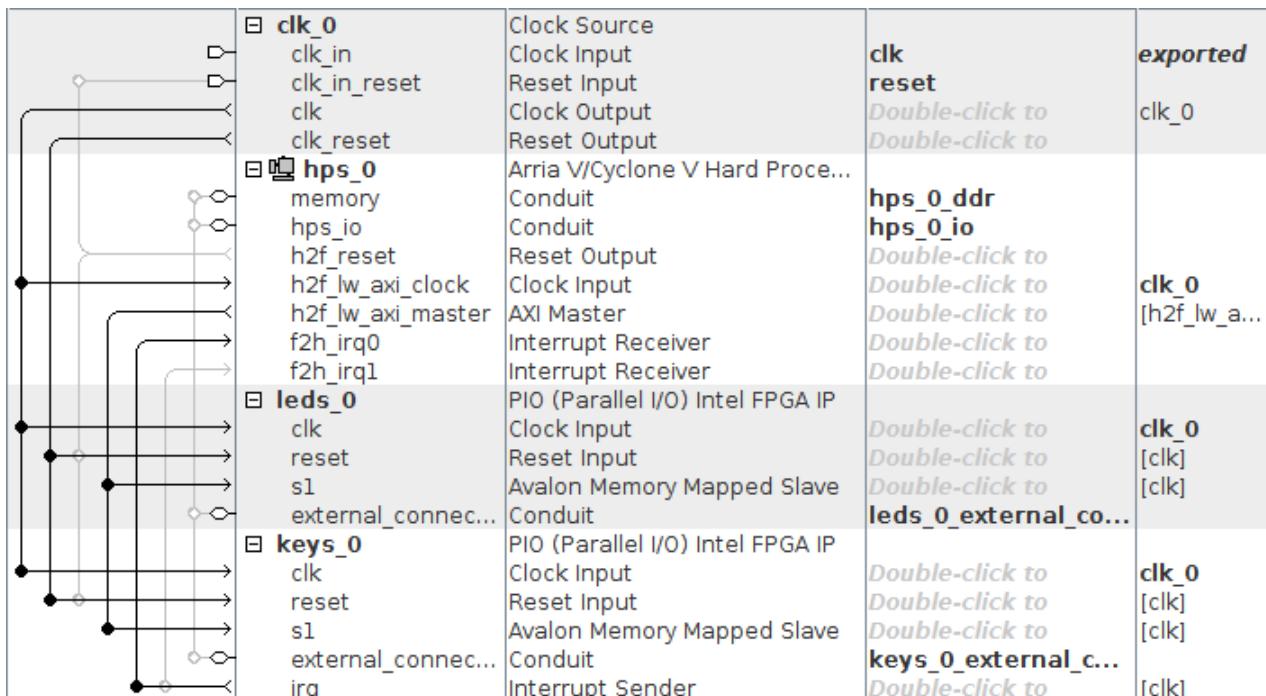


Slika 10: Povezivanje HPS i takt signala

- Iz prozora IP Catalog izabrati Processors and Peripherals > Peripherals > PIO (Parallel I/O) Intel FPGA IP. Ovim se otvara meni za podešavanje PIO IP bloka
- U podešavnjima PIO IP bloka pod Basic Settings postaviti Width: 8 i Direction: Output
- Preimenovati PIO blok u `leds_0`. Duplim klikom u Export koloni eksportovati signale `external_connection` i podesiti ime `leds_0_external_connection`.
- Povezati `leds_0` blok sa izvorom takta i resetom, zatim povezati Avalon Memory Mapped Slave pod imenom `s1` sa `hps_0` interfejsom `h2f_lw_axi_master`, kao što je prikazano na slici 11
- Ponovo iz prozora IP Catalog izabrati Processors and Peripherals > Peripherals > PIO (Parallel I/O) Intel FPGA IP. Ovim se otvara meni za podešavanje PIO IP bloka
- U podešavnjima PIO IP bloka pod Basic Settings postaviti Width: 8, Direction: Input.
- U podešavnjima PIO IP bloka pod Edge capture register uključiti opciju Synchronously capture, Edge Type podesiti na ANY, i uključiti bit-clearing for edge capture register.
- U podešavnjima PIO IP bloka pod Interrupt uključiti opciju Generate IRQ i izabrati IRQ Type: EDGE
- Preimenovati PIO blok u `keys_0`. Duplim klikom u Export koloni eksportovati signale `external_connection` i podesiti ime `keys_0_external_connection`.



Slika 11: Povezivanje **leds_0** bloka

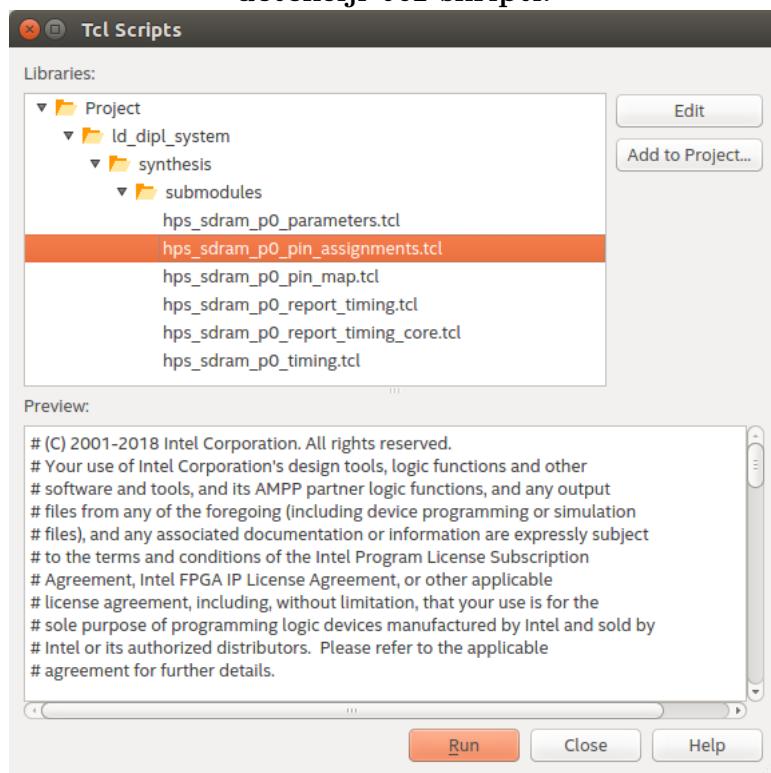


Slika 12: Povezivanje **keys_0** bloka

25. Povezati **keys_0** blok sa izvorom takta i resetom, zatim povezati Avalon Memory Mapped Slave pod imenom **s1** sa **hps_0** interfejsom **h2f_lw_axi_master** i na kraju povezati **irq** signal na **f2h_irq0** interfejs **hps_0** bloka, kao što je prikazano na slici 12
26. Duplim klikom u koloni **Base** podesiti adresu porta **s1** bloka **leds_0** i porta **s1** bloka **keys_0** kao na tabeli 1.
27. Sačuvati Platform Designer projekat izborom **File > Save** i sačuvati ga pod imenom **ld_dipl_system.qsys**
28. Trebalo bi da se pojavi obaveštenje **Save System: completed successfully**. Zatim odabratи iz menija **Generate > Generate HDL...** U novom prozoru podesiti **Create HDL design files for synthesis:** VHDL i isključiti opciju **Create block symbol file (.bsf)**. Pokrenuti generisanje klikom na **Generate**. Proces bi trebalo da se završi bez

- grešaka ali može imati upozorenja.
29. Zatvoriti Platform Designer. U prozoru Quartus-a izabrati Project > Add/Remove Files in Project... i u meniju klikom na '...' izabrati fajl ld_dipl_system/synthesis/ld_dipl_system.qip.
 30. Izabrati File > New VHDL File i novi fajl nazvati ld_dipl.vhd. Pod Project Navigator > Files desnim klikom na ld_dipl.vhd izabrati Set as Top-Level Entity.
 31. U ovom fajlu je potrebno instancirati HPS komponentu iz Platform Designer-a. Potrebno je ručno napiati ovaj fajl (primer za ovaj rad dat je u dodatku. Takođe među generisanim fajlovima nalazi se deklaracija komponente ld_dipl_system koja može biti od pomoći (fajl ~/ld_dipl/hw/ld_dipl_system/ld_dipl_system.cmp)).
 32. Izabrati Processing > Start > Start Analysis and Synthesis
 33. Izabrati Tools > Tcl Scripts...

Važno: Prozor koji se otvori mora da izgleda upravo kao na slici 13 (generisani fajlovi ne smeju biti duplirani). Ukoliko su fajlovi duplirani neophodno je zatvoriti Quartus i pokrenuti ponovo. Neke verzije Quartus-a imaju ovu grešku pri detekciji tcl skripti.



Slika 13: Ispravan izgled menija

- Izabrati hps_sdram_p0_pin_assignments.tcl i kliknuti Run. Ukoliko dođe do grešaka proveriti da li je izvršen prethodni korak.
34. Pokrenuti kompajliranje projekta izborom Processing > Start Compilation

4.1.2 Podešavanja okruženja i preuzimanje kompjajlera

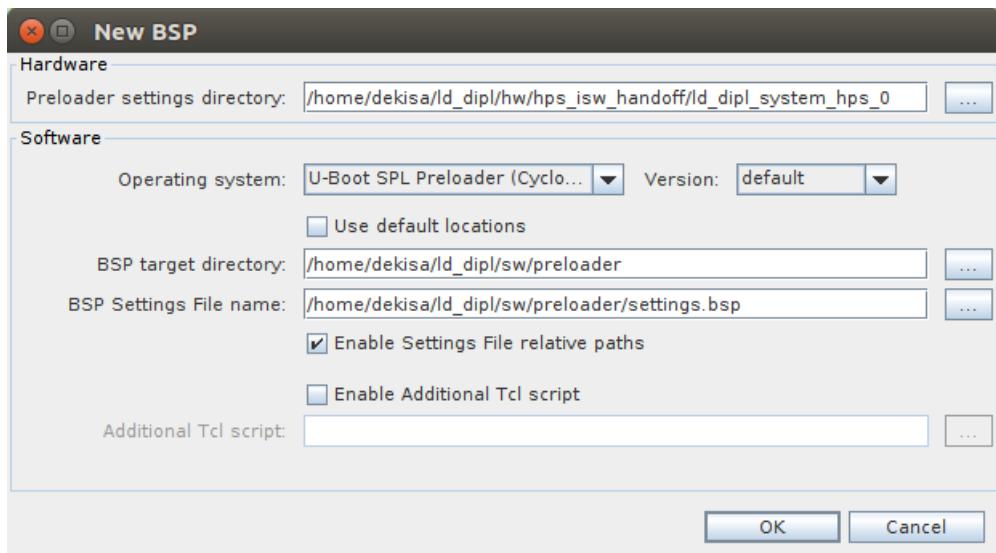
35. Preuzeti arhiv sa Linaro toolchain-om
`wget https://releases.linaro.org/components/toolchain/binaries/7.2-2017.11/arm-linux-gnueabihf/gcc-linaro-7.2.1-2017.11-x86_64_arm-linux-gnueabihf.tar.xz`
36. Otpakovati preuzeti toolchain:
`tar -xf arm-linux-gnueabihf/gcc-linaro-7.2.1-2017.11-x86_64_arm-linux-gnueabihf.tar.xz
~/ld_dipl/sw/toolchain`
37. Dodati putanju za toolchain u promenljivu okruženja PATH.
`export PATH=~/ld_dipl/sw/toolchain/bin:$PATH`
38. Podesiti promenljive okruženja
`export ARCH=arm
export CROSS_COMPILE=arm-linux-gnueabihf-`
39. Pokrenuti podešavanja Altera SOC EDS:
`cd /intelFPGA/18.0/embedded
source embedded_command_shell.sh`
40. Dodati putanju za Quartus alate u promenljivu okruženja PATH:
`export PATH=/intelFPGA_lite/18.0/quartus/bin:$PATH`

4.1.3 Generisanje RBF fajla

41. `quartus_cpf -c -o bitstream_compression=on \
~/ld_dipl/hw/ld_dipl.sof \
~/ld_dipl/sdcard/fat32/socfpga.rbf`

4.1.4 Generisanje i kompajliranje Preloader-a

42. Pokrenuti Preloader generator komadnom:
`bsp-editor`
43. Izabrati File > New HPS BSP...
44. U novom prozoru podesiti Preloader Settings Directory:
`~/ld_dipl/hw/hps_isw_handoff/ld_dipl_system_hps_0`
45. Isključiti opciju: Use default locations i podesiti BSP target directory:
`~/ld_dipl/sw/preloader`. Podešavanja bi trebalo da izgledaju kao na slici 14. Izabrati OK.
46. U podešavanjima pod `spl.boot` uključiti `FAT_SUPPORT` i ostala podešavanja ostaviti na podrazumevanim vrednostima. Izabrati Generate i po završenom generisanju zatvoriti program.
47. Izvršiti komande za kompajliranje Preloader-a
`cd ~/ld_dipl/sw/preloader
make`



Slika 14: Podešavanja bsp-editor-a

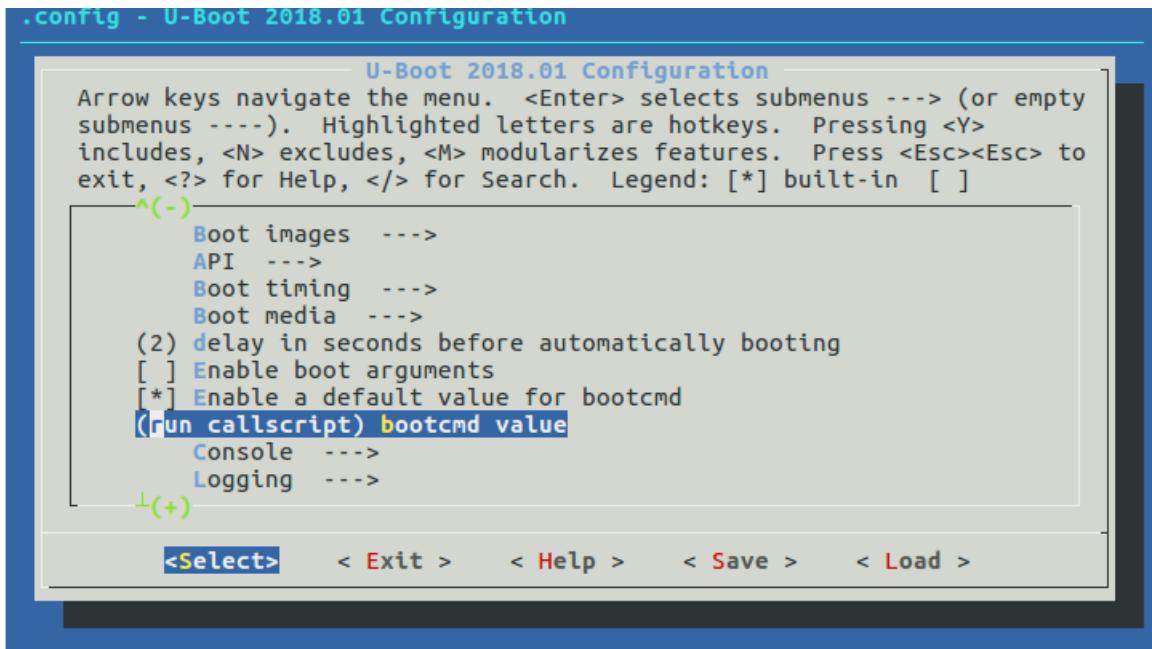
48. Kopirati kompajlirani Preloader: `cp ~/ld_dipl/sw/preloader/preloader-mkpimage.bin ~/ld_dipl/sdcard/a2/`

4.1.5 Generisanje Device Tree

49. Uporebom Alterinog alata generisati Device Tree komandom:
`sopc2dts -i /ld_dipl/hw/ld_dipl_system.sopcinfo
-o /ld_dipl/hw/ld_dipl_generated.dts`

4.1.6 U-Boot

50. Preuzeti izvorni kod projekta U-boot komandom
`git clone git://git.denx.de/u-boot.git ~/ld_dipl/sw/u-boot`
51. Promeniti radni direktorijum
`~/ld_dipl/sw/u-boot/`
52. Napraviti novu granu na osnovu verzije v2018.01 komandom:
`git checkout v2018.01 -b tmp`
53. Konfigurisati U-Boot za DE1-SoC razvojni sistem
`make socfpga_de1_soc_defconfig`
54. Otvoriti meni za konfigurisanje U-Boot
`make menuconfig`
55. U ovom meniju je potrebno podešiti da se pri pokretanju U-Boot pokrene naša skripta `callscript`. Izabratи opciju `Enable a default value for bootcmd` pritiskom tastera `space`.
56. U novoj opciji `bootcmd value` pritiskom tastera `enter` otvoriti novi meni i upisati tekst `run callscript`
 Nakon ovog podešavanja bi prozor trebalo da izgleda kao na slici 15



Slika 15: Konfiguracija U-Boot

57. Izaći iz konfiguracionog menija izborom opcije `exit` (strelica desno i zatim `enter`). Sačuvati podešavanja izborom `Yes`.

58. Otvoriti konfiguracioni .h fajl komandom:

```
vi include/configs/socfpga_de1_soc.h
```

U ovom fajlu je potrebno uneti izmene kao na slici 16

```
#define CONFIG_HW_WATCHDOG

/* Memory configurations */
#define PHYS_SDRAM_1_SIZE          0x40000000      /* 1GiB */

/* Booting Linux */
#define CONFIG_LOADADDR           0x01000000
#define CONFIG_SYS_LOAD_ADDR      CONFIG_LOADADDR

/* Ethernet on SoC (EMAC) */

/* The rest of the configuration is shared */
#include <configs/socfpga_common.h>

/* Custom env vars */
#define CONFIG_EXTRA_ENV_SETTINGS \
"scriptfile=u-boot.scr\"\\0\" \\"
"scraddr=0x2000000\"\\0\" \\"
"callscript=fatload mmc 0:1 $scraddr $scriptfile;" \
"source $scraddr\"\\0\""

#endif /* __CONFIG_TERASIC_DE1_SOC_H__ */
```

Slika 16: Izmena U-Boot konfiguracionog fajla

59. Pokrenuti kompajliranje U-Boot komandom
`make`

60. Kopirati kompajlirani U-Boot
`cp ~/ld_dipl/sw/u-boot/u-boot.img ~/ld_dipl/sdcard/fat32/u-boot.img`

61. Otvoriti novi tekstualni fajl:

```
vi u-boot.script
```

U ovom fajlu je potrebno napisati skriptu za U-Boot. Ranije u radu je dano objašnjenje skripte, dok se ceo kod nalazi u dodatku rada.
62. Konvertovati napisanu skriptu u odgovarajući format komandom:

```
mkimage -A arm -O linux -T script -a 0 -e 0 -n Boot_script -d u-boot.script
u-boot.scr
```
63. Kopirati kompajlirani U-Boot

```
cp ~/ld_dipl/sw/u-boot/u-boot.scr ~/ld_dipl/sdcard/fat32/u-boot.scr
```

4.1.7 Linuks kernel

64. Preuzeti izvorni kod Linuks kernela:

```
git clone git://github.com/torvalds/linux ~/ld_dipl/sw/linux
```
65. Promeniti radni direktorijum

```
cd ~/ld_dipl/sw/linux
```
66. Napraviti novu granu na osnovu verzije 4.6-rc2 komandom:

```
git checkout 4.6-rc2 -b tmp
```
67. Konfigurisati linuks za Cyclone V arhitekturu

```
make socfpga_defconfig
```
68. Pokrenuti kompajliranje linuksa

```
make zImage
```
69. Kopirati kompajlirani kernel u pripremni folder

```
cp arch/arm/boot/zImage ~/ld_dipl/sdcard/fat32/zImage
```

Ručno pisanje Device Tree

70. Početi od Device Tree fajla

```
cp arch/arm/boot/dts/socfpga_cyclone5_socdk.dts
arch/arm/boot/dts/socfpga_cyclone5_ld_dipl.dts
```
71. Isključiti CAN podešavanjem parametra `status = 'disabled'`. Ovaj korak je neophodan za uspešno pokretanje sistema jer DE1-SoC nema CAN. Pogledati sliku ??
72. Iz Device Tree fajla generisanog Alterinim alatom izdvajiti samo najbitnije informacije i upisati u novi fajl. Pogledati sliku ??
73. Pokrenuti kompajliranje Device Tree fajla

```
make socfpga_cyclone5_ld_dipl.dtb
```
74. Kopirati kompajlirani Device Tree Blob u pripremni folder

```
cp ~/ld_dipl/sw/linux/arch/arm/boot/dts/socfpga_cyclone5_ld_dipl.dtb
~/ld_dipl/sdcard/fat32/socfpga.dtb
```

```

        regulator-max-microvolt = <3300000>;
    };
};

/ {
    soc {
        lddipl0: lddipl@0xff200000 {
            compatible = "ld,dipl";
            reg = <0xff200000 0x00000020>;
            interrupts = <0 40 1>;
        };
    };
};

&can0 {
    status = "disabled";
};

&gmac1 {
    status = "okay";
    phy-mode = "rgmii";
    rxd0-skew-ps = <0>;
}

```

Slika 17: Izmena Device Tree fajla

4.1.8 Kompajliranje drajvera

75. Promeniti radni folder

```
cd ~/ld_dipl/sw/device_driver
```

U ovom folderu se nalazi izvorni fajl drajvera `ld_dipl.c` i odgovarajući `Makefile`

76. Podesiti promenljivu koja ukazuje na izvorni kod linuks kernela

```
export OUT=~/ld_dipl/sw/linux
```

77. Pokrenuti kompajliranje komandom

```
make
```

78. Kopirati kompajlirani drajver u pripremni direktorijum

```
cp ~/ld_dipl/sw/device_driver/ld_dipl.ko ~/ld_dipl/sdcard/ld_dipl.ko
```

4.1.9 Generisanje Root File System

79. Preuziti Buildroot `git clone git://git.busybox.net/buildroot ~/ld_dipl/sw/buildroot`

80. Promeniti radni folder

```
cd ~/ld_dipl/sw/buildroot
```

81. Napraviti novu granu na osnovu verzije 2017.08 komandom: `git checkout 2017.08 -b tmp`

82. Pokrenuti konfiguracioni meni komandom

```
make menuconfig
```

83. U konfoguracionim meniju izvršiti sledeće izmene

- (a) Target options —

- Target Architecture (ARM(little endian))
- Target Architecture Variant (cortex-A9)

- Target ABI (EABIhf)
- Floating point strategy (VFPv3)

(b) Toolchain —

- Toolchain type (External toolchain)
- Toolchain (Linaro ARM 2017.11)
- Toolchain origin (Pre-installed toolchain) —
`~/ld_dipl/sw/gcc-linaro-7.2.1-2017.11-x86_64_arm-linux-gnueabihf`

(c) Kernel —

Linux Kernel

84. Pokrenuti izvršavanje komandom

`make`

85. Kopirati fajl sistem u pripremni direktorijum

`cp ~/ld_dipl/sw/buildroot/output/images/rootfs.tar ~/ld_dipl/sdcard/rootfs.tar`

4.1.10 Priprema SD kartice

86. Ubaciti SD karticu u računar. Zaključiti pod kojim imenom se kartica pojavila u sistemskom folderu `/dev` (za primer je uzeto `/dev/sdx`)

87. Formatirati SD karticu komandom

`sudo dd if=/dev/zero of=/dev/sdx bs=512 count=1`

88. Particionisati SD karticu pokretanjem interaktivnog programa `fdisk`

`sudo fdisk /dev/sdx`

u kom treba uneti sledeće komande

`n p 3 <default> 4095 t a2`

`n p 1 <default> +32M t 1 b`

`n p 2 <default> +512M t 2 83`

`w`

89. Napraviti prazne fajl sisteme na odgovarajućim particijama

`sudo mkfs.vfat /dev/sdx1`

`sudo mkfs.ext3 -F /dev/sdx2`

90. Promeniti radni folder

`cd ~/ld_dipl/sdcard`

91. Napraviti foldere na koje će se mount-ovati SD kartica

`mkdir mountfat32`

`mkdir mountext3`

92. Mount-ovati odgovarajuće particije

`sudo mount /dev/sdx1 ~/ld_dipl/sdcard/mountfat32`

`sudo mount /dev/sdx2 ~/ld_dipl/sdcard/mountext3`

93. Napisati Preloader na odgovarajuću particiju

`sudo dd if=~/ld_dipl/sdcard/a2/preloader-mkpimage.bin of=/dev/sdx3 bs=64K seek=0`

94. Kopirati binarne fajlove za pokretanje sistema na odgovarajuću particiju
`sudo cp ~/ld_dipl/sdcard/fat32/* ~/ld_dipl/sdcard/mountfat32`
95. Otpakovati fajlsistem na SD karticu
`taf -xf rootfs.tar mountext3`
96. Kopirati modul drajvera na SD karticu
`cp ld_dipl.ko mountext3/root/home`
97. Uveriti se da su svi fajlovi uspešno kopirani komandom
`sync`
98. Unmount-ovati sve particije
`umount ~/ld_dipl/sdcard/mountfat32 umount ~/ld_dipl/sdcard/mountext3`

4.1.11 Testiranje sistema

99. Proverti da li su MSEL podešeni za konfigurisanje FPGA sa HSP (ima negde lepa slika)
100. Pruključiti DE1-SoC na napajanje
101. Povezati Mini USB kablom PC računar i port na DE1-SoC ploči koji je obležen sa UART
102. Ubaciti SD karticu u odgovarajući slot
103. Na PC računaru pokrenuti i podseiti `minicom`
`sudo minicom -s`
104. U podešavanjima `Serial port setup` uneti izmene kao na slici 18 (može se razlikovati ime `Serial device`)



Slika 18: Podešavanje `minicom`-a

105. Pokrenuti DE1-SoC pritiskom crvenog tastera

106. Nakon pokretanja sistema potrebno je ulogovati se kao korisnik sa:q
 Username:root
 Password:root
107. Učitati modul drajvera u linuks kernel komandom:
 insmod /root/home/ld_dipl.ko
108. Promeniti radni direktorijum:
 cd /sys/bus/platform/
109. Uveriti se da su se pojavili sistemski fajlovi komandom ls
110. Uključiti LE diode upisom u fajl leds i omogućiti pristizanje prekidnog zahteva upisom u fajl irq_mask

4.2 Izvorni kodovi

4.2.1 Drajver

```
/* ld_dipl.c */

#include <linux/init.h>
#include <linux/module.h>
#include <linux/io.h>
#include <linux/interrupt.h>
#include <linux/uaccess.h>
#include <linux/platform_device.h>
#include <linux/of.h>

#define DEVICE_FILE_NAME          "ld_dipl"
#define DRIVER_NAME               "lddipldrv"

#define LED_OFFSET                 0
#define KEYS_OFFSET                16
#define DATA_OFFSET                0
#define INTERRUPT_MASK_OFFSET     8
#define EDGE_CAPTURE_OFFSET        12

static struct platform_driver ld_dipl_driver;

/* globalne promenljive */
static int g_ld_dipl_driver_irq;
static void *g_ld_dipl_driver_base_addr;
static int g_driver_mem_base_addr;
static int g_driver_mem_size;

static ssize_t irq_mask_show(struct device_driver *driver, char *buf)
{
    uint32_t value;
    value = ioread32(g_ld_dipl_driver_base_addr + KEYS_OFFSET
                     + INTERRUPT_MASK_OFFSET);
    return scnprintf(buf, PAGE_SIZE, "mask= %u\n", value);
}

static ssize_t irq_mask_store(struct device_driver *driver, const char *buf,
                           size_t count)
{
    uint32_t value;
    sscanf(buf, "%u", &value);
```

```

        iowrite32(value, g_ld_dipl_driver_base_addr + KEYS_OFFSET
                  + INTERRUPT_MASK_OFFSET);
    return count;
}

DRIVER_ATTR(irq_mask, (S_IWUSR | S_IRUSR), irq_mask_show, irq_mask_store);

static ssize_t irq_flag_show(struct device_driver *driver, char *buf)
{
    uint32_t value;
    value = ioread32(g_ld_dipl_driver_base_addr + KEYS_OFFSET
                      + EDGE_CAPTURE_OFFSET);
    return scnprintf(buf, PAGE_SIZE, "flags= %u\n", value);
}

static ssize_t irq_flag_store(struct device_driver *driver, const char *buf,
                             size_t count)
{
    uint32_t value;
    sscanf(buf, "%u", &value);
    iowrite32(value, g_ld_dipl_driver_base_addr + KEYS_OFFSET
              + EDGE_CAPTURE_OFFSET);
    return count;
}

DRIVER_ATTR(irq_flag, (S_IWUSR | S_IRUSR), irq_flag_show, irq_flag_store);

static ssize_t keys_show(struct device_driver *driver, char *buf)
{
    uint32_t value;
    value = ioread32(g_ld_dipl_driver_base_addr + KEYS_OFFSET
                      + DATA_OFFSET);
    return scnprintf(buf, PAGE_SIZE, "keys= %u\n", value);
}

DRIVER_ATTR(keys, (S_IRUSR), keys_show, NULL);

static ssize_t leds_store(struct device_driver *driver, const char *buf,
                        size_t count)
{
    uint32_t value;
    sscanf(buf, "%u", &value);
    iowrite32(value, g_ld_dipl_driver_base_addr + LED_OFFSET +
              DATA_OFFSET);
    return count;
}

DRIVER_ATTR(leds, (S_IWUSR), NULL, leds_store);

static irqreturn_t ld_dipl_isr(int irq, void *data)
{
    uint32_t value;

    value = ioread32(g_ld_dipl_driver_base_addr + KEYS_OFFSET
                      + EDGE_CAPTURE_OFFSET);

    pr_info("irq received: %u\n", value);

    iowrite32(value, g_ld_dipl_driver_base_addr + KEYS_OFFSET
              + EDGE_CAPTURE_OFFSET);
}

```

```

        return IRQ_HANDLED;
    }

static struct of_device_id ld_dipl_of_match[] = {
{
    .compatible = "ld,dipl"
},
{ /* end of table */ }
};

MODULE_DEVICE_TABLE(of, ld_dipl_of_match);

static int ld_dipl_probe(struct platform_device *pdev)
{
    int ret;
    struct resource *res;
    struct resource *driver_mem_region;

    pr_info("probe\u201eenter\n");

    ret = -EINVAL;

    /* get memory resource */
    res = platform_get_resource(pdev, IORESOURCE_MEM, 0);
    if (res == NULL) {
        pr_err("IORESOURCE_MEM ,\u00d70does\u201enot\u201eexist\n");
        return ret;
    }

    g_driver_mem_base_addr = res->start;
    g_driver_mem_size = resource_size(res);

    /* reserve our memory region */
    driver_mem_region = request_mem_region(g_driver_mem_base_addr,
                                            g_driver_mem_size,
                                            "demo_driver_hw_region")
                        ;
    if (driver_mem_region == NULL) {
        pr_err("request_mem_region\u201efailed\n");
        return ret;
    }

    /* ioremap memory region */
    g_ld_dipl_driver_base_addr = ioremap(g_driver_mem_base_addr,
                                          g_driver_mem_size);
    if (g_ld_dipl_driver_base_addr == NULL) {
        pr_err("ioremap\u201efailed\n");
        goto bad_exit_release_mem_region;
    }

    /* get interrupt resource */
    g_ld_dipl_driver_irq = platform_get_irq(pdev, 0);
    if (g_ld_dipl_driver_irq < 0) {
        pr_err("invalid\u201eIRQ\n");
        goto bad_exit_iounmap;
    }
    pr_info("interrupt\u201eis:\u201e%d\n", g_ld_dipl_driver_irq);

    /* register interrupt handler */

```

```

    ret = request_irq(g_ld_dipl_driver_irq,
                      ld_dipl_isr,
                      0,
                      ld_dipl_driver.driver.name,
                      &ld_dipl_driver);
    if (ret < 0) {
        pr_err("unable to request IRQ\n");
        goto bad_exit_iounmap;
    }

/* create the sysfs entries */
ret = driver_create_file(&ld_dipl_driver.driver,
                        &driver_attr_irq_mask);
if (ret != 0) {
    pr_err("failed to create irq_mask sysfs entry");
    goto bad_exit_remove_irq_mask;
}

ret = driver_create_file(&ld_dipl_driver.driver,
                        &driver_attr_keys);
if (ret != 0) {
    pr_err("failed to create keys sysfs entry");
    goto bad_exit_remove_keys;
}

ret = driver_create_file(&ld_dipl_driver.driver,
                        &driver_attr_leds);
if (ret != 0) {
    pr_err("failed to create leds sysfs entry");
    goto bad_exit_remove_leds;
}

ret = driver_create_file(&ld_dipl_driver.driver,
                        &driver_attr_irq_flag);
if (ret != 0) {
    pr_err("failed to create irq_flag sysfs entry");
    goto bad_exit_remove_irq_flag;
}
pr_info("probe exit success\n");
return 0;

bad_exit_remove_irq_flag:
    driver_remove_file(&ld_dipl_driver.driver,
                      &driver_attr_irq_flag);
bad_exit_remove_leds:
    driver_remove_file(&ld_dipl_driver.driver,
                      &driver_attr_leds);
bad_exit_remove_keys:
    driver_remove_file(&ld_dipl_driver.driver,
                      &driver_attr_keys);
bad_exit_remove_irq_mask:
    driver_remove_file(&ld_dipl_driver.driver,
                      &driver_attr_irq_mask);
bad_exit_freeirq:
    free_irq(g_ld_dipl_driver_irq, &ld_dipl_driver);
bad_exit_iounmap:
    iounmap(g_ld_dipl_driver_base_addr);
bad_exit_release_mem_region:
    release_mem_region(g_driver_mem_base_addr, g_driver_mem_size);

```

```

        pr_info("probe\u201eexit\u201efail\n");
        return ret;
    }

static int ld_dipl_remove(struct platform_device *pdev)
{
    pr_info("remove\u201eenter\n");
    driver_remove_file(&ld_dipl_driver.driver,
                       &driver_attr_irq_flag);
    driver_remove_file(&ld_dipl_driver.driver,
                       &driver_attr_leds);
    driver_remove_file(&ld_dipl_driver.driver,
                       &driver_attr_keys);
    driver_remove_file(&ld_dipl_driver.driver,
                       &driver_attr_irq_mask);
    driver_remove_file(&ld_dipl_driver.driver,
                       &driver_attr_irq_flag);
    driver_remove_file(&ld_dipl_driver.driver,
                       &driver_attr_leds);
    driver_remove_file(&ld_dipl_driver.driver,
                       &driver_attr_keys);
    driver_remove_file(&ld_dipl_driver.driver,
                       &driver_attr_irq_mask);
    free_irq(g_ld_dipl_driver_irq, &ld_dipl_driver);

    iounmap(g_ld_dipl_driver_base_addr);

    release_mem_region(g_driver_mem_base_addr, g_driver_mem_size);

    pr_info("remove\u201eexit\n");
    return 0;
}

static struct platform_driver ld_dipl_driver = {
    .probe = ld_dipl_probe,
    .remove = ld_dipl_remove,
    .driver = {
        .name = DRIVER_NAME,
        .of_match_table = ld_dipl_of_match,
    },
};

static int __init ld_dipl_init(void)
{
    int ret;

    pr_info("init\u201eenter\n");

    ret = platform_driver_register(&ld_dipl_driver);
    if (ret != 0)
        pr_err("platform_driver_register\u201ereturned\u201d%d\u201d\n", ret);
    pr_info("init\u201exit\n");

    return ret;
}

static void __exit ld_dipl_exit(void)
{
    pr_info("ld_dipl_exit\u201eenter\n");
    platform_driver_unregister(&ld_dipl_driver);
}

```

```

        pr_info("ld_dipl_exit\n");
}

module_init(ld_dipl_init);
module_exit(ld_dipl_exit);

MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("ETF Diploma Thesis FPGA Driver and Device");
MODULE_AUTHOR("Dejan Lukic");

```

4.2.2 Skripta za U-Boot

```

echo --- Reseting env variables...---

# reset environment variables to default
env default -a

echo --- Setting env variables...---

# Set kernel image
setenv bootimage zImage;

# adress to wich the device tree will be loaded
setenv fdtaddr 0x00000100

# set device tree image
setenv fdtimage socfpga.dtb;

#set kernel boot arguments, boot kernel
setenv mmcboot 'setenv bootargs mem=1024M console=ttyS0,115200 root=${mmcroot} rw rootwait; bootz ${loadaddr} - ${fdtaddr}';

#load linux kernel image and device tree to memory
setenv mmcload 'mmc_rescan; ${mmcloadcmd} mmc 0:${mmcloadpart} ${loadaddr} ${bootimage}; ${mmcloadcmd} mmc 0:${mmcloadpart} ${fdtaddr} ${fdtimage}'

#command to be executed to read from sdcard
setenv mmcload fatload

# sdcard fat32 partition number
setenv mmcloadpart 1

# sdcard ext3 identifier
setenv mmcroot /dev/mmcblk0p2

#standard io
setenv stderr serial
setenv stdin serial
setenv stdout serial

#save environment to sdcard
saveenv

echo --- Programming FPGA...---

#load rbf from fat partition to memory
fatload mmc 0:1 ${fpgadata} socfpga.rbf;

# program fpga
fpga load 0 ${fpgadata} ${filesize}

```

```

# enable h2f, f2h, lw_h2f
bridge enable;

echo --- Booring Linux.....
#load kernel and device tree to memory
run mmcload;

# set bootargs, boot kernel
run mmcboot;

```

4.2.3 Device Tree

```

#include "socfpga_cyclone5.dtsi"

/ {
    model = "Altera\u2022SOCFPGA\u2022Cyclone\u2022V\u2022SoC\u2022Development\u2022Kit";
    compatible = "altr,socfpga-cyclone5", "altr,socfpga";

    chosen {
        bootargs = "earlyprintk";
        stdout-path = "serial0:115200n8";
    };

    memory {
        name = "memory";
        device-type = "memory";
        reg = <0x0 0x40000000>; /* 1GB */
    };

    aliases {
        /* this allow the ethaddr uboot environment variable
         * contents
         * to be added to the gmac1 device tree blob.
         */
        ethernet0 = &gmac1;
    };

    regulator_3_3v: 3-3-v-regulator {
        compatible = "regulator-fixed";
        regulator-name = "3.3V";
        regulator-min-microvolt = <3300000>;
        regulator-max-microvolt = <3300000>;
    };
};

/ {
    soc {
        lddipl0: lddipl@0xff200000 {
            compatible = "ld,dipl";
            reg = <0xff200000 0x00000020>;
            interrupts = <0 40 1>;
        };
    };
};

&gmac1 {
    status = "okay";
    phy-mode = "rgmii";
    rxd0-skew-ps = <0>;
}

```

```

rxd1-skew-ps = <0>;
rxd2-skew-ps = <0>;
rxd3-skew-ps = <0>;
txen-skew-ps = <0>;
txc-skew-ps = <2600>;
rxdv-skew-ps = <0>;
rxc-skew-ps = <2000>;
};

&gpi01 {
    status = "okay";
};

&i2c0 {
    status = "okay";

    eeprom@51 {
        compatible = "atmel,24c32";
        reg = <0x51>;
        pagesize = <32>;
    };

    rtc@68 {
        compatible = "dallas,ds1339";
        reg = <0x68>;
    };
};

&mmc0 {
    cd-gpios = <&portb 18 0>;
    vmmc-supply = <&regulator_3_3v>;
    vqmmc-supply = <&regulator_3_3v>;
    status = "okay";
};

&usb1 {
    status = "okay";
};

```

Literatura

- [1] DE1-SoC User Manual(rev.E Board).pdf, Terasic, <https://www.terasic.com.tw/cgi-bin/page/archives>
- [2] Albert Einstein. *Zur Elektrodynamik bewegter Körper.* (German) [*On the electrodynamics of moving bodies*]. Annalen der Physik, 322(10):891–921, 1905.
- [3] Knuth: Computers and Typesetting,
<http://www-cs-faculty.stanford.edu/~uno/abcde.html>