SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 1246

RAZVOJ DEMONSTRACIJSKIH APLIKACIJA NA ZYNQ-7000 SOC ARHITEKTURI POD OPERACIJSKIM SUSTAVOM LINUX

Petar Matković

Zagreb, lipanj 2015.

DIPLOMSKI ZADATAK br. 1246

Pristupnik: **Petar Matković (0036456409)**
Studij: Elektrotehnika i informacijska tehnologija
Profil: Elektroničko i računalno inženjerstvo

Zadatak: **Razvoj demonstracijskih aplikacija na Zynq-7000 SoC arhitekturi pod operacijskim sustavom Linux**

Opis zadatka:

U okviru diplomskog rada potrebno je instalirati odgovarajuću inačicu Linux distribucije na razvojni sustav ZedBoard koji je temeljen na Zynq-7000 System-on-Chip (SoC) komponenti. U kombinaciji s implementacijom sklopovskih modula u FPGA programabilnoj matrici ovog sklopa potrebno je razviti pokazne aplikacije za laboratorijske vježbe na Linux operacijskom sustavu koje iskorištavaju procesorske i programabilne resurse moderne Zynq SoC arhitekture. Prva aplikacija mora biti jednostavna manipulacija LED diodama sa svrhom demonstracije postupaka podizanja operacijskog sustava i aplikacija na ugradbenim SoC arhitekturama, te pristupa projektiranom sklopovlju iz korisničke aplikacije pod operacijskim sustavom. Druga aplikacija mora demonstrirati napredne mogućnosti modernog SoC dizajna s Linux distribucijom.

Zadatak uručen pristupniku: 13. ožujka 2015.
Rok za predaju rada: 30. lipnja 2015.

Mentor:
Prof. dr. sc. Davor Petrinović

Djelovođa:
Izv. prof. dr. sc. Dražen Jurišić

Predsjednik odbora za diplomski rad profila:
Prof. dr. sc. Mladen Vučić

Sadržaj

1. Uvod

Moderna sklopovska rješenja ostvarena kao sustav na čipu - SoC (engl. *System on Chip*) integriraju jedan ili više procesora opće namjene s programabilnom logikom i mnoštvom funkcionalnih podsustava. Takve snažne i modularne arhitekture impliciraju korištenje naprednog koncepta izvođenja pod određenim operacijskim sustavom. Taj višezadaćni (engl. *multitasking*) režim efektivno alocira, povezuje i iskorištava resurse ovakve arhitekture što rezultira unaprijeđenim performansama, visokom razinom sigurnosti i korisniku orijentiranim pristupom.

Naša ciljana arhitektura je Zynq-7000 generacija sustava na čipu razvijena od firme Xilinx koja prvenstveno integrira dvojezgreni ARM procesor s 28-nanometarskim poljem programabilne logike. Ostvarena kroz razvojnu pločicu ZedBoard iskorištava brojne periferije i tako omogućuje širok spektar aplikacija. Zbog otvorenosti programske podrške pa samim time i široke distribucije na Zynq-7000 SoC dizajnima, instalirat ćemo Linux inačicu operacijskog sustava sa svrhom izvođenja aplikacija.

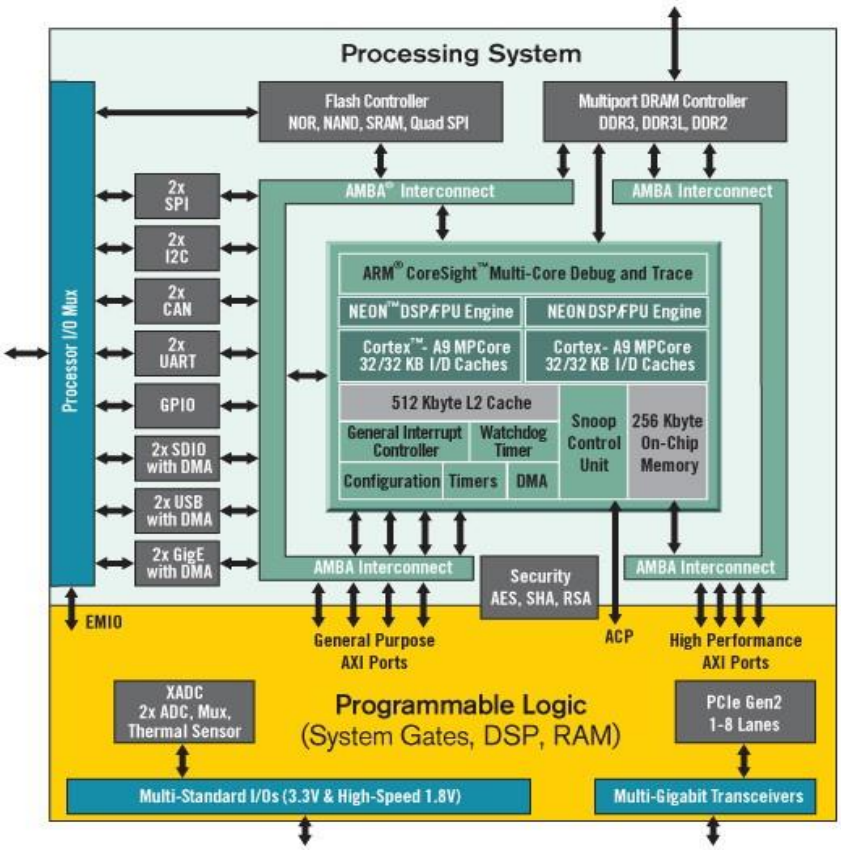
Ponekad zahtjevne faze i postupci razvijanja SoC aplikacija pod operacijskim sustavom zanimljivi su sa iskustvenog pogleda računalnog inženjerstva, a prva takva iskustva moguće je ostvariti pomoću laboratorijskih vježbi. Zato je i cilj ovog rada razvijanje i pokretanje upravo demonstracijskih aplikacija koje bi na zanimljiv način studentima pokazale mogućnosti modernog SoC dizajna s Linux distribucijom. Stoga ćemo prvo na primjeru LED aplikacija prikazati cjelovit postupak podizanja Linuxa na SoC-u. Taj postupak uključuje sklopovski razvoj i implementaciju dizajna, prilagodbu programske podrške, kompilaciju Linux inačice, pisanje *drivera* i aplikacija te upogonjavanje istih. Zatim ćemo na primjeru ZedBoard samostalnog računala i pripadnih aplikacija koje se izvršavaju pod grafičkom Linux distribucijom demonstrirati napredne mogućnosti Zynq-7000 arhitekture.

2. Zynq-7000 ciljana platforma

Zynq-7000 je moderna širokoaplikacijska SoC arhitektura razvijena od tvrtke Xilinx korištena kao jezgra tradicionalno hardverskih razvojnih pločica raznih firmi. Najpoznatije takve su ZedBoard, MicroZed, PicoZed, Zybo, Xilinx ZC706 i ZC702 uz koje dolazi i programska podrška čak i u obliku virtualnih razvojnih platformi kao što je to slučaj za Xilinxove pločice. U ovom poglavlju razmotrit ćemo Zynq-7000 SoC arhitekturu, prateće alate, funkcionalnost ciljane ZedBoard razvojne pločice te aplikativne domene.

2.1 Zynq-7000 SoC arhitektura i alati

Dizajn sveprogramabilne Zynq-7000 SoC generacije integrira brzi dvojezgreni procesorski sustav dva ARM Cortex-A9 MPCore procesora s naprednom 28 nm FPGA logikom i raznim drugim podsustavima. Blok shema Zynq-7000 arhitekture s osnovnom podjelom na dva djela, procesorski i programabilni, dana je slikom 1. Ova generacija SoC-ova se odlikuje optimizacijom veličine, potrošnje i cijene.



Slika 1: *Zynq-7000 arhitektura [14]*

Svaka procesorska jezgra sadrži NEON ekstenziju pomičnog zareza. Procesor ima hijerarhijski organiziranu internu memoriju - L1 (po 32 KB *cache* memorije za instrukcije i podatke) i L2 (512 KB) s dodatnih 256 KB *on-chip* memorije. Također podržani su dodaci vanjske memorije (dinamički DDR, statički RAM i *Flash* memorija). L2 memorija je podržana PL310 *cache* kontrolerom prekomjernog podatkovnog prometa. Za proširenje ugradbenih performansi koriste se mnoge uobičajene periferije kao što su SPI, I2C, CAN, UART, GPIO, USB, Ethernet i SD koje su s okolinom povezane preko procesorskog ulazno-izlaznog multipleksora (MIO). Koristi se 8 DMA kanala za prijenos podataka od kojih 4 pripadaju programabilnoj logici [11].

Programabilna logika se sastoji od konfigurabilnih logičkih blokova (CLB), memorijskih RAM blokova, programabilnih ulazno-izlaznih blokova, kontrolera takta i DSP odsječaka za efikasnu obradu signala. Programabilna logika sadrži XADC - efikasni Xilinxov AD pretvornik kojeg zapravo čine dva AD pretvornika brzine uzorkovanja 1 Msample/s. Za potpunu i parcijalnu konfiguraciju Zynq koristi PCAP (engl. *Processor Configuration Access Port*) sa DevC jedinicama (engl. *Device Configuration Unit*) za prijenos slijeda bitova - *bitstreama*. PCAP i DevC omogućuju također efikasnu dinamičku rekonfiguraciju programabilne logike. Ovisno o inačici Zynq-7000 generacije programabilna logika može sadržavati i do 16 serijskih brzih primopredajnika - *transceivera* [3].

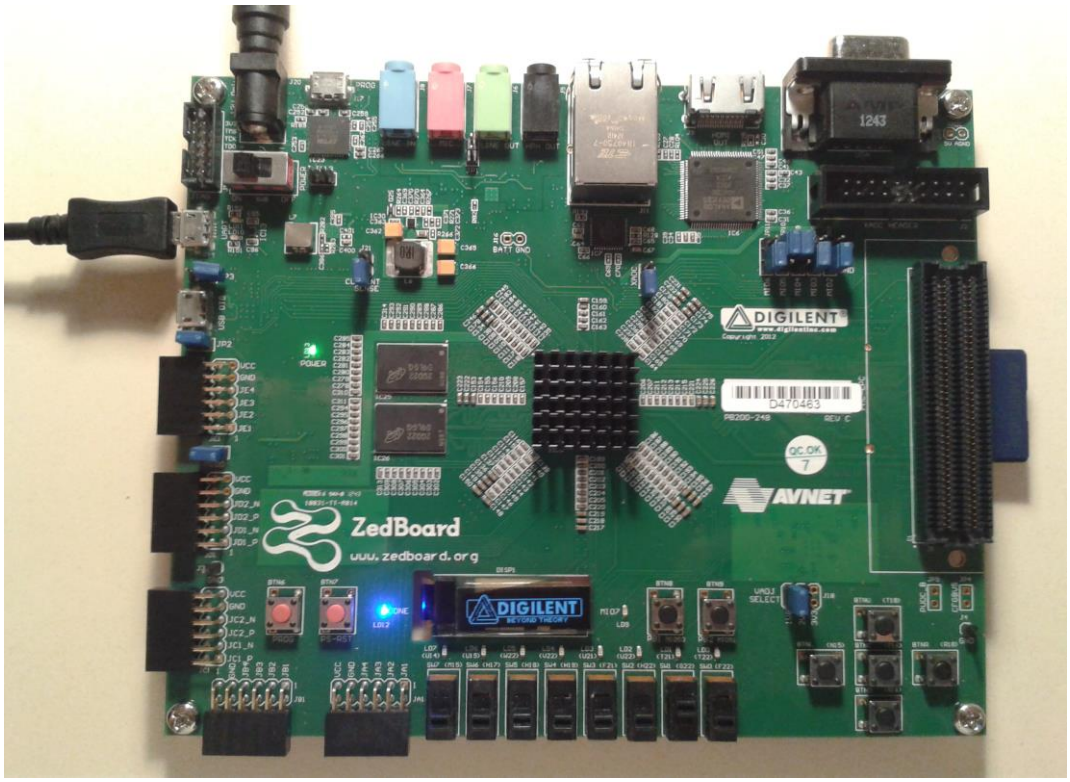
Procesor, periferije i programabilna logika podatke međusobno razmjenjuju preko AMBA-AXI sabirničkog sustava (engl. *Advanced Micro-controller Bus Architecture - Advanced Extensible Interface*). Sigurnost podataka se provodi za procesorski i programabilni dio od strane istog bloka, a on koristi RSA autentikaciju sa ključem određenim kod prvog podizanja (engl. *Boot*) te AES i SHA enkripciju i autentikaciju za iduće procesorske *bootove* i konfiguracije logike [11].

Xilinx za svoje proizvode osigurava sofisticirane razvojne alate za prilagodbu sklopovlja i stvaranje programske podrške. Za Zynq-7000 generaciju su na raspolaganju **ISE Design Suite - WebPack** sa paketom **XPS** (engl. *Xilinx Platform Studio*) te **Vivado Design Suite**. Oba alata u sebi imaju uključen Xilinx **SDK** (engl. *Software Development Kit*) alat za razvoj programske podrške. Vivado i XPS su zapravo sustavni razvojni alati visoke apstrakcijske razine te omogućuju HLS sintezu (engl. *High Level Synthesis*), a za implementaciju naprednijih sustava koriste razne aplikacijske dizajne i ekspertize na razini

cjelokupnog sustava. Iako kreću od razine sustava, alati vrše mnogobrojne RTL optimizacije programabilne logike te automatske preinake starih sustava i komponenti u nove. Vivado se sugerira za razvoj Zynq platformi, ali se za tri najmanja Zynq-7000 uređaja, među kojima je i Zynq jezgra ZedBoard razvojne pločice, može koristiti ISE Design Suite. SDK se koristi za generiranje prvog sustavnog podizača - FSBL-a (engl. *First System Boot Loader*) te dalje za kompilaciju BOOT.BIN slike potrebne za podizanje operacijskog sustava na SoC-u. Od ostalih alata za razvoj Zynq sustava često se koriste ARM Development Studio, Sourcery CodeBench, Wind River Workbench te Matlab kao primjerice polazna točka sustavnog blok modeliranja [3].

2.2 ZedBoard razvojna pločica

Za razvoj i evaluaciju Zynq-7000 programabilnih SoC-ova koristit ćemo ZedBoard razvojnu pločicu firme Avnet (slika 2). Ona iskorištava i povezuje blokove i periferne module Zynq-7000 čipa s paletom namjenskih konektora postajući tako široka platforma za prototipne ugradbene i ostale konceptualne sustavne dizajne. ZedBoard je otvorena razvojna pločica za koju su sheme, PCB *layouti*, sklopovski opisi, upute i referentni dizajni dostupni na zajedničkoj *web* stranici [13].

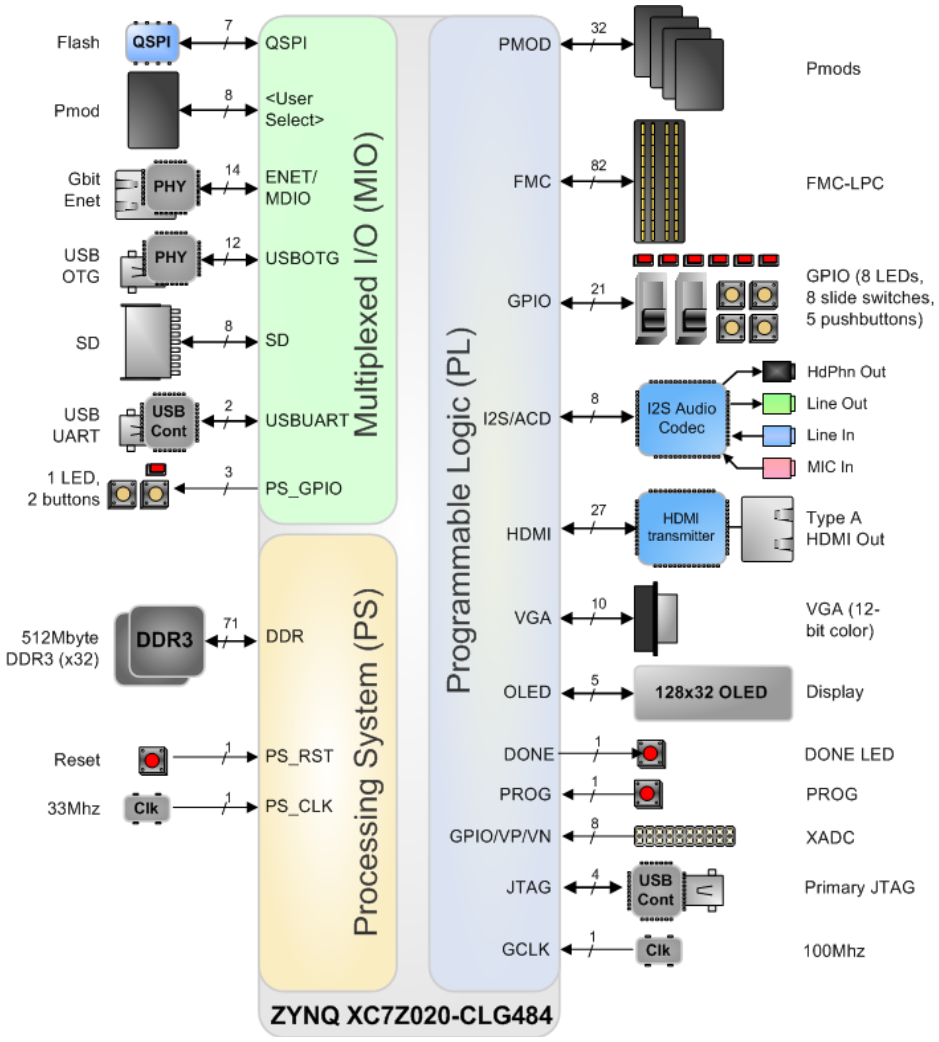


Slika 2: *ZedBoard pločica*

ZedBoard je temeljen na Z-7020 uređaju Zynq-7000 sveprogramabilne SoC generacije koji se u odnosu na druge uređaje razlikuje u sljedećim karakteristikama: maksimalna frekvencija je 866 MHz, ima 85.000 logičkih ćelija, 53.200 LUT-ova, 106.400 Flip-Flopova, 220 DSP odsječaka što omogućuje do 276 GMAC operacija po sekundi. Ovaj uređaj je po programabilnoj logici ekvivalentan Atrix-7 FPGA uređaju. Na slici 3 je prikazana sklopovska blok shema ZedBoard pločice sa Zynq-7000 jezgrom i ostalim funkcionalnim blokovima [11]:

- **Memorija** - Procesorski sustav (PS) podržan je s 512 MB dinamičke DDR memorije. Na 7 pinova ulazno-izlaznog Quad SPI kontrolnog bloka spojena je Spansion FL-S serijska *Flash* memorija veličine 256 Mb za brzu konfiguraciju FPGA i podizanje procesora sa brzinom do 400 Mb/s. Za podizanje operacijskog sustava koristi se priključak SD kartice s 4 GB.
- **Komunikacija** - ZedBoard podržava 10/100/1000 Gigabitni Ethernet protokol pomoću Ethernet kontrolera, USB-UART niskopotrošnu komunikaciju sa Cypress USB-UART mosnim kontrolerom koji stvara virtualni ACM COM Port.
- **Takt** - vanjski izvori takta: za procesorski dio (PS) 33.33 MHz, dok je za programabilni dio (PL) 100 MHz.
- **Prikaznici** - HDMI 1080p60 izlaz sa 225MHz ADV7511 HDMI transmiterom, VGA izlaz 12-bitne rezolucije boje i 128x32 OLED prikaznik s organskim filmom.
- **Zvuk** - 96 kHz ADAU1761 niskopotrošni, 24-bitni *audio codec* s integriranim PLL-om i izvodima (Line In, Line Out, MIS In, HdPhn Out).
- **Konfiguracija i debugiranje** - USB-JTAG sučelje sa JTAG konektorom.
- **Konektori** - FMC (engl. *FPGA Mezzanine Card*) konektor sa 68 priključaka ili 34 diferencijalna priključka za spajanje brzih analognih modula za bežičnu komunikaciju (primjerice OFDM u fizičkom sloju). Također ZedBoard sadrži 5 Pmod 2x6 konektora za spajanje dodatnih perifernih modula, najčešće za analognu i mješovitu obradu signala kao što su ADC ili DAC pretvornik, ali i razni MEMS-ovi i drugi senzori. Konačno tu je i AMS (engl. *Agile Mixed Signaling*) konektor za evaluaciju jezgre Xilinx ADC (XADC) pretvornika.

- **Napajanje** - Vanjsko DC napajanje od 12 V sa DC-DC pretvornicima na 1.3 V, 1.5 V, 1.8 V, 2.5 V i 3.3 V, sklopovlje za naponsku i strujnu zaštitu kao što su osigurači i *PolyZen* uređaj sa Zener diodom i resetabilnim energetskim *PolySwitch* uređajem.
- **Ostalo** - ZedBoard ima 8 korisničkih LED dioda i ostale signalne LED diode, 7 općih tipkala te tipkala za PROG i Reset, 8 DIP sklopki i sklopku za napajanje. Za razne modove i preklapanja napona napajanja ZedBoard sadrži mnoštvo kratkospojnika.

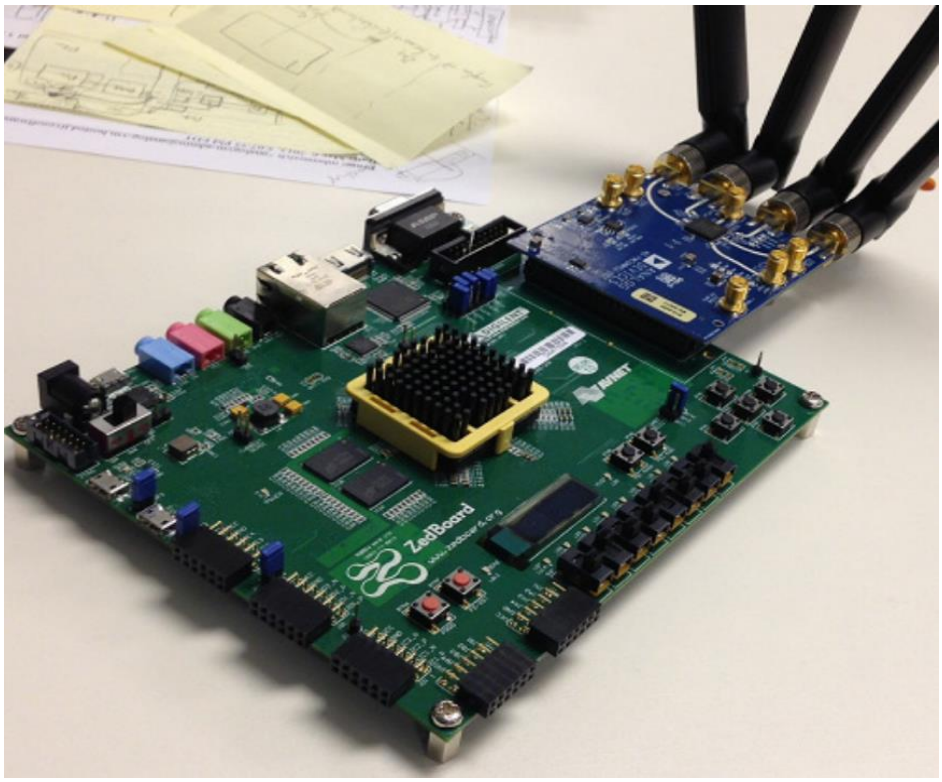


Slika 3: *Sklopovska blok shema ZedBoard razvojne pločice [10]*

2.3 Aplikacije

Zynq-7000 snagovna i arhitekturalna rješenja omogućuju razvijanje velikog niza aplikacija. Mogućnošću podizanja operacijskog sustava zadovoljavaju se sigurnosne i

pouzdane pretpostavke za korištenje u industriji, medicini, prometu i komunikaciji. Snažna ARM jezgra osigurava implementacije algoritama obrade slike, prepoznavanja objekta, kodiranja videa i zvuka [12]. Obrada signala potpomognuta je u programabilnoj logici koja sa svojim DSP odsječcima i AD pretvornicima osigurava optimalnu implementaciju i brzo izvođenje FIR i IIR filtriranja (inherentna zbrajala i množila u DSP odsječcima). Ranije spomenuti PCAP i DevC blokovi za bržu rekonfiguraciju omogućuju razne adaptivne aplikacije u komunikaciji (adaptacija snage ili konstelacije, pobijanje jeke), kriptografiji i raznim predikcijama.



Slika 4: *ZedBoard sa FMC modulom - programski definiran radio [4]*

Zynq-7000 arhitektura tako može služiti kao jezgra profesionalnih kamera, sustava video konferencija, monitora, multifunkcionalnih printera, video nadzora, mobilnih i žičnih mreža (primjerice LTE), sustava strojnog vida (engl. *Machine Vision*) ili medicinskih endoskopa [14]. Na slici 4 je prikazan bežični 802.11 komunikacijski čvor koji u fizičkom sloju može programski implementirati OFDM (engl. *Orthogonal Frequency-Division Multiplexing*), proširenje direktnim slijedom - DSSS (engl. *Direct-Sequence Spread Spectrum*) ili skakanje frekvencije - FHSS (engl. *Frequency Hopping Spread Spectrum*). On je realiziran ZedBoard pločicom na čiji se FMC konektor povezuje analogni FMC modul RF odašiljača s antenama.

3. SoC operacijski sustavi

Dizajn modernih sustava na čipu svojim naprednim memorijskim i procesnim karakteristikama omogućuje podizanje i distribuciju operacijskog sustava pod kojim se odvija željena aplikacija. No distribucija operacijskih sustava nije samo mogućnost takve arhitekture već i potreba činjenicom da je SoC dizajn u velikoj mjeri složen i slojevit te integrira mnoštvo funkcionalnih podsustava čija uključivanja u sustav moraju biti pouzdana, podaci istovremeno zaštićeni, a zajednički resursi osigurani od kolizija. S aspekta korisnika, operacijski sustavi premošćuju insuficijenciju detaljnog poznavanja arhitekture te se orijentiraju prema razvijanju aplikacija sa višeg stupnja apstrakcije. Upravo Linux operacijski sustav zbog otvorenosti programske potpore i validnosti postaje naš ciljani operacijski sustav za Zynq-7000 SoC.

3.1 Raspodjela zadataka i operacijski tipovi

Računarska snaga sustava na čipu se kapitalizira upravo u velikom inherentnom paralelizmu. Taj paralelizam se ostvaruje kroz dvojezgrenost procesora Zynq-7000 SoC-a, ali i kroz raspodjelu zadaća na sklopovske podsustave sadržane unutar procesorskog dijela ili implementirane u programabilnu logiku. Višejezgrenost (engl. *Multi-Core*) određuje operacijski tip sustava tj. način distribucije operacijskog sustava koji provodi koncept višezadaćnosti dodjeljivanjem zadataka nekog paralelnog procesa odgovarajućim sklopovskim podsustavima. Upravo su sklopovski menadžment i jednostavnija korisnička sučelja za pristup uređajima glavne prednosti operacijskih sustava. Tako korisničke aplikacije pod operacijskim sustavima mogu pristupati svim periferijama, a pojednostavljena je skalabilnost (engl. *Scalability*) tj. prilagodba novim funkcionalnostima. Aplikacije su izvođene na zajedničkom aplikacijskom sučelju koje osigurava izvođenje zadataka sa skrivenom sklopovskom (programabilna logika) ili programskom (ARM procesor) implementacijom niske razine [1]. Osim toga operacijski sustav efikasno upravlja potrošnjom koristeći skaliranje frekvencije i mogućnost ulaženja u optimizacijski *idle* mod.

Procesorski dvojezgreni ARM sustav kod Zynq-7000 sustava na čipu omogućuje dvije vrste višeprocesnih operacijskih tipova:

- **Simetrični mod - SMP** (engl. *Symmetric Multi-Processing*)
- **Asimetrični mod - AMP** (engl. *Asymmetric Multi-Processing*)

Kod simetričnog operacijskog tipa obje ARM jezgre pokreću isti operacijski sustav i zajednički sudjeluju u izvođenju operacija tj. koristi se raspoređivač procesa po jezgrama. To je inicijalna konfiguracija većine operacijskih *kernela* za Zynq-7000 SoC. Kod asimetričnog tipa jezgre operiraju nezavisno tj. svaka sa svojim vlastitim operacijskim sustavom, a čest je slučaj da jedna jezgra pokreće ozbiljniji operacijski sustav kao što je Linux, dok druga pokreće neki jednostavniji operacijski sustav kao što je FreeRTOS. Moguće je da jedna ili čak obje jezgre aplikacije izvršavaju kao samostojeće (engl. *bare-metal*). U ovom modu jezgre komuniciraju preko zajedničkih resursa kao što su DDR memorija ili kontroler prekida [7].

3.2 Linux otvoreni operacijski sustav

Operacijski sustav Linux čine jezgra tj. *kernel* slična jezgri starijeg višezadaćnog operacijskog sustava UNIX (engl. *Uniplexed Information and Computing System*), pogonski programi (engl. *driver*), GNU sistemski i aplikacijski programi koji čine standardnu C biblioteku te ostale korisničke biblioteke kao primjerice *widget* biblioteke za grafička sučelja - GUI (engl. *Graphical User Interface*). Razne distribucije Linuxa se koriste na osobnim računalima, serverima i ugradbenim sustavima.

Potpuni Linux operacijski sustav na nekom računalnom sustavu možemo strukturalno podijeliti na 5 osnovnih komponenata kao što je prikazano slikom 5 - sklopovlje, univerzalni programski podizač (engl. *Bootloader*), *kernel* operacijskog sustava, usluge operacijskog sustava (engl. *OS Services*) i korisničke aplikacije (engl. *User Applications*). *Bootloader* u RAM memoriju učitava *kernel* iz stalne memorije kod uključivanja računala. To je mali program koji detektira sklopovlje i podiže sustav. *Kernel* je baza svakog operacijskog sustava. On vrši kontrolu procesa i povezivanja te pristup periferijama i načinu organizacije podataka (engl. *File System*). *Kernel* povezuje korisničke programe sa sklopovljem rukovodeći ulazno-izlaznim programskim zahtjevima prevodeći instrukcije za CPU jezgru. Usluge operacijskog sustava opslužuju programe korisničkog sučelja kao što je sustavni poziv (engl. *system call*) koji zahtijeva uslugu od *kernela*. Zahtjevi se svode na pokretanje procesa, pokretanje *drivera*, pristup memoriji i slično.

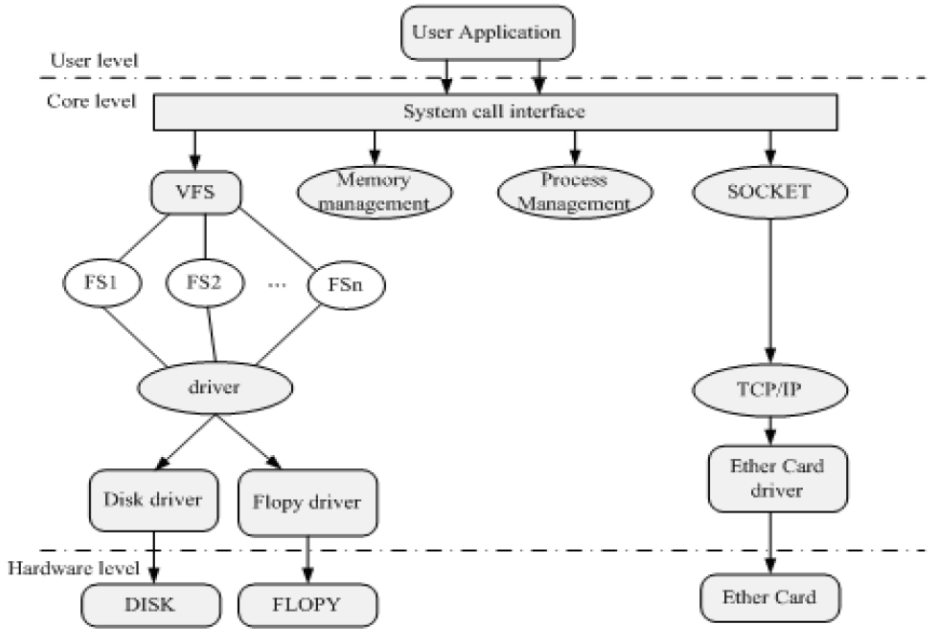
Korisničke aplikacije zapravo su sveobuhvatni naziv za programe koji se izvršavaju na vrhu jezgre operacijskog sustava. Tu spada izvršavanje funkcija standardne C biblioteke (čak i izvod biblioteke za Android) i implementacija korisničkog sučelja - komandne CLI (engl. *Command Line Interface*) ljuske ili grafičkog GUI-a sve u postupku izvođenja neke korisničke aplikacije u širem smislu kao što je internetska (npr. Mozilla Firefox), uredska (npr. LibreOffice) ili multimedijska aplikacija (npr. VLC Media Player, Blender). *Bootloader*, *kernel* i usluge operacijskog sustava čine operacijski sustav u užem smislu [2].

Slika 5: *Komponente Linux operacijskog sustava [2]*

Uz sučelje za sustavne pozive i *drivere* za uređaje koji mogu biti unutar njega integrirani, *kernel* operacijskog sustava sačinjava pet osnovnih komponenti tj. modula prikazanih u sloju jezgre na slici 6:

- **Modul rasporeda procesa** (engl. *Process Scheduling/Management*) - određuje sljedeću zadaću koja se dodaje u sustav i sljedeći proces koji se izvodi. Koristeći tehnike raspoređivanja modul dugoročno, srednjoročno ili kratkoročno određuje korištenje CPU resursa.
- **Modul upravljanja memorijom** (engl. *Memory Management*) - upravlja računalnom memorijom tako da dinamički alocira dijelove memorije na zahtjeve programa osiguravajući pritom sigurnost svih procesa koji dijele istu memoriju.
- **Modul virtualnog organiziranja podataka - VFS** (engl. *Virtual File System*) - apstraktni sloj organizacije koji se nalazi na stvarnom *file systemu.* Taj modul zapravo podržava dodavanje vanjskih *drivera* i pristupanje lokalnim, ali i mrežnim uređajima za pohranu podataka. Koristi se za uklanjanje razlika u organizaciji podataka između različitih operacijskih sustava.

- **Modul za komunikaciju između procesa - IPC** (engl. *Inter-Process Communication*) - sadrži protokole razmjene informacija između procesa na temelju odnosa klijent - server.
- **Modul mrežnog sučelja** (engl. *Network Interface*) - za određeni komunikacijski standard osigurava sklopovsku podršku [2].



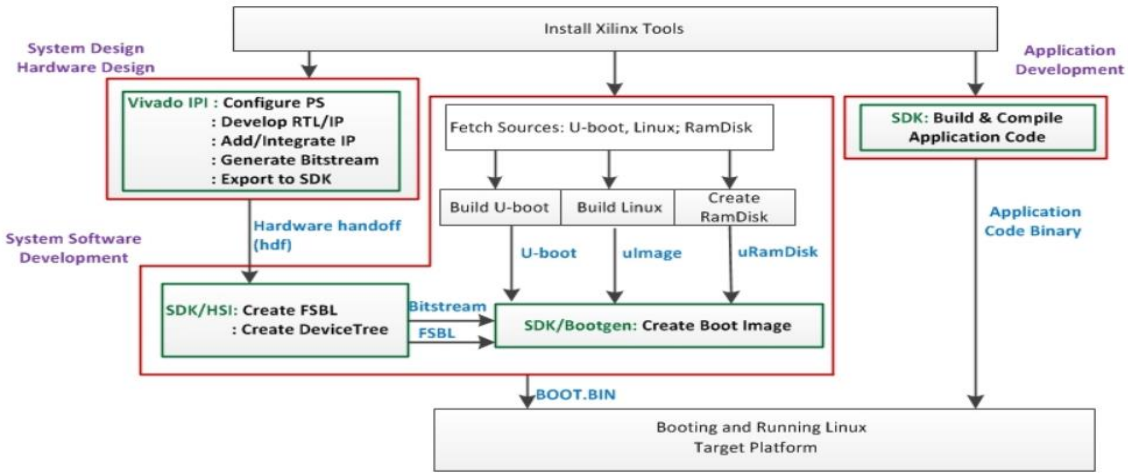
Slika 6: *Kernel Linux operacijskog sustava [2]*

3.3 Linux distribucija na Zynq-7000 SoC-u

Xilinx za svoje silicijske sustave osigurava uz PetaLinux kao potpunu komercijalnu ugradbenu inačicu Linux distribucije i projekt otvorenog Linuxa - OSL (engl. *Open Source Linux*). On sadrži programsku podršku Linux OS-a u GIT repozitoriju te *wiki* stranicu sa opsežnom razvojnom dokumentacijom.

Podizanje otvorenog operacijskog sustava kao što je Linux je zapravo transplantacija koda iz zajedničkog repozitorija na ciljanu arhitekturu nakon čega slijedi dodavanje *drivera* za uređaje tj. za periferije i konačno stvaranje aplikacije C programskim jezikom. U svom GIT repozitoriju Xilinx za Zynq-7000 sustav na čipu daje unaprijed izgrađen Linux *kernel*, *drivere* za periferije, univerzalni *bootloader* (engl. *U-Boot*), generator stabla uređaja (engl. *Device Tree Generator*) pa čak QEMU emulator i vizualizator mašine.

Tok razvoja Linux aplikacije prikazan je slikom 7. On je podijeljen na tri osnovna bloka: dizajn sklopovlja, razvoj programske podrške te razvoj aplikacije. Nakon instalacije potrebnih Xilinx razvojnih alata kreće se sa razvojem ili prilagodbom sklopovlja. S Vivado ili ISE - WebPack razvojnim alatom se konfigurira procesorski dio i razvijaju se IP (engl. *Intellectual Property*) jezgre u programabilnoj logici. Zatim se IP jezgre dodaju u sustav i povezuju, a konačnom se sustavu generira sekvenca bitova (engl. *Bitstream*) kao osnovni element konfiguracije sustava na čipu. Ta se sekvenca prosljeđuje SDK alatu koji od nje stvara prvi sustavni podizač - FSBL. FSBL konfigurira polje programabilne logike sa *bitstreamom* te počinje s podizanjem operacijskog sustava. On se učitava sa *on-chip* memorije, ali može i sa *Flash* memorije (opcijom XIP_mode). Od skinutog i kompiliranog U-Boot podizača, stvorenog FSBL-a te sistemskog *bitstreama* SDK stvara BOOT sliku. Skidanjem i kompilacijom Linux *kernela* stvara se uImage ili zImage slika, a kompilacijom izvorišne datoteke stabla uređaja tj. DTS datoteke (engl. *Device Tree Source*), također dostupne u otvorenom repozitoriju, stvara se *blob* datoteka stabla uređaja tj. DTB datoteka (engl. *Device Tree Blob*). U-Boot, zImage, DTB datoteka i RamDisk *file system* (također otvoreno dostupan) čine distribuciju Linuxa na Zynq-7000 SoC-u koju je moguće podignuti preko SD kartice. Nakon toga se može pristupiti razvoju *drivera* i aplikacija za koje se piše C-kod.

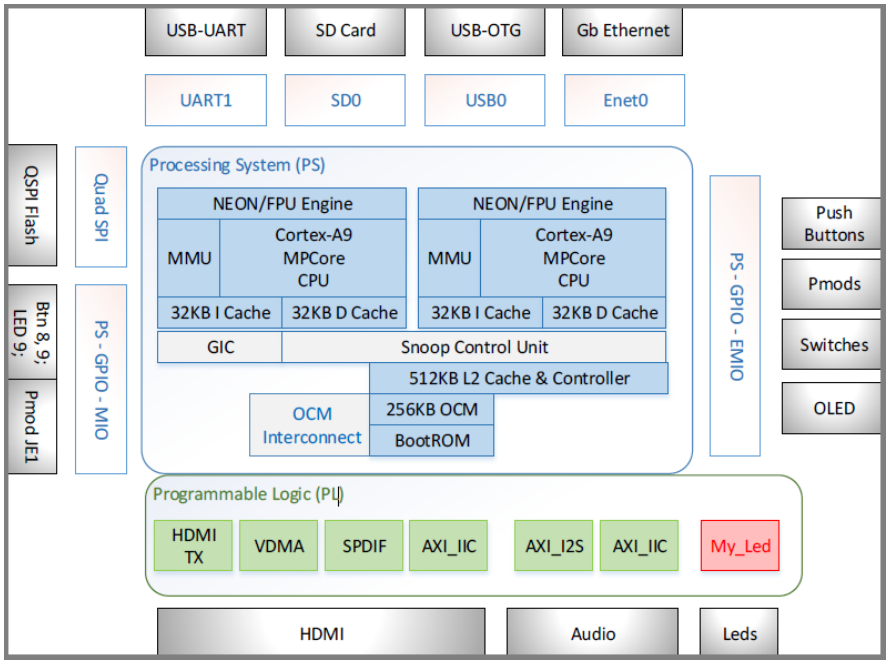


Slika 7: *Tok podizanja Linux distribucije i razvoja aplikacije na Zynq-7000 SoC-u [15]*

Razvoj aplikacije vrši se na visokoj razini apstrakcije, ali u podizanju Linux distribucije potrebno je toj aplikaciji stvoriti programsku podršku koja implicira sklopovsku prilagodbu aplikaciji. Upravo zato su moderni sustavi na čipu koji integriraju programabilnu logiku pogodni adaptaciji zahtijevanoj od strane aplikacije tj. korisnika.

4. LED Linux demonstracijska aplikacija

Proces i faze razvoja aplikacije koja se vrši pod Linux operacijskim sustavom prikazat ćemo na primjeru jednostavne manipulacije LED diodama. Kao što je istaknuto u prethodnom poglavlju taj razvoj implicira prilagodbu sklopovlja, razvoj programske podrške te samo pisanje aplikacije. Referentna arhitektura sustava ZedBoard razvojne pločice (slika 8) dostupna je na *web* stranici tvrtke Digilent u zajedničkom repozitoriju, a specifična je po tome što u polju programabilne logike već ima implementirane IP jezgre tj. module za operiranje i komunikaciju sa HDMI transmiterom i *audio codecom*. U polje programabilne logike dodat ćemo zasebnu IP jezgru za kontrolu LED diodama koju ćemo nazvati ***myled*** (crveno označeni modul na slici 8). U *constraint* datoteci izlaze te IP jezgre ćemo spojiti na fizičke priključke LED dioda te generirati *bitstream*. Slijedi kreiranje BOOT slike i kompilacija Linux *kernela* te pisanje i dodavanje našeg LED kontrolera u stablo uređaja. Na kraju se piše i kompilira aplikacija, a datoteke svih navedenih postupaka se stavljaju na SD karticu i tvore ugradbenu Linux distribuciju.



Slika 8: *Dodavanje LED kontrolera u polje programabilne logike[8]*

4.1 Preduvjeti

Za razvoj Linux ugradbene aplikacije se od strane računala domaćina koristi **Ubuntu 12.04** operacijski sustav te **ISE Design Suite 14.7** (WebPack) razvojni alat sa

XPS paketom te SDK programskim alatom. Prije početka razvoja Linux aplikacije potrebno je izvršiti određene predradnje tj. osigurati sljedeće:

- **Micro SD karticu** s najmanje 4 GB prostora na kojem bi se stvorile odgovarajuće particije za podizanje Linuxa. Korištenjem Disk Utility alata ili Linux komandnim sučeljem kao što je opisano u [9] stvorit ćemo dvije particije. Prva je veličine 1 GB i služi za podizanje Linuxa s komandnim sučeljem uz RamDisk (BusyBox) *file system* koji ne vrši pohranu promjena nakon gašenja sustava. Prva particija mora biti FAT (engl. *File Allocation Table*) formatirana i nazivamo ju **FAT_boot**. Druga particija je veličine 3 GB i opcionalno se može koristiti za *file system* Linux inačice sa grafičkim sučeljem kao što je to Linaro Ubuntu koja omogućuje pohranu promjena u memoriju. Ova particija mora biti EXT4 (engl. *Fourth Extendend File System*) formatirana i nazivamo ju **EXT4_FS**. Za našu LED Linux aplikaciju koristit ćemo samo prvu particiju.
- U radni direktorij skinuti i raspakirati repozitorij **Linux Hardware Design** sa:

  http://www.digilentinc.com/Data/Products/ZEDBOARD/ZedBoard_OOB_Design.zip
- Na računalu instalirati **U-Boot alate** za generiranje univerzalnog *bootloadera*:

  sudo apt-get install u-boot-tools
- Iz GIT repozitorija u radni direktorij skinuti **U-Boot** podizač i **Linux *kernel* v3.6-digilent-13.01** za Digilent uređaje:

  git clone https://github.com/Digilent/u-boot-digilent

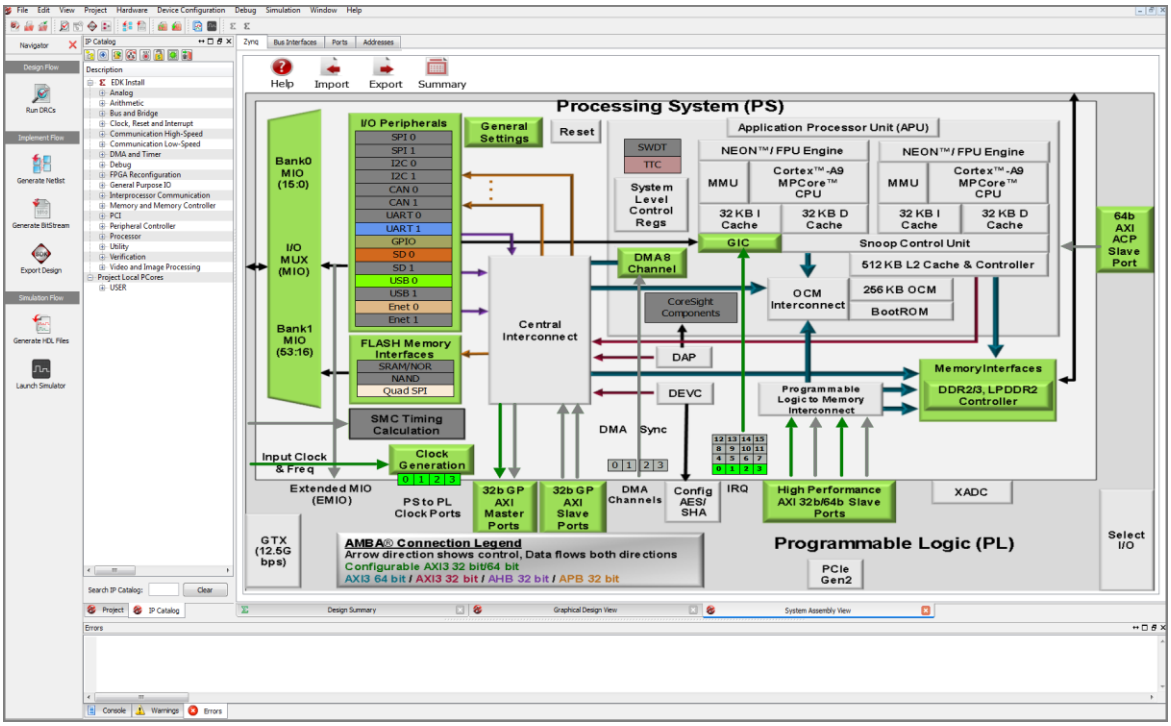
  git clone https://github.com/Digilent/linux-digilent

Naš radni direktorij nazovimo **ZedBoard_Diplomski**. On zasad sadrži tri repozitorija: **ZedBoard_Linux_Design**, **u-boot-digilent** i **linux-digilent**.

4.2 Sklopovska prilagodba

Dodavanje LED kontrolera u polje programabilne logike vršimo u XPS paketu ISE alata. U skinutom repozitoriju pod *ZedBoard_Linux_Design/hw/xps_proj* otvaramo referentni ZedBoard dizajn **system.xmp**. U XPS grafičkom okruženju (slika 9) otvara nam

se blok dijagram našeg sustava sa prikazanom Zynq arhitekturom. Blokovi su prilagodljivi potrebama aplikacije, a obzirom da ćemo upravljanje LED diodama obavljati kroz GPIO blok priključaka opće namjene otvaramo blok **I/O Peripherals.** U otvorenom *Zynq PS MIO* prozoru za konfiguraciju priključaka proširimo GPIO izbornik te smanjimo širinu EMIO pinova sa 60 na **52** kako bismo uklonili referentni spoj sa LED diodama. EMIO su oni priključci koji mogu povezati procesorski dio sa programabilnom logikom ili pinovima kućišta. U ovom prozoru možemo vidjeti i spoj svih periferija na multipleksirane MIO priključke kojih u Zynq arhitekturi ima 54. Pritiskom na *Show MIO Table* dobivamo mapu MIO priključaka sa opisom svakog priključka ovisno o periferiji kojoj je dodijeljen.

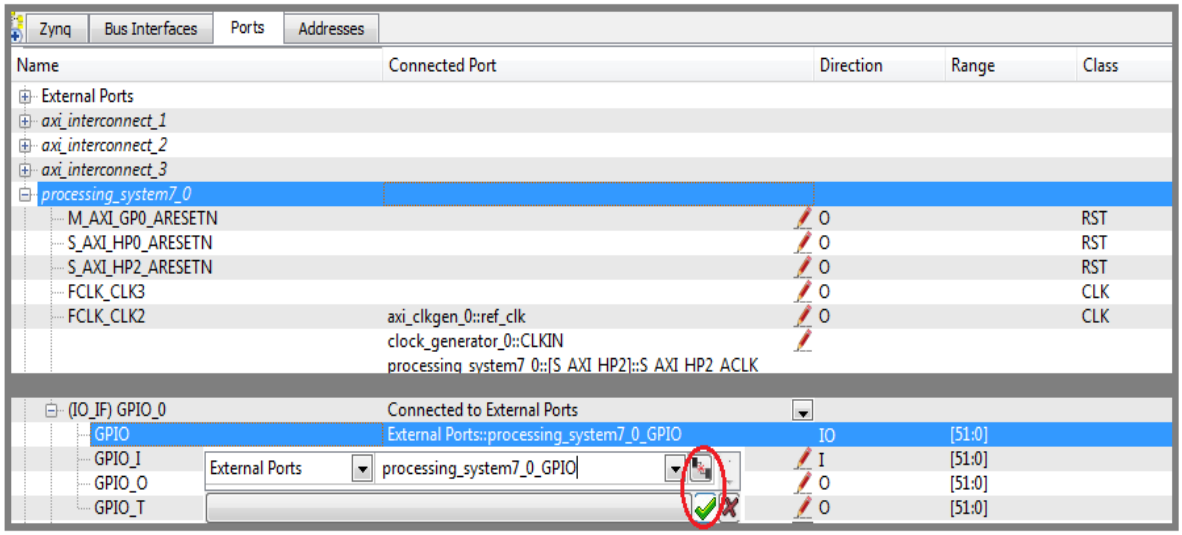


Slika 9: *Referentni ZedBoard dizajn u XPS razvojnom okruženju*

Zbog izmjena priključaka potrebno je osvježiti vanjske portove pa u glavnom prozoru pod sučeljem **Ports** odabiremo izbornik **processing_system7_0** koji sadrži sve portove procesorskog sustava PS. Odabiremo podizbornik **(IO_IF) GPIO_0** u kojem se nalazi **GPIO** port sa 52 priključka kojeg moramo osvježiti. Odabirom tog porta otvara se prozor prema slici 10. Gornjom ikonicom u crvenom krugu brišemo vezu, a nakon toga donjom ikonicom dodajemo GPIO port kao vanjski port te ga nazivamo **processing_system7_0_GPIO** zbog kasnije strukture *constraint* datoteke.

Nakon definicije vanjskog porta za GPIO periferiju, možemo kreirati novu IP jezgru tj. LED kontroler u polju programabilne logike. U glavnom prozoru odabiremo

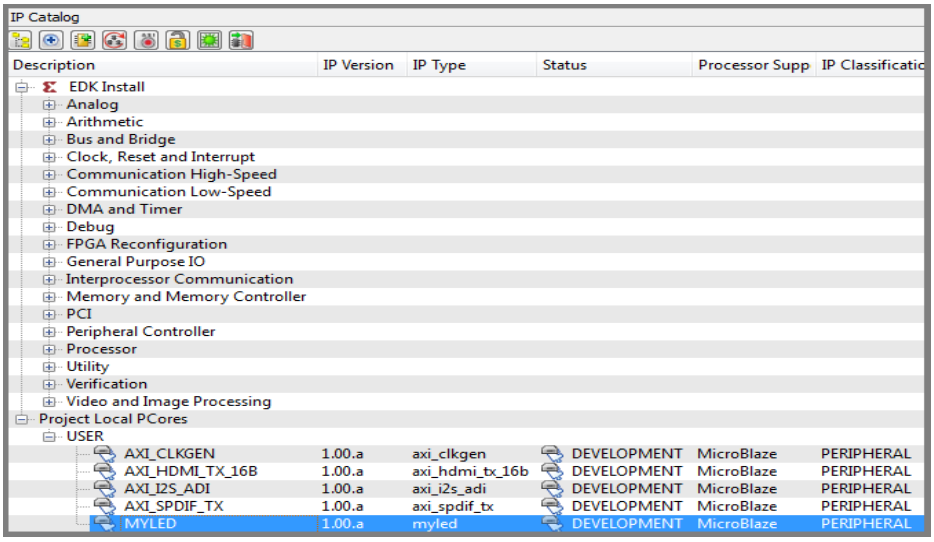
izbor **Hardware→Create or Import Peripheral** nakon čega se pojavljuje vodič za kreiranje ili dodavanje periferije. *Next* odabirom dolazimo na izbornik gdje biramo kreiranje nove periferije tj. **Create templates for a new peripheral**. Sljedećim odabirom, nakon odabira direktorija, otvara nam se *Create Peripheral* prozor. Za ime IP jezgre stavljamo ***myled*** te opcionalno kratki opis. Nadalje odabiremo sabirničko sučelje **AXI4-Lite** za komunikaciju našeg IP kontrolera sa AXI prospojem. *Nextom* se otvara usluga IP sučelja gdje odabiremo samo **User logic software register** kao osnovni element *slave* konfiguracije (za kontrolu, status itd.). U idućem prozoru *User S/W Register* konfiguriramo programske registre koje implementiramo u polje programabilne logike. Za broj 32-bitnih registara ovdje odabiremo **1**. Donji bajt tog registra će kontrolirati LED-ice. U sljedećim prozorima *IP Interconnect, Peripheral Simulation Support* i *Peripheral Implementation Support* ostavljamo zadane postavke. Odabirom *Finish* se VHDL jezikom opisuje IP sučelje **myled.vhd** definirano prethodnim koracima zajedno sa **user_logic.vhd** predloškom za korisnički opis kontrolera preko programskog registra. Također se generira **MPD** (engl. *Microprocessor Peripheral Description*) ugradbeni predložak kako bi se kreirana IP jezgra mogla uskladiti sa procesorskim sustavom.



Slika 10: *Spajanje GPIO periferije na vanjski port procesorskog sustava PS*

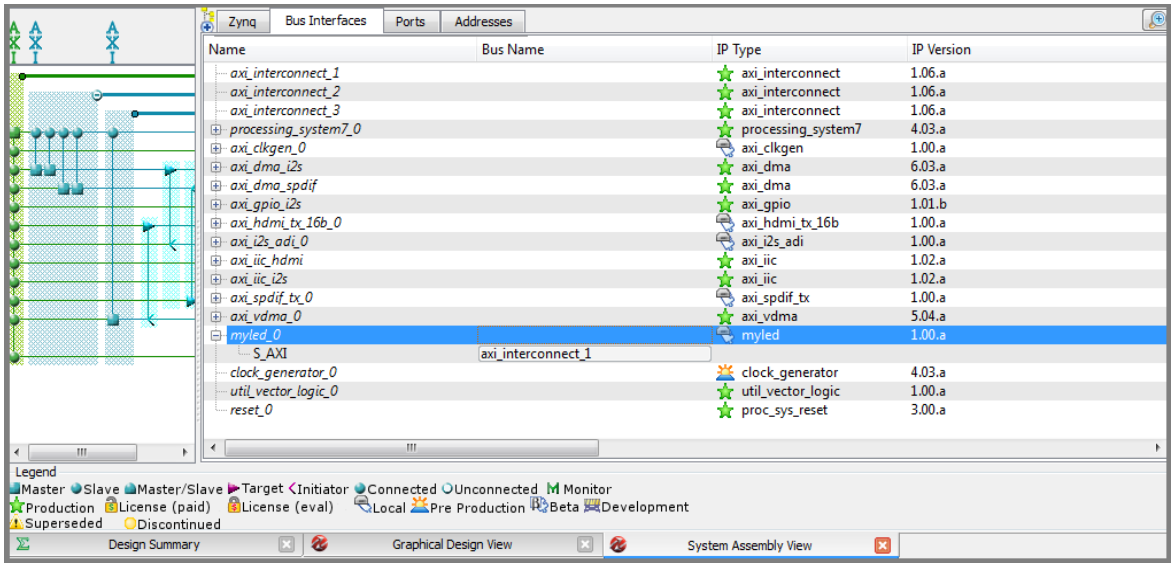
U lokalnom direktoriju se generirala ***myled*** IP jezgra opisana prethodnim koracima koju moramo dodati u naš referentni dizajn. U glavnom prozoru XPS alata pod sučeljem **IP Catalog** pronalazimo **MYLED** jezgru kao što je prikazano slikom 11. Desnim klikom na jezgru odabiremo **Add IP** čime nam se otvara konfiguracijski prozor *XPS Core Config* sa blok prikazom LED kontrolera koji je spojen preko S_AXI porta na AXI sučelje. U

ovom prozoru ostavljamo zadane postavke i odabiremo **OK**. Otvara se prozor *Instantiate and Connect IP* gdje odabiremo **processing_system7_0** kao procesor kojeg povezujemo s našom IP jezgrom preko prije definiranog sabirničkog sučelja i vanjskog porta. Dodavanje IP jezgre završavamo odabirom **OK**.



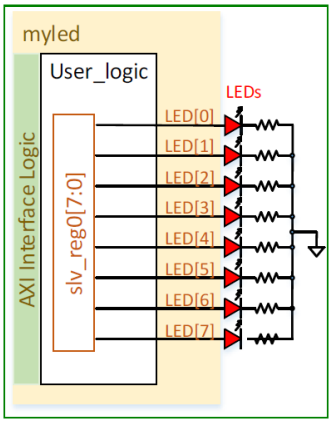
Slika 11: *Dodavanje LED kontrolera u dizajn*

Time se u sučeljima **Bus Interfaces** i **Ports** pojavljuje **myled_0** modul. U *Bus Interfaces* sučelju (slika 12) uz *myled_0* vidimo i ostale module referentnog dizajna i način njihova spajanja na AXI sabirnički prospoj. Tako se primjerice može uočiti da je u prvom AXI sučelju nazvanom *axi_interconnect_1* (zeleno označena AXI matrica u lijevom prozoru) *processing_system7* procesni sustav zapravo *master*, a svi ostali moduli su *slave* pa među njima i naš LED kontroler. Na to sučelje je *myled_0* modul spojen preko S_AXI porta.



Slika 12: *Sabirničko sučelje i spojeni IP moduli*

Nakon spajanja LED kontrolera na sabirnički prospoj preostaje nam ga u polju programabilne logike opisati prema okolini tj. prema LED diodama na ZedBoardu. Ranijim kreiranjem IP jezgre LED kontroler je strukturiran kao VHDL *top-level* modul ***myled*** koji integrira AXI sabirničko sučelje i korisnički modul *user_logic* sa kontrolnim registrom širine 8 (slika 13). Potrebno je u VHDL kodu *user_logic.vhd* korisničkog modula provesti kontrolni registar *slv_reg0* do *myled* modula, a u *myled.vhd* kodu provesti te iste ulaze na LED izlazni port *myled* modula.



Slika 13: *Blok shema myled kontrolera u programabilnoj logici [8]*

Desnim klikom na *myled_0* modul sa slike 12 i odabirom **Browse HDL Sources** otvaraju nam se VHD datoteke *myled.vhd* i *user_logic.vhd*. **User_logic.vhd** kod izmjenjujemo dodavanjem sivo označenih linija prema slici 14, a **myled.vhd** prema slici 15. Također je procesoru potrebno definirati LED port pa desnim klikom na *myled_0* sa slike 12 i odabirom **View MPD→myled.mpd** definiramo LED port prema slici 16.

```
84  entity user_logic is
85    generic
86    (
87      -- ADD USER GENERICS BELOW THIS LINE ---------------
88      --USER generics added here
89      -- ADD USER GENERICS ABOVE THIS LINE ---------------
90
91      -- DO NOT EDIT BELOW THIS LINE ----------------------
92      -- Bus protocol parameters, do not add to or delete
93      C_NUM_REG                      : integer              := 1;
94      C_SLV_DWIDTH                   : integer              := 32
95      -- DO NOT EDIT ABOVE THIS LINE ----------------------
96    );
97    port
98    (
99      -- ADD USER PORTS BELOW THIS LINE ------------------
100     LED   : out std_logic_vector(7 downto 0);
101     -- ADD USER PORTS ABOVE THIS LINE ------------------

146   --USER logic implementation added here
147   LED <= slv_reg0(7 downto 0);
148   ------------------------------------------
149   -- Example code to read/write user logic slave model s/w accessible registers
```

Slika 14: *User_logic.vhd*

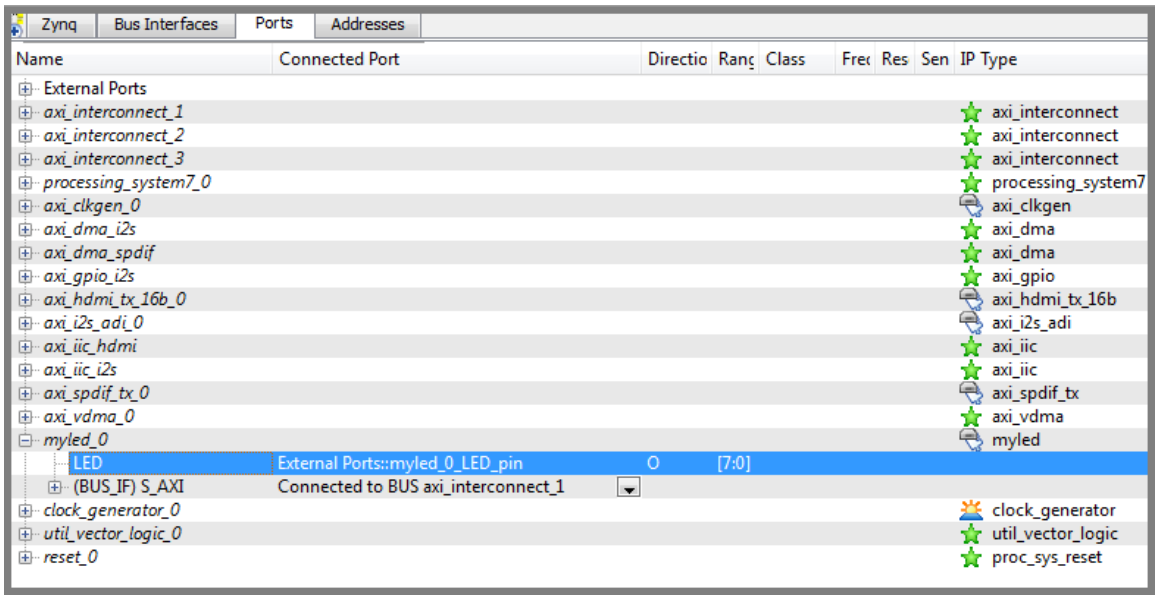
```
139     -- ADD USER PORTS BELOW THIS LINE ------------------
140     --USER ports added here
141     LED  : out std_logic_vector(7 downto 0);
142     -- ADD USER PORTS ABOVE THIS LINE ------------------
143
144     -- DO NOT EDIT BELOW THIS LINE ---------------------
145     -- Bus protocol ports, do not add to or delete
146     S_AXI_ACLK                      : in  std_logic;
147     S_AXI_ARESETN                   : in  std_logic;

302     port map
303     (
304       -- MAP USER PORTS BELOW THIS LINE ------------------
305       --USER ports mapped here
306       LED => LED,
307       -- MAP USER PORTS ABOVE THIS LINE ------------------
308
309       Bus2IP_Clk                    => ipif_Bus2IP_Clk,
310       Bus2IP_Resetn                 => ipif_Bus2IP_Resetn,
```

Slika 15: *Myled.vhd*

```
39  ## Ports
40  PORT LED = "", DIR = O, VEC =[7:0]
41  PORT S_AXI_ACLK = "", DIR = I, SIGIS = CLK, BUS = S_AXI
42  PORT S_AXI_ARESETN = ARESETN, DIR = I, SIGIS = RST, BUS = S_AXI
43  PORT S_AXI_AWADDR = AWADDR, DIR = I, VEC = [(C_S_AXI_ADDR_WIDTH-1):0], ENDIAN = LITTLE, BUS = S_AXI
```

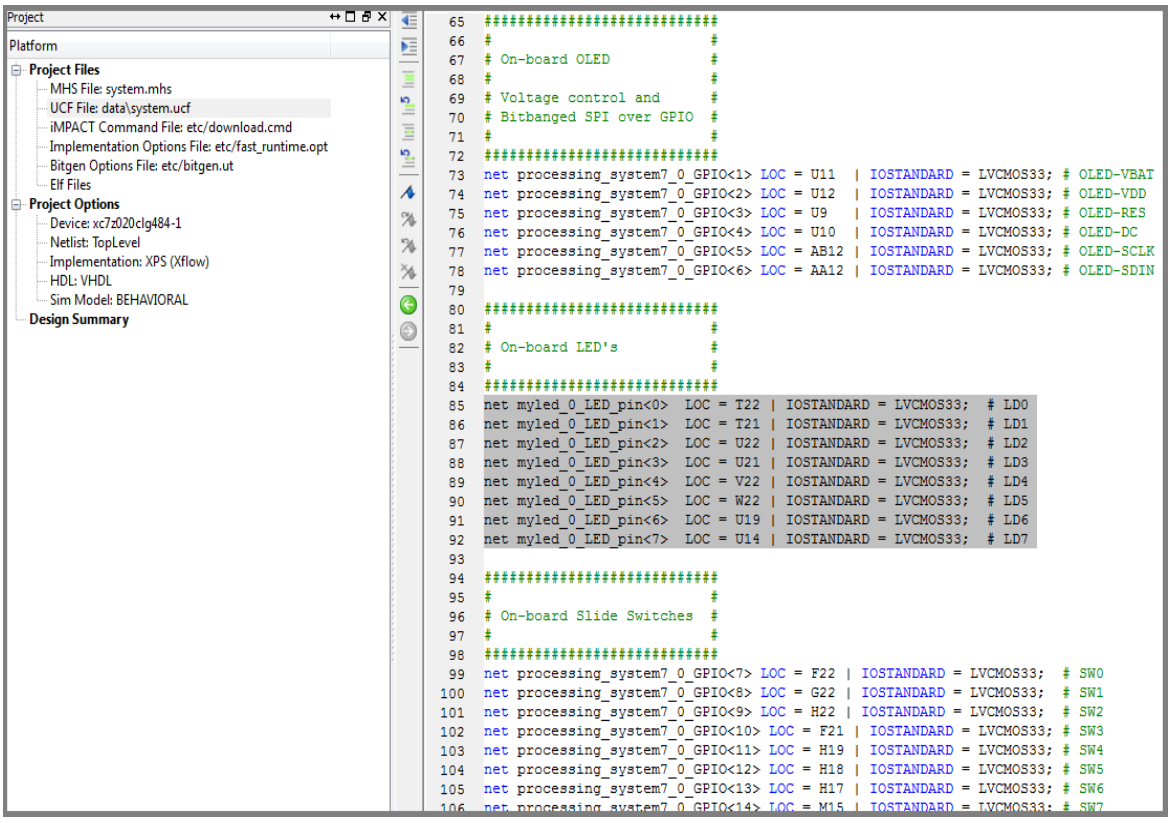
Slika 16: *Myled.mpd*

Obzirom da u **Ports** sučelju pod **myled_0** modulom nema pridruženog LED porta, odabiremo **Project→Rescan User Repositories.** Ponovnim pregledom *Ports* sučelja pod *myled_0* se pojavljuje LED port kojeg klikom spajamo na **External Ports** kao što je prikazano slikom 17. Ovih 8 priključaka vanjskog porta za povezivanje sa LED diodama su nazvani **myled_0_LED_pin** i pod tim ćemo ih nazivom u *constraint* datoteci spajati sa onim fizičkim priključcima Zynq kućišta koji su spojeni na LED diode ZedBoard pločice.



Slika 17: *Spajanje LED porta myled kontrolera na vanjski port*

Prije generiranja sekvence bitova za konfiguraciju tj. *bitstreama* preostaje nam samo prilagoditi referentnu *constraint* UCF datoteku koja portovima modula u programabilnoj logici i GPIO periferiji pridodjeljuje fizičke priključke Zynq kućišta spojene na određene komponente ZedBoard pločice. U glavnom XPS prozoru, u lijevom podprozoru odabiremo sučelje **Project** gdje pod **Project Files** otvaramo **data\system.ucf** datoteku. Priključke LED dioda spajamo na **myled_0_LED_pin** port kao što je prikazano slikom 18 dok ostale komponente spojene na **processing_system7_0_GPIO** slijedno modificiramo. Tako je *JD10* priključak Pmod JD konektora kao zadnji GPIO priključak spojen na *processing_system7_0_GPIO<51>* port.



Slika 18: *Prilagodba constraint UCF datoteke*

Bitstream za konfiguraciju našeg sklopovskog dizajna generiramo odabirom **Hardware→Generate Bitstream** u glavnom prozoru ili klikom na **Generate Bitstream** ikonu u lijevom stupcu glavnog prozora. Naredbom se obavljaju operacije sinteze i implementacije dizajna, a ukupno izvođenje generiranja traje oko 15 minuta. Završetkom tih operacija generiran je *bitstream* pod nazivom **system.bit**, a detaljno izvješće implementacije dizajna prikazano je slikom 19. Time je završena sklopovska prilagodba Zynq-7000 SoC-a, prva faza u razvoju Linux aplikacije [8].

Project Status (05/22/2015 - 13:31:09)			
Project File:	system.xmp	**Implementation State:**	Programming File Generated
Module Name:	system	**• Errors:**	No Errors
Product Version:	EDK 14.7	**• Warnings:**	272 Warnings (83 new)

XPS Reports [+]

XPS Synthesis Summary (estimated values) [+]

Device Utilization Summary (actual values) [+]

Performance Summary [-]			
Final Timing Score:	0 (Setup: 0, Hold: 0, Component Switching Limit: 0)	**Pinout Data:**	Pinout Report
Routing Results:	All Signals Completely Routed	**Clock Data:**	Clock Report
Timing Constraints:	All Constraints Met		

Detailed Reports [-]					
Report Name	**Status**	**Generated**	**Errors**	**Warnings**	**Infos**
Translation Report	Current	čet 21. svi 12:12:53 2015	0	189 Warnings (0 new)	5 Infos (0 new)
Map Report	Current	čet 21. svi 12:17:52 2015	0	31 Warnings (31 new)	1410 Infos (1410 new)
Place and Route Report	Current	čet 21. svi 12:19:49 2015	0	27 Warnings (27 new)	2 Infos (2 new)
Post-PAR Static Timing Report	Current	čet 21. svi 12:20:20 2015	0	0	4 Infos (4 new)
Bitgen Report	Current	čet 21. svi 12:21:31 2015	0	25 Warnings (25 new)	1 Info (1 new)

Secondary Reports [+]

Date Generated: 05/22/2015 - 13:31:09

Slika 19: *Izvješće sinteze i implementacije dizajna*

4.3 Generiranje BOOT slike

Prije programskog razvoja moramo kompilirati U-Boot podizač u **ELF** (engl. *Executable and Linkable File*) datoteku sukladno ZedBoard pločici čija je konfiguracija zapisana u **zynq_zed.h** zaglavlju koje se nalazi pod *u-boot-digilent/include/configs*. Konfiguraciju moramo prilagoditi tako da ZedBoard pločica radi u lokalnoj mreži. Mijenjamo IP adresu, IP server i Ethernet adresu prema označenim linijama na slici 20.

zynq_zed.h

```
#ifndef __CONFIG_ZYNQ_ZED_H
#define __CONFIG_ZYNQ_ZED_H

#include <configs/zynq_common.h>

/*
 * High Level Configuration Options
 */
#define CONFIG_ZED /* Community Board */

/* Default environment */
#define CONFIG_IPADDR   192.168.10.250
#define CONFIG_SERVERIP 192.168.10.1

#undef CONFIG_ZYNQ_XIL_LQSPI

/* No NOR Flash available on ZedBoard */
#define CONFIG_SYS_NO_FLASH
#define CONFIG_ENV_IS_NOWHERE

#undef CONFIG_EXTRA_ENV_SETTINGS
#define CONFIG_EXTRA_ENV_SETTINGS       \
        "ethaddr=00:0a:35:00:01:45\0"   \
        "kernel_size=0x140000\0"        \
        "ramdisk_size=0x200000\0"       \
        "qspiboot=sf probe 0 0 0;" \
```

Slika 20: *Zynq_zed.h zaglavlje*

Kompilaciju vršimo pomoću **cross-compile** alata za ARM jezgru koji dolaze uz ISE i Vivado razvojne alate. Stoga moramo prije kompilacije postaviti postavke ISE ili Vivado

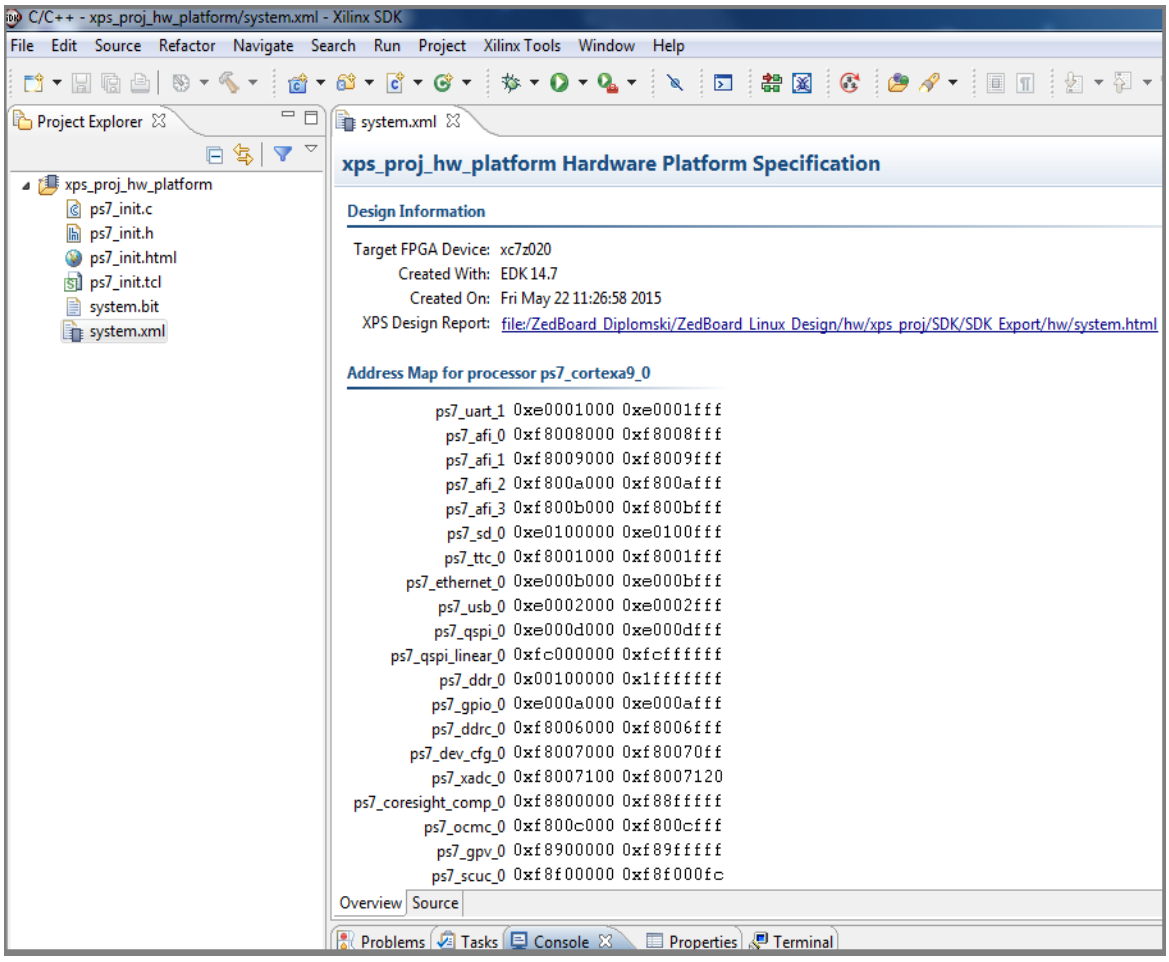
alata kao izvorišne uz pomoć naredbe **source** u Linux terminalu. U *u-boot-digilent* direktoriju kompiliramo sljedećim terminalnim naredbama:

[petar@petar u-boot-digilent]$ make CROSS_COMPILE=arm-xilinx-linux-gnueabi- zynq_zed _config

[petar@petar u-boot-digilent]$ make CROSS_COMPILE=arm-xilinx-linux-gnueabi-

Dobivenoj *u-boot* datoteci dodajemo .ELF ekstenziju i kopiramo ju u *boot-image* direktorij kako bi bila lakše dobavljiva SDK alatu:

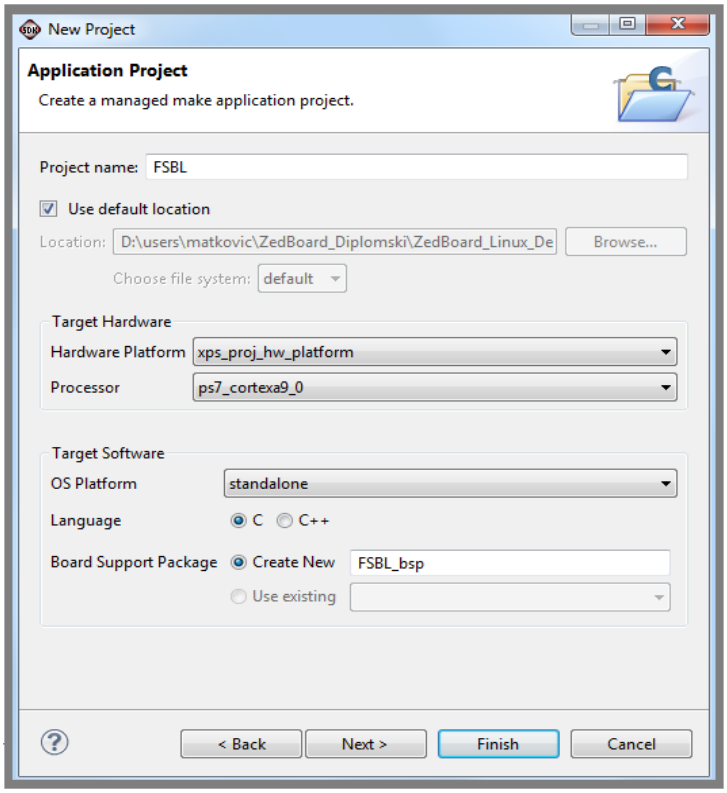
[petar@petar u-boot-digilent]$ cp u-boot ../ZedBoard_Linux_Design/boot-image/u-boot.elf



Slika 21: *SDK razvojno okruženje sa sklopovskim specifikacijama*

Sklopovski dizajn u XPS okruženju koji uključuje generirani *bitstream* prosljeđuje se SDK alatu u kojemu prilagođavamo programsku podršku tj. stvaramo prvi sustavni podizač - FSBL te generiramo BOOT sliku. U glavnom prozoru XPS alata pod izbornikom **Project** odabiremo **Export Hardware Design to SDK**. U otvorenom prozoru uključujemo

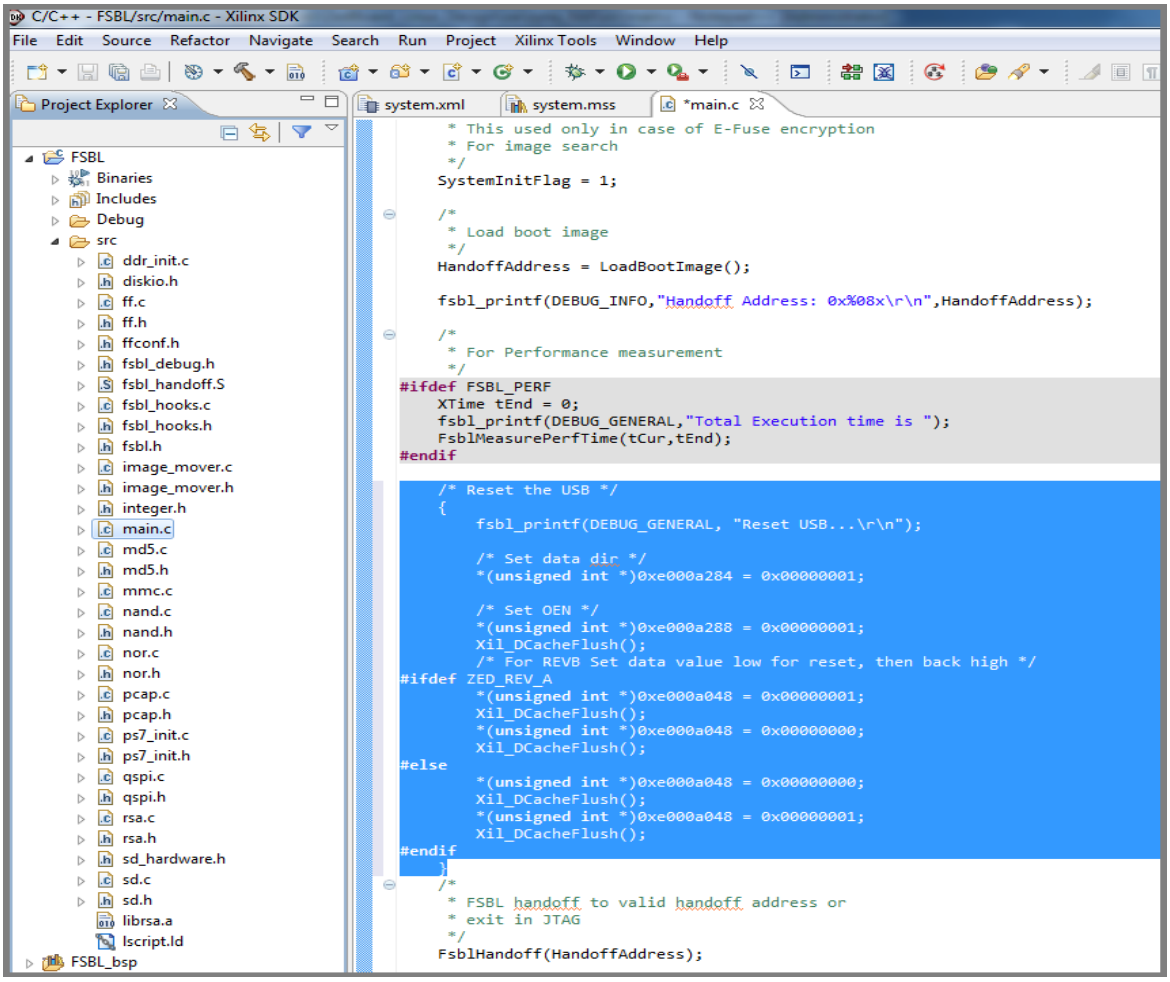
bitstream odabirom **Include bitstream and BMM file** te pokrećemo SDK sa **Export & Lunch SDK**. Kao radni direktorij se sugerira *ZedBoard_Linux_Design/hw/xps_proj/SDK/SDK_Export* kojeg i prihvaćamo odabirom **OK**. Otvara se SDK okruženje sa otvorenom sklopovskom platformom prema slici 21 koja uključuje inicijalizacijski program za naš procesorski sustav, *bitstream* za programabilnu logiku te razne specifikacije kao primjerice cijelu adresnu mapu. Kako bismo stvorili FSBL u glavnom SDK prozoru odabiremo **File→New Project** te u novootvorenom prozoru označimo **Xilinx→Application Project** te odaberemo **Next**. Otvara se novi prozor za postavke. Prema slici 22 odabiremo naziv, sklopovsku platformu te platformu operacijskog sustava (**standalone**). Nakon **Next** odabira otvara se prozor za odabir predloška gdje odabiremo **Zynq FSBL**, a zatim **Finish**. Stvoren je novi projekt naziva *FSBL* sa svim potrebnim izvorišnim kodovima u *src* poddirektoriju.



Slika 22: *Postavke za FSBL podizač*

U *main.c* programu *src* poddirektorija potrebno je dodati dio koda za resetiranje USB sklopa. Taj dio koda je označen plavo na slici 23. Taj se odsječak koda može pronaći unutar *main.c* programa u *ZedBoard_Linux_Design/sw/zynq_fsbl/src* direktoriju te prekopirati u *main.c* program našeg FSBL projekta na slikom 23 prikazano mjesto. Izmjene se moraju pohraniti, a projekt iznova podignuti. Odabiremo stoga **Project→Clean**

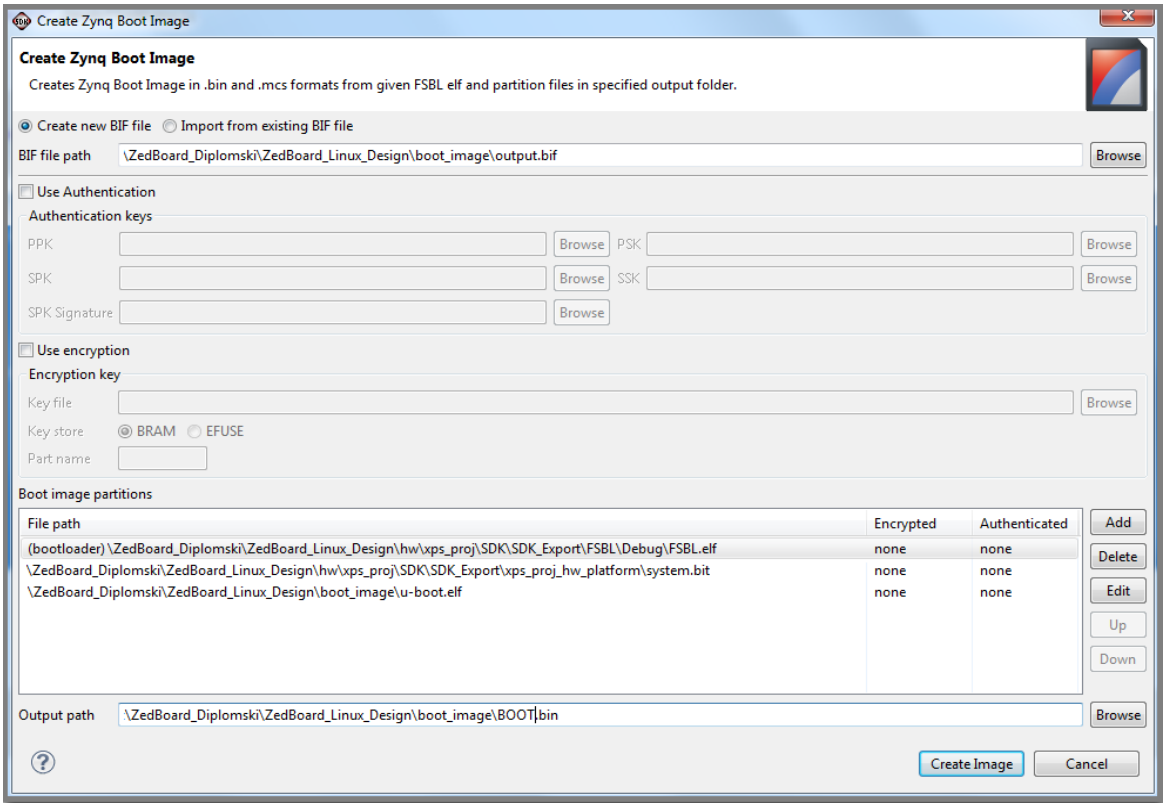
kako bismo projekt uklonili, a zatim **Project→Build all** kako bismo ga ponovno aktualizirali. Time se stvara **FSBL.elf** datoteka pohranjena u *ZedBoard_Linux_Design/hw /xps_proj/SDK/SDK_Export/FSBL /Debug* direktoriju.



Slika 23: *Modifikacija main.c FSBL programa za ispravan rad USB-a*

Sada konačno imamo sve datoteke koje su potrebne kao particije za generiranje BOOT slike. U glavnom SDK prozoru odabiremo **Xilinx Tools→Create Zynq Boot Image** čime se otvara prozor za konfiguraciju BOOT slike prikazan slikom 24. Odabirom **Add** kao prvu particiju postavljamo **FSBL.elf** datoteku koja nužno mora biti klasificirana kao *bootloader*. Druga particija je **system.bit** datoteka koja je postavljena kao *data files*. Treća particija je ranije kreiran U-Boot podizač **u-boot.elf** koji je također postavljen kao *data files*. Podatkovni putevi i redoslijed odabira particija zajedno sa izlaznim putem ciljane BOOT.bin datoteke također su prikazani slikom 24. Kreiranje izvršavamo odabirom **Create**, a generiranoj BOOT.bin datoteci mijenjamo naziv u **BOOT.BIN** kako bi mogla biti učitana nakon uključivanja ZedBoard pločice. BOOT.BIN slika jedna je od datoteka

Linux distribucije koju stavljamo na SD karticu i ona čini izravnu vezu operacijskog sustava i ciljanog Zynq sklopovlja [8].



Slika 24: *Stvaranje BOOT slike*

4.4 Kompilacija Linux kernela

Izvorišni kod otvorene Linux inačice već je skinut sa GIT repozitorija u opći Linux repozitorij *linux-digilent* kao što je i navedeno u 4.1 poglavlju. Sada je potrebno inačicu konfigurirati sukladno ZedBoard pločici. Ulazimo u *linux-digilent* repozitorij i za konfiguraciju koristimo osnovnu **digilent_zed_defconfig** datoteku u *arch/arm/configs* poddirektoriju. Koristimo *make* naredbu:

```
[petar@petar linux-digilent]$ make ARCH=arm CROSS_COMPILE=arm-xilinx-linux-gnueabi- digilent_zed_defconfig
```

Prije nastavka potrebno je promijeniti postavke za jedan Pmod *driver*. Taj je *driver* gornjom konfiguracijom postavljen kao OLED ugrađeni *built-in driver*, a mi ćemo ga postaviti kao *kernel* modul. Kako bismo to učinili moramo pokrenuti konfiguracijski izbornik za kojeg su na Linux mašini potrebni **Ncurses** alati. Njih instaliramo naredbom:

```
sudo apt-get install ncurses -dev
```

Sada možemo otvoriti konfiguracijski izbornik:

```
[petar@petar linux-digilent]$ make ARCH=arm CROSS_COMPILE=arm-xilinx-linux-gnueabi- menuconfig
```

U izborniku odabiremo **Device Driver→PMOD Support→PmodOLED1**. *PmodOLED1* postavljamo kao modul pritiskom slova **M**. Pritiskom na **Exit** i pohranom konfiguracije izlazimo iz konfiguracijskog izbornika. Preostaje nam kompilacija Linux *kernela* prema gornjoj konfiguraciji:

```
[petar@petar linux-digilent]$ make ARCH=arm CROSS_COMPILE=arm-xilinx-linux-gnueabi-
```

Nakon desetak minuta izgrađen je *kernel* tj. **zImage** slika u poddirektoriju *linux-digilent/arch/arm/boot* koju također stavljamo na SD karticu. Ona predstavlja operacijski sustav u užem smislu [8].

4.5 RamDisk

Obzirom na **standalone** odabir FSBL podizača naš ugradbeni sustav nezavisan je od vanjskih resursa poput NFS (engl. *Network File System*) *file system* distribuirane organizacije te ne učitava organizaciju podataka sa postavljenog servera već koristi malu komprimiranu **RamDisk** sliku tj. *file system* koji se kod podizanja preslikava iz izvorišne memorije u RAM memoriju i omogućuje komandno sučelje. RamDisk slika se nalazi u *Linux Hardware Design* repozitoriju pod *sd_image*.

4.6 Kontrolni driver i stablo uređaja

LED kontroler je nakon sklopovskog opisa i dodavanja u FPGA polje potrebno dodati kao novi čvor u stablo uređaja kako bi se u Linux distribuciji pod imenom *myled* fizički označila kreirana IP jezgra. U radnom direktoriju stvaramo novi direktorij pod nazivom *drivers*. U njega kopiramo **digilent-zed.dts** datoteku stabla uređaja za ZedBoard iz *linux-digilent/arch/arm/boot/dts* poddirektorija te joj zbog jednostavnosti mijenjamo ime u **devicetree.dts**. Tu DTS datoteku je prije kompilacije u DTB *blob* datoteku potrebno izmijeniti tako da joj se doda *myled* čvor sa točno određenom fizičkom adresom čvora i kompatibilnošću (koja će se kasnije referirati u C kodu za *driver*), ali joj je također,

obzirom da koristimo RamDisk *file system*, potrebno definirati okruženje (engl. *Environment*) preko **bootargs** argumenata. Definirano okruženje i dodani *myled* čvor označeni su crveno na slici 25. Fizička adresa *myled* čvora odgovara adresi *myled* IP jezgre kreirane ranije u XPS alatu. Ponovnim pokretanjem dizajna u XPS-u odabiremo u glavnom prozoru izbornik **Addresses** u kojem pronalazimo adresu *myled* jezgre kao što je prikazano slikom 26. Tu očitavamo baznu adresu 0x7E400000 i veličinu prostora od 64K što odgovara rasponu 0x10000. Upravo su te veličine čvora *myled* i upisane u stablo uređaja.

devicetree.dts (~/Desktop/ZedBoard_Diplomski/drivers) - gedit

Open Save Undo

devicetree.dts

```
/dts-v1/;
/ {
        #address-cells = <1>;
        #size-cells = <1>;
        compatible = "xlnx,zynq-zed";
        model = "Xilinx Zynq ZED";

        aliases {
                ethernet0 = &ps7_ethernet_0;
                serial0 = &ps7_uart_1;
        };

        chosen {
                /*bootargs = "console=ttyPS0,115200 root=/dev/mmcblk0p2 rw earlyprintk rootfstype=ext4
rootwait devtmpfs.mount=1";*/
                bootargs = "console=ttyPS0,115200 root=/dev/ram rw initrd=0x800000,8M earlyprintk
rootwait devtmpfs.mount=1";
                linux,stdout-path = "/axi@0/serial@e0001000";
        };

                        spi-sclk-gpio = <&ps7_gpio_0 59 0>;
                        spi-sdin-gpio = <&ps7_gpio_0 60 0>;
                };
                myled {
                        compatible = "dglnt,myled-1.00.a";
                        reg = <0x7E400000 0x10000>;
                };
        };
};
```

Slika 25: *Devicetree.dts - stablo uređaja*

Zynq | Bus Interfaces | Ports | Addresses

Instance	Base Name	Base Address	High Address	Size	Bus Interface(s)	Bus Name	Lock
processing_system7_0's Addres...							
processing_system7_0	C_DDR_RAM_B...	0x00000000	0x1FFFFFFF	512M			☑
axi_dma_spdif	C_BASEADDR	0x40400000	0x4040FFFF	64K	S_AXI_LITE	axi_interconnec...	☐
axi_dma_i2s	C_BASEADDR	0x40420000	0x4042FFFF	64K	S_AXI_LITE	axi_interconnec...	☐
axi_gpio_i2s	C_BASEADDR	0x41200000	0x4120FFFF	64K	S_AXI	axi_interconnec...	☐
axi_iic_hdmi	C_BASEADDR	0x41600000	0x4160FFFF	64K	S_AXI	axi_interconnec...	☐
axi_iic_i2s	C_BASEADDR	0x41640000	0x4164FFFF	64K	S_AXI	axi_interconnec...	☐
axi_vdma_0	C_BASEADDR	0x43000000	0x4300FFFF	64K	S_AXI_LITE	axi_interconnec...	☐
axi_hdmi_tx_16b_0	C_BASEADDR	0x70E00000	0x70E0FFFF	64K	S_AXI	axi_interconnec...	☐
axi_spdif_tx_0	C_BASEADDR	0x75C00000	0x75C0FFFF	64K	S_AXI	axi_interconnec...	☐
axi_i2s_adi_0	C_BASEADDR	0x77600000	0x7760FFFF	64K	S_AXI	axi_interconnec...	☐
axi_clkgen_0	C_BASEADDR	0x79000000	0x7900FFFF	64K	S_AXI	axi_interconnec...	☐
myled_0	C_BASEADDR	0x7E400000	0x7E40FFFF	64K	S_AXI	axi_interconnec...	☐
processing_system7_0	C_UART1_BASE...	0xE0001000	0xE0001FFF	4K			☑
processing_system7_0	C_USB0_BASEA...	0xE0002000	0xE0002FFF	4K			☑
processing_system7_0	C_GPIO_BASEA...	0xE000A000	0xE000AFFF	4K			☑
processing_system7_0	C_ENET0_BASE...	0xE000B000	0xE000BFFF	4K			☑
processing_system7_0	C_SDIO0_BASE...	0xE0100000	0xE0100FFF	4K			☑
processing_system7_0	C_TTC0_BASEA...	0xF8001000	0xF8001FFF	4K			☑

Slika 26: *Fizička adresa i opseg myled IP jezgre*

Slijedi kompilacija **devicetree.dtb** datoteke iz devicetree.dts stabla uređaja. DTB *blob* datoteka se predaje *kernelu* od strane *bootloadera* te je nužna za Linux distribuciju. Ona zapravo zamjenjuje određeni fond specifičnih C izvorišnih datoteka. Nakon sljedeće naredbe generirana je devicetree.dtb datoteka u *drivers* direktoriju:

```
[petar@petar drivers]$ ../linux-digilent/scripts/dtc/dtc -I dts -O dtb -o devicetree.dtb devicetree.dts
```

Kako bismo mogli napisati aplikacije koje iskorištavaju stvorenu IP jezgru LED kontrolera potrebno je za kontroler napisati i *driver* u C jeziku koji stvara datoteku naziva *myled* u *Linux/proc* direktoriju. Pisanjem broja u tu datoteku postavlja se status LED dioda na pločici. Taj *driver* zapravo rukovodi osnovnim sustavnim funkcijama poput *write*, *callback*, *open* i *remove*. Program **myled.c** *drivera* dan je u privitku. Taj C kod unosimo naredbom ***vim*** u *drivers* direktoriju te ga pohranjujemo naredbom ***:x***. Za kompilaciju *driver* modula potreban nam je i **Makefile** kojeg također stvaramo u *drivers* direktoriju naredbom *vim*. U otvoreni prozor unosimo sljedeći kod te ga spremamo *:x* naredbom:

Makefile1

```
obj-m := myled.o

all:
	make -C ../linux-digilent/ M=$(PWD) modules

clean:
	make -C ../linux-digilent/ M=$(PWD) clean
```

Sada imamo potrebne *Makefile* i *myled.c* datoteke za kreiranje *driver* modula. *Driver* kreiramo *make* naredbom prije koje je potrebno postaviti izvorišne postavke ISE ili Vivado alata:

```
[petar@petar drivers]$ make ARCH=arm CROSS_COMPILE=arm-xilinx-linux-gnueabi-
```

Kreiran je **myled.ko** modul u *drivers* direktoriju [8].

4.7 Korisnička aplikacije

Na korisniku je sada da napiše aplikaciju za LED kontroler u C jeziku uzimajući u obzir sklopovski dizajn te strukturu *drivera*. Za demonstraciju mi ćemo napisati dvije aplikacije za manipulaciju LED diodama. Prva će biti jednostavno paljenje i gašenje svih osam dioda od korisnika zadanom frekvencijom u Herzima. Druga aplikacija je *dimmer*

koji slijedno pali i gasi LED diode također zadanom frekvencijom. Stoga u radnom direktoriju stvaramo nova dva poddirektorija: *user_app1* i *user_app2*. Naredbom *vim* u oba direktorija unosimo C kod i spremamo ga sa *:x*. C program za prvu aplikaciju zove se **led_blink.c** i dan je u privitku. Drugi C program se zove **led_dimmer.c** te je također priložen u privitku. U programu koristimo **getopt** funkciju za dobivanje argumenata iz komandnog Linux sučelja. Kako bismo kreirali aplikaciju potrebno je u oba direktorija napisati gotovo isti **Makefile**. Slijede redom *Makefile* programi za prvu i drugu aplikaciju:

Makefile2:

```
CC = arm-xilinx-linux-gnueabi-gcc      /* Definirana arhitektura za cross-compile alate */
CFLAGS = -g -lm                        /* Varijable za prosljeđivanje argumenata kompajleru */
                                       /* -g za informaciju o debugiranju */
all : led_blink                        /* -lm za korištenje matematičke biblioteke u led_blink.c*/


led_blink : led_blink.o
        ${CC} ${CFLAGS} -o $@ $^

clean :
        rm -rfv *.o
        rm -rfv led_blink

.PHONY : clean
```

Makefile3:

```
CC = arm-xilinx-linux-gnueabi-gcc   /* Definirana arhitektura za cross-compile alate */
CFLAGS = -g -lm                     /* Varijable za prosljeđivanje argumenata kompajleru */
                                    /* -g za informaciju o debugiranju */
all : led_dimmer                    /* -lm za korištenje matematičke biblioteke u led_dimmer.c*/

led_dimmer : led_dimmer.o
        ${CC} ${CFLAGS} -o $@ $^

clean :
        rm -rfv *.o
        rm -rfv led_dimmer

.PHONY : clean
```

Naredbom *make* kompiliramo ELF datoteke iz gore unesenih C i *Makefile* programa:

[petar@petar user_app1]$ make

[petar@petar user_app2]$ make

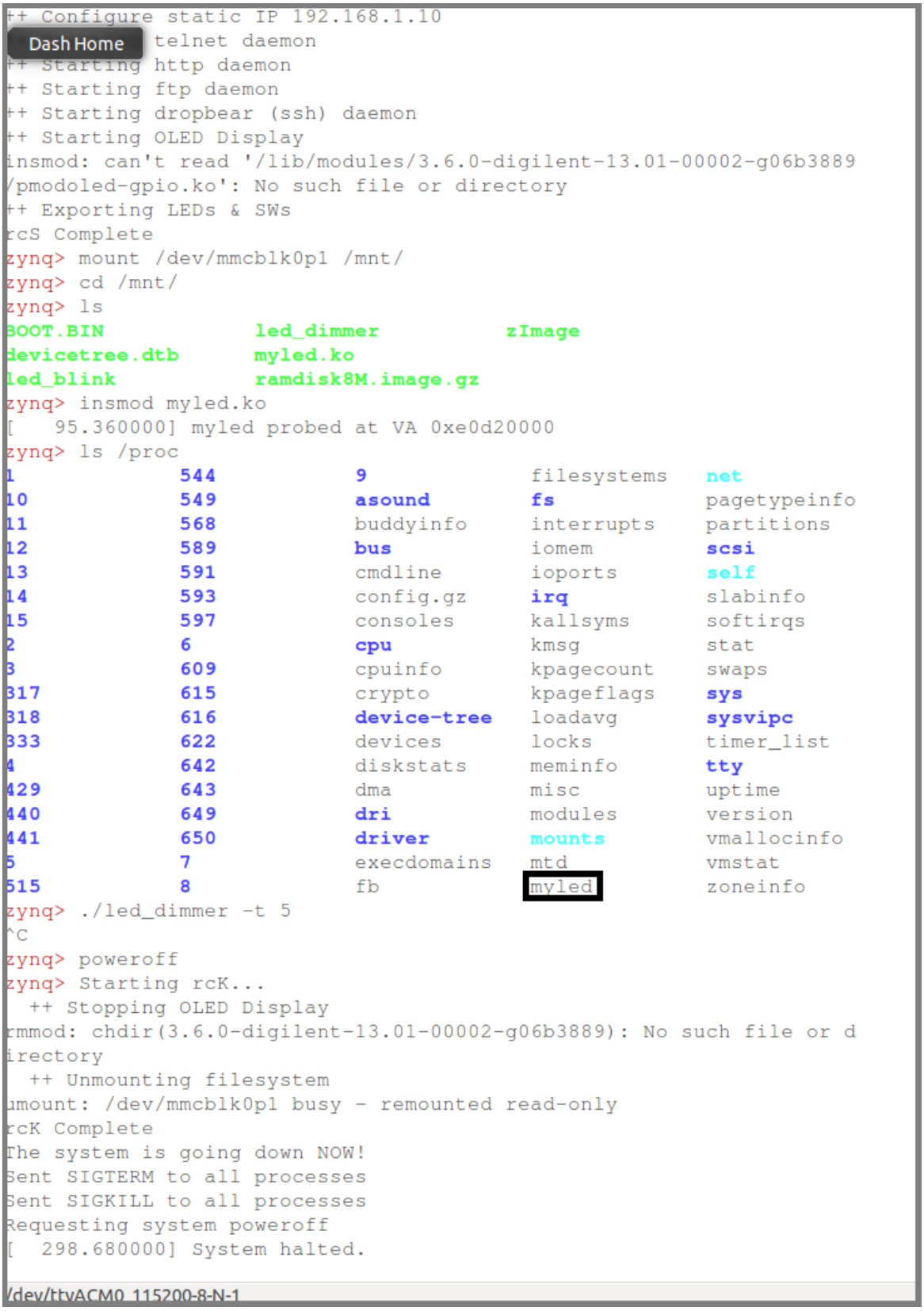
Rezultat su **led_blink** i **led_dimmer** izvršne aplikacije [8].

Na formatiranu SD karticu kopiramo sve datoteke koje smo u prethodnim koracima dobili. One sačinjavaju našu Linux ugradbenu distribuciju sa razvijenim modulom i pripadnim *driverom* nad kojim se izvršavaju korisničke aplikacije. SD kartica mora sadržavati:

- **BOOT.BIN** sliku iz direktorija *ZedBoard_Linux_Design/boot_image*
- **zImage** sliku iz direktorija *linux-digilent/arch/arm/boot*
- **ramdisk8M.image.gz** sliku iz direktorija *ZedBoard_Linux_Design/sd_image*
- **devicetree.dtb** *blob* datoteku iz direktorija *drivers*
- **myled.ko** *driver* iz direktorija *drivers*
- **led_blink** aplikaciju iz direktorija *user_app1*
- **led_dimmer** aplikaciju iz direktorija *user_app2*

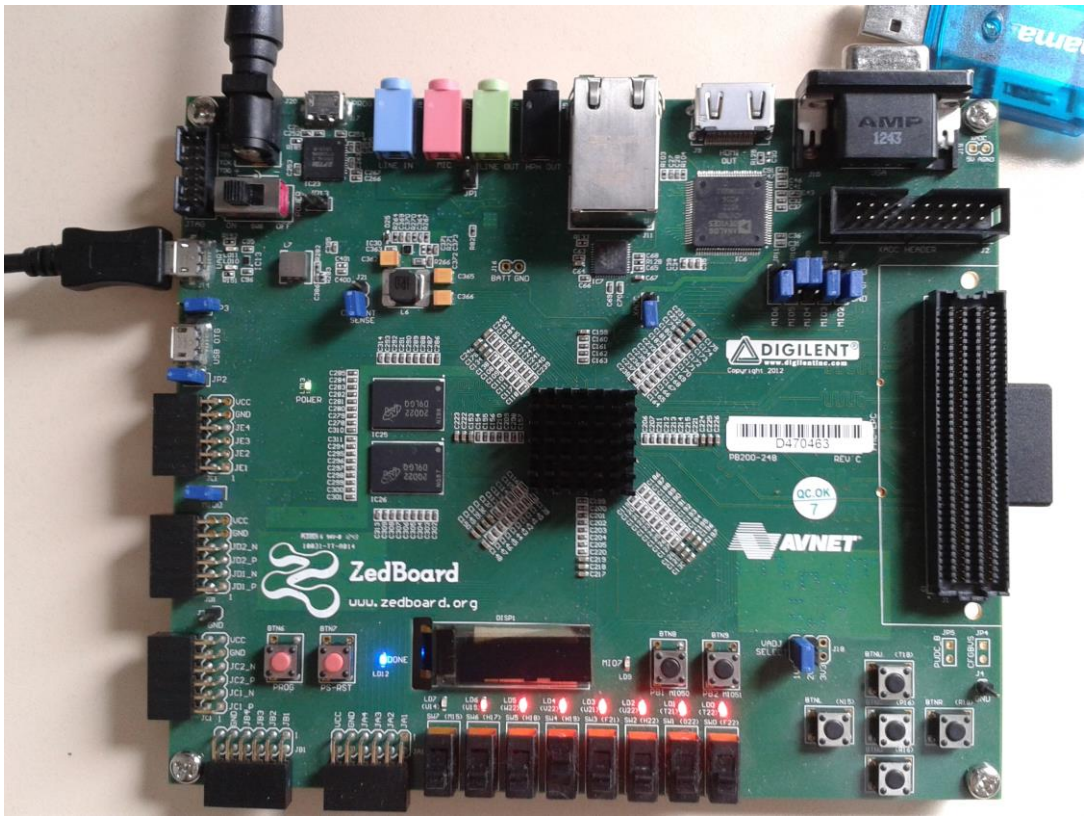
SD karticu sa Linux komponentama utisnemo u ZedBoard pločicu te postavimo kratkospojnike u mod podizanja sa SD kartice. To znači da kratkospojnike MIO2, MIO3 i MIO6 spajamo na masu, a kratkospojnike MIO4 i MIO5 na visoku razinu. Također je potrebno da kratkospojnik JP6 bude spojen. Ispravno postavljeni kratkospojnici se mogu vidjeti na slici 2. Micro USB kabelom povežemo UART priključak na ZedBoard pločici sa osobnim računalom. Na osobnom Linux računalu instaliramo **Gtk terminal** te ga pokrenemo sa naredbom **GtkTerm**. Uključujemo ZedBoard pločicu preklopkom **SW8**. Otvaramo konfiguracijski prozor u Gtk terminalu gdje za port odabiremo **/dev/ttyACM0,** brzinu postavimo na **115200**, odabiremo **8** podatkovnih bitova, **1** stop bit te opciju **no parity** i **no flow control**. U terminalu se pojavljuje naša konzola koja započinje sa podizanjem Linux distribucije preko U-Boot podizača čitanjem zImage slike, devicetree.dtb *bloba* i RamDiska. Slika 27 prikazuje terminal tj. komandno sučelje sa postupcima pokretanja aplikacije na ZedBoard pločici. Prvo je potrebno naredbom **mount** sadržaj prve particije SD kartice dodati u *file system*. Izlistavanjem ***mnt*** direktorija uočavamo sve datoteke koje smo ranije kopirali na SD karticu (zeleno označene datoteke na slici 27). Naredbom **insmod** dodajemo *driver* LED kontrolera u Linux *kernel*. Sada je ***myled*** modul vidljiv u *Linux/proc* direktoriju (crno označen na slici 27). Pisanjem u */proc/myled* možemo paliti ili gasiti LED diode. Konačno možemo pokrenuti izvršne aplikacije. Primjerice, drugu aplikaciju pokrećemo unosom **./led_dimmer -t 5.** Zbog funkcije prosljeđivanja *getopt* korištene u aplikacijskom C programu koristimo zastavicu **-t** za predavanje programu cjelobrojnog argumenta tj. u našem slučaju frekvencije paljenja.

Broj 5 u ovom slučaju označava frekvenciju paljenja dioda od 5 Herza. Na isti način bismo pokrenuli prvu aplikaciju **led_blink**.



Slika 27: *Pokretanje LED aplikacije komandnim sučeljem*

Naredbom **Ctrl-C** prekidamo izvršavanje aplikacije, a naredbom **poweroff** moguće je potpuno isključiti sustav. Slika 28 prikazuje ZedBoard pločicu na kojoj se izvodi demonstracijska LED *dimmer* aplikacija.



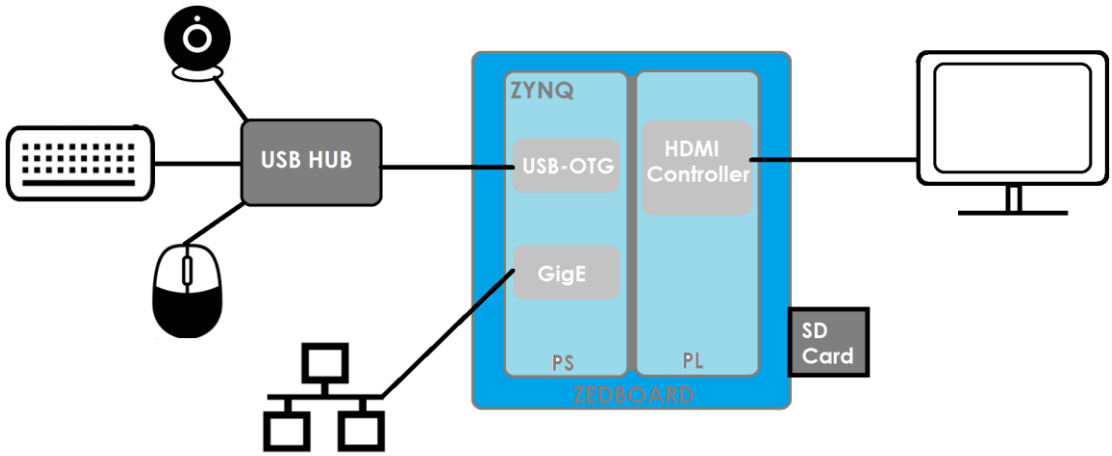
Slika 28: *LED dimmer aplikacija na ZedBoard pločici*

5. ZedBoard samostalno računalo

S aspekta sklopovlja, u prethodnom poglavlju instalirana Linux inačica utilizira samo LED diode razvojne ZedBoard pločice činjenicom kompilacije *drivera* LED kontrolera kojoj je prethodio pripadni sklopovski i programski razvoj. Iako je postupak razvoja i upogonjenja samo tog jednog modula i njime omogućenih aplikacija podosta dug i slojevit, iskorištenost resursa i performansi pripadne Zynq arhitekture je niska. Ciljana Zynq arhitektura je kompozicijom i pristupom visokoperformansna stoga ćemo u ovom poglavlju pokazati njene napredne mogućnosti i veće iskorištenje sklopovlja upravo na primjeru ZedBoard samostalnog računala sa Linaro Ubuntu grafičkim sučeljem.

5.1 Računalni sustav i Zynq-7000 jezgra.

Naš računalni sustav integrira ZedBoard razvojnu pločicu (Zynq-7000 jezgra) sa osnovnim ulaznim i izlaznim jedinicama osobnog računala. Od jedinica potrebno je osigurati USB *hub* uređaj, miš, tipkovnicu, *web* kameru, LCD monitor sa HDMI priključkom te Ethernet kabel za pristup mreži. Na slici 29 je prikazana blok shema prema kojoj spajamo ZedBoard samostalno računalo.



Slika 29: *Blok shema ZedBoard samostalnog računala*

Prikazani su i resursi Zynq jezgre koje računalo iskorištava. To su USB-OTG i gigabitni Ethernet, periferije koje su preko ulazno-izlaznog multipleksora (MIO) spojene na pripadne kontrolere fizičkog sloja i dalje na konektore ZedBoard pločice. Na USB-OTG konektor spajamo USB *hub*, a u USB *hub* spajamo miš, tipkovnicu i *web* kameru. Na

Ethernet konektor spajamo mrežni kabel. Linaro grafičko sučelje računala se ostvaruje na LCD monitoru preko HDMI sučelja koje se u Zynq SoC-u razvija u programabilnoj logici. Stoga ovaj dizajn sadrži IP jezgre HDMI kontrolera implementirane u polje programabilne logike. Ta jezgra objedinjuje odašiljački kontroler, VDMA (engl. *Video Direct Memory Access*) kontroler razmjene podataka i AXI I2C sabirničko sučelje. HDMI IP kontroler je preko ADV7511 odašiljača povezan na HDMI konektor. HDMI kabelom povezujemo konektor i LCD monitor [17].

5.2 Podizanje računala

Dizajn podiže Linux inačicu sa Linaro Ubuntu grafičkim sučeljem i mnoštvom aplikacija za koje je unaprijed obavljena sklopovska i programska prilagodba. Xilinx na svojoj stranici [16] za dani dizajn osigurava repozitorij sa svim datotekama Linux GUI distribucije. Repozitorij sačinjavaju dva komprimirana direktorija. Prvi direktorij sadrži sve datoteke potrebne za podizanje tj. *booting* Linuxa. To su BOOT.BIN slika, devicetree.dtb *blob*, zImage *kernel*, Digilentov logo (logo.bin) te dvije testne skripte za LED diode i OLED prikaznik. Drugi repozitorij sadrži cjelokupni Linaro Ubuntu *file system* koji za Linux inačicu omogućuje trajno spremanje podataka. Kao preduvjet za podizanje Linuxa sa SD kartice potrebno je obaviti sukladno formatiranje SD kartice na dvije particije kao što je opisano u poglavlju 4.1. Time prva particija veličine 1 GB biva FAT formatirana i nazvana **FAT_boot** te će sadržavati *boot* datoteke prvog direktorija. Druga particija veličine 3 GB je EXT4 formatirana i nazvana **EXT4_FS**. Ona će sadržavati Linaro *file system* dan u drugom direktoriju. Na toj se particiji također automatski generira direktorij *lost+found* kojeg možemo ostaviti u *file systemu*. Nakon formatiranja skidamo komprimirane direktorije **Boot File** i **File System** sa Xilinx repozitorija [16]. Kao što je navedeno, *Boot File* direktorij ekstrahiramo na prvu particiju, a *File System* na drugu. Kako bismo detaljnije upoznali dotičnu Linux distribuciju podizanje ćemo promotriti preko serijske veze. Stoga moramo na prvu particiju dodati RamDisk sliku iz poglavlja 4.5. Kratkospojnike MIOx postavimo u mod podizanja sustava sa SD kartice, a kratkospojnike JP2, JP3 i JP6 spojimo kratko. Zatim spojimo napajanje i uključimo računalo preklopkom SW8. Na računalu domaćinu otvorimo **Gtk terminal** gdje za port postavljamo **/dev/ttyACM0,** brzinu postavljamo na **115200**, odabiremo **8** podatkovnih bitova, **1** stop bit te opciju **no parity** i **no flow control**. Uključuje se plava LED dioda te se na LCD

monitoru pokazuje Linux logo, a zatim se pojavljuje Linaro grafičko sučelje. Istodobno se na Gtk terminalu prikazuje *booting*, čije su određene faze prikazane slikom 30.

```
U-Boot 2011.03-00226-gd4000b9-dirty (Jun 19 2012 - 17:52:48)
Copying Linux from SD to RAM...

reading zImage
2472096 bytes read

reading devicetree.dtb
4197 bytes read

reading ramdisk8M.image.gz
3694324 bytes read

## Starting application at 0x00008000 ...
Uncompressing Linux... done, booting the kernel.
[    0.000000] Booting Linux on physical CPU 0
[    0.000000] Linux version 3.3.0-57645-g78a8fc1 (tinghui.wang@DIGILENT_LINUX)
(gcc version 4.6.1 (Sourcery CodeBench Lite 2011.09-50) ) #1 SMP PREEMPT Fri Jun 29 PDT 2012
[    0.000000] CPU: ARMv7 Processor [413fc090] revision 0 (ARMv7), cr=18c5387d
[    0.000000] CPU: PIPT / VIPT nonaliasing data cache, VIPT aliasing instruction cache
[    0.000000] Machine: Xilinx Zynq Platform, model: Xilinx Zynq ZED
[    0.000000] Kernel command line: console=ttyPS0,115200 root=/dev/mmcblk0p2 rw earlyprintk
rootfstype=ext4 rootwait devtmpfs.mount=0
[    0.000000] Memory: 240MB 256MB = 496MB total
[    0.000000] xlnx,ps7-ttc-1.00.a #0 at 0xe0800000, irq=43
[    0.090000] CPU0: thread -1, cpu 0, socket 0, mpidr 80000000
[    0.100000] smp_twd: clock not found: -2
[    0.100000] Calibrating local timer... 399.33MHz.
[    0.170000] hw perfevents: enabled with ARMv7 Cortex-A9 PMU driver, 7 counters available
[    0.170000] Setting up static identity map for 0x2f6110 - 0x2f6144
[    0.270000] CPU1: Booted secondary processor
[    0.310000] CPU1: thread -1, cpu 1, socket 0, mpidr 80000001
[    0.310000] Brought up 2 CPUs
[    0.310000] SMP: Total of 2 processors activated (3188.32 BogoMIPS).
[    0.320000] devtmpfs: initialized
[    0.480000] # #              Board ZED Init             # #
[    0.540000] gpiochip_add: registered GPIOs 0 to 245 on device: xgpiops
[    0.560000] usbcore: registered new interface driver usbfs
[    0.560000] usbcore: registered new interface driver hub
[    0.570000] usbcore: registered new device driver usb
[    0.570000] Advanced Linux Sound Architecture Driver Version 1.0.24.
[    0.710000] eth0, pdev->id -1, baseaddr 0xe000b000, irq 54
[    0.790000] xusbps-ehci xusbps-ehci.0: USB 2.0 started, EHCI 1.00
[    0.790000] hub 1-0:1.0: USB hub found
[    0.790000] hub 1-0:1.0: 1 port detected
[    0.800000] Initializing USB Mass Storage driver...
[    0.800000] usbcore: registered new interface driver usb-storage
[    0.810000] USB Mass Storage support registered.
[    0.810000] Xilinx PS USB Device Controller driver (Apr 01, 2011)
[    0.820000] mousedev: PS/2 mouse device common for all mice
[    0.820000] Linux video capture interface: v2.00
[    0.830000] gspca_main: v2.14.0 registered
[    0.830000] usbcore: registered new interface driver uvcvideo
[    0.830000] USB Video Class driver (1.1.1)
[    0.960000] mmc0: new high speed SDHC card at address 0002
[    0.960000] mmcblk0: mmc0:0002 00000 3.70 GiB
[    1.270000] EXT4-fs (mmcblk0p2): recovery complete
[    1.270000] EXT4-fs (mmcblk0p2): mounted filesystem with ordered data mode. Opts: (null)
[    1.280000] VFS: Mounted root (ext4 filesystem) on device 179:2.
[    1.280000] Freeing init memory: 152K
 * Starting mDNS/DNS-SD daemon                                    [ OK ]
 * Starting network connection manager                            [ OK ]

Welcome to Linaro 11.12 (GNU/Linux 3.3.0-57645-g78a8fc1 armv7l)
root@linaro-ubuntu-desktop:~#
```

Slika 30: *Podizanje Linux distribucije na ZedBoard samostalnom računalu*

Podizanje započinje preko FAT kompatibilnog U-Boot univerzalnog *bootloadera* ukomponiranog unutar BOOT slike. On sadržaj SD kartice preslikava u RAM memoriju i čita ostale elemente FAT particije: sliku *kernela, blob* datoteku te RamDisk *file system*. Slijedi podizanje kernela inicijalno preko prve jezgre - CPU0. Ako pogledamo drugi blok slike 30 uočavamo namještanje *kernel* okruženja preko **bootargs** argumenata za način podizanja Linuxa sa memorijskim *file systemom*. Ti se argumenti namještaju u devicetree.dts izvoru kao što je opisano u poglavlju 4.6, a zatim se kompiliraju u *blob* datoteku. Taj se *file system* kao što je ranije navedeno nalazi na drugoj EXT4 particiji koja je u *dev* direktoriju uređaja instancirana kao **mmcblk0p2** memorijski blok. Dalje su prikazane određene arhitekturalne karakteristike vezane za Zynq-7000 jezgru. Tako uočavamo veličinu unutarnje memorije, simetrični **SMP** operacijski mod procesorskog sustava koji je i ovdje nazvan **processing system 7** (ps7), namještanje takta itd. Podizanje se nastavlja sa inicijalizacijom ZedBoard pločice. Terminal tako prikazuje detektiranje i registriranje uključenih uređaja te dodavanje odgovarajućih *drivera* u *kernel*. Registrira se USB, *hub*, Ethernet, miš, *web* kamera, SD kartica itd. Od *drivera* istaknuto je dodavanje **xgpiops** *drivera* za GPIO kontroler procesorskog sustava, drivera za USB i *hub*, audio i **UVC** (engl. *USB Video Class*) *drivera* za *web* kameru. Podizanje završava konačnim uključivanjem *file systema* u VFS modulu virtualnog organiziranja podataka. Istovremeno pojavljivanju komandnog sučelja gotovo je podizanje grafičkog Linaro sučelja.

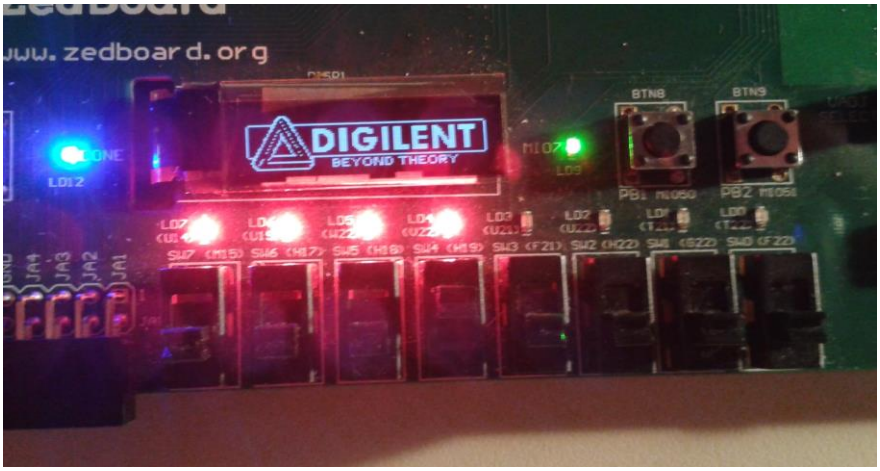
5.3 Korisničke aplikacije

U komandnom sučelju pogledat ćemo sadržaje obiju particija. Slika 31 prikazuje prateće naredbe. Kao i u slučaju LED aplikacija naredbom **mount** dodajemo sadržaj SD particije FAT_boot u postojeći *file system*. Ta se particija nalazi kao memorijski blok **mmcblk0p1** u *dev* direktoriju uređaja koji je izlistan za našu Linux distribuciju ZedBoard samostalnog računala. Izlistavanjem *mnt* direktorija uočavamo sve datoteke prve particije (označene zelenim slovima na slici 31). Kao što smo i ranije naveli tu se nalaze i testne skripte **led_test.sh** i **oled_test.sh** koje i pokrećemo u terminalu. Programski kod skripti dan je u privitku. Prva skripta u sys sistemskom direktoriju manipulira sa 8 GPIO priključaka (točnije sa priključcima 61-68) i pogoni sukcesivno paljenje i gašenje LED-ica. Druga skripta u direktorij uređaja *dev* dodaje *zed_oled* čvor za ASCII znakove kojeg puni sa Digilentovim logom tj. sa logo.bin datotekom. To rezultira Digilentovim logom na OLED

prikazniku. Na slici 32 prikazane su LED diode i OLED prikaznik za vrijeme izvođenja skripti. Za pregled datoteka druge particije EXT4_FS *mount* naredbom uključujemo memorijski blok *mmcblk0p2* i izlistavamo datoteke (plavo označene na slici 31).

```
root@linaro-ubuntu-desktop:~# cd /dev
root@linaro-ubuntu-desktop:/dev# ls
bus                 mem                 ram4    tty15  tty33  tty51      vcs
char                mmcblk0             ram5    tty16  tty34  tty52      vcs1
console             mmcblk0p1           ram6    tty17  tty35  tty53      vcs2
cpu_dma_latency     mmcblk0p2           ram7    tty18  tty36  tty54      vcs3
dri                 network_latency     ram8    tty19  tty37  tty55      vcs4
fb0                 network_throughput  ram9    tty2   tty38  tty56      vcs5
fd                  null                random  tty20  tty39  tty57      vcs6
full                psaux               shm     tty21  tty4   tty58      vcs7
input               ptmx                snd     tty22  tty40  tty59      vcsa
kmsg                pts                 stderr  tty23  tty41  tty6       vcsa1
log                 ram0                stdin   tty24  tty42  tty60      vcsa2
loop0               ram1                stdout  tty25  tty43  tty61      vcsa3
loop1               ram10               tty     tty26  tty44  tty62      vcsa4
loop2               ram11               tty0    tty27  tty45  tty63      vcsa5
loop3               ram12               tty1    tty28  tty46  tty7       vcsa6
loop4               ram13               tty10   tty29  tty47  tty8       vcsa7
loop5               ram14               tty11   tty3   tty48  tty9       watchdog
loop6               ram15               tty12   tty30  tty49  ttyPS0     zero
loop7               ram2                tty13   tty31  tty5   urandom
loop-control        ram3                tty14   tty32  tty50  usbdev1.1
root@linaro-ubuntu-desktop:/dev# mount /dev/mmcblk0p1 /mnt/
root@linaro-ubuntu-desktop:/dev# cd /mnt/
root@linaro-ubuntu-desktop:/mnt# ls
BOOT.BIN         led_test.sh  oled_test.sh         zImage
devicetree.dtb   logo.bin     ramdisk8M.image.gz
root@linaro-ubuntu-desktop:/mnt# ./led_test.sh
root@linaro-ubuntu-desktop:/mnt# ./oled_test.sh
root@linaro-ubuntu-desktop:/mnt# ^C
root@linaro-ubuntu-desktop:/mnt# cd

root@linaro-ubuntu-desktop:~# mount /dev/mmcblk0p2 /mnt/
root@linaro-ubuntu-desktop:~# cd /mnt/
root@linaro-ubuntu-desktop:/mnt# ls
bin   dev  home  lost+found  mnt  proc  run   selinux  sys  usr
boot  etc  lib   media       opt  root  sbin  srv      tmp  var
root@linaro-ubuntu-desktop:/mnt#
```

Slika 31: *Sadržaj SD particija i pokretanje testnih skripti*



Slika 32: *Izvođenje testnih skripti*

Na primjeru testnih skripti prikazali smo način koji upravljanjem elementima podatkovne organizacije tj. *file systema* manipulira sklopovljem. Zapravo svako se izvođenje aplikacija na visokoj razini bazira na takvoj vrsti operacija u *file system* prostoru.

Vratimo se osnovnoj funkciji ZedBoard samostalnog računala, a to je izvođenje uobičajenih aplikacija unutar Linaro Ubuntu grafičkog sučelja već odavno prikazanog na LCD monitoru. Omogućen je pristup internetu kojeg vršimo preko **Mozilla Firefox** preglednika. Unutar *file systema* nalaze se brojni programi kao primjerice programi za igrice. Aplikaciju sa *web* kamerom pokrećemo odabirom **VLC media player** programa. U VLC izborniku odabiremo opciju **Media→Open Capture Device** i u otvorenom prozoru odabiremo **Play**. Ukoliko se pojavi upozorenje pri pokretanju *web* kamere ignoriramo ga sa **Close**. Otvara se prozor sa direktnim prikazom kamere. Na slici 33 prikazano je ZedBoard samostalno računalo koje izvodi internet i video aplikacije [17].

Slika 33: *Internet i video aplikacija na ZedBoard samostalnom računalu*

6. Zaključak

Na primjeru LED aplikacija i ZedBoard samostalnog računala opisani su osnovni koncepti razvoja i pokretanja aplikacija na Zynq-7000 arhitekturi pod operacijskim sustavom Linux. Postupak podizanja Linux distribucije na SoC-u sa ciljem izvođenja raznih aplikacija bazira se na poznavanju ciljane arhitekture, vještini rada u razvojnim alatima i otvorenosti programske podrške, kako samog Linux *kernela* tako i osnovnih konfiguracijskih datoteka SoC razvojne pločice. Stoga ovim radom opisani postupci razvoja aplikacija postaju korisne upute za laboratorijske vježbe računalnog inženjerstva u domeni ugradbenih operacijskih sustava. Također pokrenute LED aplikacije i ZedBoard samostalno računalo postaju prateći demonstracijski sadržaj.

Ipak višedimenzionalnost razvoja i naprednost Zynq-7000 arhitekture sustava na čipu zahtijeva visok stupanj znanja i iskustva kako u elektroničkoj tako i u programskoj domeni računalnog inženjerstva. Stoga je i razvoj sofisticiranijih Zynq-7000 digitalnih sustava pod operacijskim sustavom fokus i posao cijelog tima računalnih, elektroničkih i programskih inženjera.

7. Literatura

[1] A. Donato, F. Ferrandi, M. Santambrogio, D. Scuito, *Operating system support for dynamically reconfigurable SoC architectures*, IEEE, 2005

[2] Hu Jie, Zhang Gen-bao, *Research transplanting method of embedded Linux kernel based on ARM platform*, IEEE International Conference of Science and Management Engineering, 2010

[3] P. Matković, *SoC (System on Chip) Linux Application*, Institute of Computer Technology - TU Wien, ožujak 2015

[4] Xilinx, MathWorks, Analog Devices, Avnet, *Integrated Software-Defined Radio on Zynq-7000 All Programmable SoC Design Seminar*, 2013

[5] J. McDougall, *Simple AMP: Bare-Metal System Running on Both Cortex-A9 Processors*, Xilinx Application Note XAPP1079, siječanj 2014

[6] J. McDoughall, *Simple AMP Running Linux and Bare-Metal System on Both Zynq SoC Processors*, Xilinx Application Note XAPP1078, veljača 2013

[7] Xilinx, *Zynq-7000 All Programmable SoC Software Developers Guide*, Version 10, ožujak 2015

[8] Digilent, *Embedded Linux Hands-on 8 - ZedBoard*, Pullman, ožujak 2013

[9] Digilent, *Getting Started With Embedded Linux - ZedBoard*, Pullman, ožujak 2013

[10] Avnet, *ZedBoard - Getting Started Guide*, Version 7, siječanj 2014

[11] Xilinx, *Zynq-7000 All Programmable SoCs*, Product Guide, 2014

[12] Xilinx, *A generation ahead for smarter systems: 9 reasons why the Xilinx Zynq-7000 All Programmable SOC platform is the smartest solution*, 2014

[13] ZedBoard Wiki, http://zedboard.org

[14] Xilinx, *Zynq-7000 All Programmable SoCs*, http://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html

[15] Xilinx Wiki, http://www.wiki.xilinx.com

[16] Xilinx, *Single Board Computer Demo*, http://www.xilinx.com/support/university/boards-portfolio/xup-boards/XUPZedBoard/ZedBoard-Single-Board-Computer-Demo.html

[17] Xilinx, *Single Board Computer - Linux Desktop on an Embedded Zynq Platform*, ZedBoard-Webcam Documentation, 2012

RAZVOJ DEMONSTRACIJSKIH APLIKACIJA NA ZYNQ-7000 SOC ARHITEKTURI POD OPERACIJSKIM SUSTAVOM LINUX

Sažetak

Na primjeru manipulativnih LED aplikacija upoznali smo se sa osnovnim fazama u razvoju aplikacija koje se izvode na Zynq-7000 sustavu na čipu pod operacijskim sustavom Linux. Te faze uključuju sklopovski razvoj i konfiguraciju polja programabilne logike, prilagodbu programske podrške, generiranje BOOT slike i *blob* datoteke stabla uređaja, kompilaciju Linux *kernela* te pisanje i izgradnju *drivera* i samih LED aplikacija. S druge strane, na primjeru ZedBoard samostalnog računala s grafičkom Linux distribucijom demonstrirali smo napredne aplikativne, ali i arhitekturalne mogućnosti Zynq-7000 SoC arhitekture.

Ključne riječi: *Zynq-7000, sustav na čipu, ZedBoard, Linux, operacijski sustav, aplikacija, driver, arhitektura, LED diode, ugradbeno računalo*

DEVELOPMENT OF SOC DEMONSTRATION APPLICATIONS BASED ON ZYNQ-7000 ARCHITECTURE UNDER LINUX OPERATING SYSTEM

Abstract

LED manipulation applications have acquainted us with basic steps of Zynq-7000 system on chip application development under Linux operating system. Those phases include hardware development, programmable logic configuration, software adaptation, BOOT image and device tree blob generation, Linux kernel compilation along with writing and building of both driver and applications. Thereafter, on the example of ZedBoard single computer with graphical Linux distribution we have demonstrated advanced applicative and architectural possibilities of Zynq-7000 SoC architecture.

Keywords: *Zynq-7000, system on chip, ZedBoard, Linux, operating system, application, driver, architecture, LED diodes, embedded computer*

Privitak

Myled.c [8]

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <asm/uaccess.h>              /* Needed for copy_from_user */
#include <asm/io.h>                   /* Needed for IO Read/Write Functions */
#include <linux/proc_fs.h>            /* Needed for Proc File System Functions */
#include <linux/seq_file.h>           /* Needed for Sequence File Operations */
#include <linux/platform_device.h>    /* Needed for Platform Driver Functions */
#include <linux/slab.h>

/* Define Driver Name */
#define DRIVER_NAME "myled"

unsigned long *base_addr;             /* Vitual Base Address */
struct resource *res;                 /* Device Resource Structure */
unsigned long remap_size;             /* Device Memory Size */

/* Write operation for /proc/myled
 * ------------------------------------
 *  When user cat a string to /proc/myled file, the string will be stored in
 *  const char __user *buf. This function will copy the string from user
 *  space into kernel space, and change it to an unsigned long value.
 *  It will then write the value to the register of myled controller,
 *  and turn on the corresponding LEDs eventually.*/
 static ssize_t proc_myled_write(struct file *file, const char __user * buf,
                size_t count, loff_t * ppos)
 {
     char myled_phrase[16];
     u32 myled_value;
     if (count < 11) {
         if (copy_from_user(myled_phrase, buf, count))
             return -EFAULT;
        myled_phrase[count] = '\0';
    }

    myled_value = simple_strtoul(myled_phrase, NULL, 0);
    wmb();
    iowrite32(myled_value, base_addr);
    return count;
 }

 /* Callback function when opening file /proc/myled
  * ------------------------------------------------------
  *  Read the register value of myled controller, print the value to
  *  the sequence file struct seq_file *p. In file open operation for /proc/myled
  *  this callback function will be called first to fill up the seq_file,
```

```
 *  and seq_read function will print whatever in seq_file to the terminal */
static int proc_myled_show(struct seq_file *p, void *v)
{
    u32 myled_value;
    myled_value = ioread32(base_addr);
    seq_printf(p, "0x%x", myled_value);
    return 0;
}

/* Open function for /proc/myled
  * ------------------------------------
  *  When user want to read /proc/myled (i.e. cat /proc/myled), the open function
  *  will be called first. In the open function, a seq_file will be prepared and the
  *  status of myled will be filled into the seq_file by proc_myled_show function.*/
static int proc_myled_open(struct inode *inode, struct file *file)
 {
     unsigned int size = 16;
     char *buf;
     struct seq_file *m;
     int res;

     buf = (char *)kmalloc(size * sizeof(char), GFP_KERNEL);
     if (!buf)
         return -ENOMEM;

     res = single_open(file, proc_myled_show, NULL);

     if (!res) {
         m = file->private_data;
         m->buf = buf;
         m->size = size;
     } else {
         kfree(buf);
     }
      return res;
 }

 /* File Operations for /proc/myled */
 static const struct file_operations proc_myled_operations = {
     .open = proc_myled_open,
     .read = seq_read,
     .write = proc_myled_write,
     .llseek = seq_lseek,
     .release = single_release
 };

 /* Shutdown function for myled
  * ------------------------------------
  *  Before myled shutdown, turn-off all the leds*/
static void myled_shutdown(struct platform_device *pdev)
{
```

```
   iowrite32(0, base_addr);
}

/* Remove function for myled
 * ---------------------------------
 *  When myled module is removed, turn off all the leds first,
 *  release virtual address and the memory region requested.*/
static int myled_remove(struct platform_device *pdev)
{
    myled_shutdown(pdev);

    /* Remove /proc/myled entry */
    remove_proc_entry(DRIVER_NAME, NULL);

    /* Release mapped virtual address */
    iounmap(base_addr);

    /* Release the region */
    release_mem_region(res->start, remap_size);
    return 0;
}

/* Device Probe function for myled
 * ------------------------------------
 *  Get the resource structure from the information in device tree.
 *  request the memory regioon needed for the controller, and map it into
 *  kernel virtual memory space. Create an entry under /proc file system
 *  and register file operations for that entry.*/
static int myled_probe(struct platform_device *pdev)
{
    struct proc_dir_entry *myled_proc_entry;
    int ret = 0;

    res = platform_get_resource(pdev, IORESOURCE_MEM, 0);
    if (!res) {
        dev_err(&pdev->dev, "No memory resource\n");
        return -ENODEV;
    }

    remap_size = res->end - res->start + 1;
    if (!request_mem_region(res->start, remap_size, pdev->name)) {
        dev_err(&pdev->dev, "Cannot request IO\n");
        return -ENXIO;
    }

    base_addr = ioremap(res->start, remap_size);
    if (base_addr == NULL) {
        dev_err(&pdev->dev, "Couldn't ioremap memory at 0x%08lx\n",
            (unsigned long)res->start);
        ret = -ENOMEM;
        goto err_release_region;
```

```
    }
    myled_proc_entry = proc_create(DRIVER_NAME, 0, NULL,
                    &proc_myled_operations);
    if (myled_proc_entry == NULL) {
        dev_err(&pdev->dev, "Couldn't create proc entry\n");
        ret = -ENOMEM;
        goto err_create_proc_entry;
    }

    printk(KERN_INFO DRIVER_NAME " probed at VA 0x%08lx\n",
           (unsigned long) base_addr);

    return 0;

 err_create_proc_entry:
    iounmap(base_addr);
 err_release_region:
    release_mem_region(res->start, remap_size);

    return ret;
}

/* device match table to match with device node in device tree */
static const struct of_device_id myled_of_match[] = {
    {.compatible = "dglnt,myled-1.00.a"},       /*Kompatabilnost sa devicetree.dts čvorom - slika 25 */
  {},
};

MODULE_DEVICE_TABLE(of, myled_of_match);

/* platform driver structure for myled driver */
static struct platform_driver myled_driver = {
    .driver = {
          .name = DRIVER_NAME,
          .owner = THIS_MODULE,
          .of_match_table = myled_of_match},
    .probe = myled_probe,
    .remove = myled_remove,
    .shutdown = myled_shutdown
};

/* Register myled platform driver */
module_platform_driver(myled_driver);

/* Module Informations */
MODULE_AUTHOR("Digilent, Inc.");
MODULE_LICENSE("GPL");
MODULE_DESCRIPTION(DRIVER_NAME ": MYLED driver (Simple Version)");
MODULE_ALIAS(DRIVER_NAME);
```

Led_blink.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <math.h>

int main(int argc, char **argv)
{
	int opt,freq_Hz;
	int time;
	opt = getopt(argc, argv, "nt:");			//insertred Frequency in Herz - integer
	freq_Hz = atoi(optarg);
	time = round(1/(float)freq_Hz*1000000);		//Period of LED blinking in microseconds
	FILE* fp;
	while(1) {
		fp = fopen("/proc/myled", "w");
		if(fp == NULL) {
			printf("Cannot open /proc/myled for write\n");
			return -1;
		}
		fputs("0xFF\n", fp);
		fclose(fp);
		usleep(time);

		fp = fopen("/proc/myled", "w");
		if(fp == NULL) {
			printf("Cannot open /proc/myled for write\n");
			return -1;
		}
		fputs("0x00\n", fp);
		fclose(fp);
		usleep(time);
	}
	return 0;
}
```

Led_dimmer.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <math.h>

int main(int argc, char **argv)
{
	int opt,freq_Hz;
	int time;
	opt = getopt(argc, argv, "nt:");			//insertred Frequency in Herz - inte
	freq_Hz = atoi(optarg);
```

```
time = round(1/(float)freq_Hz*1000000);          //Period of LED blinking in microseconds
FILE* fp;
while(1) {
        fp = fopen("/proc/myled", "w");
        if(fp == NULL) {
                printf("Cannot open /proc/myled for write\n");
                return -1;
        }
        fputs("0x01\n", fp);
        fclose(fp);
        usleep(time);

        fp = fopen("/proc/myled", "w");
        if(fp == NULL) {
                printf("Cannot open /proc/myled for write\n");
                return -1;
        }
        fputs("0x03\n", fp);
        fclose(fp);
        usleep(time);

        fp = fopen("/proc/myled", "w");
        if(fp == NULL) {
                printf("Cannot open /proc/myled for write\n");
                return -1;
        }
        fputs("0x07\n", fp);
        fclose(fp);
        usleep(time);

        fp = fopen("/proc/myled", "w");
        if(fp == NULL) {
                printf("Cannot open /proc/myled for write\n");
                return -1;
        }
        fputs("0x0F\n", fp);
        fclose(fp);
        usleep(time);

        fp = fopen("/proc/myled", "w");
        if(fp == NULL) {
                printf("Cannot open /proc/myled for write\n");
                return -1;
        }
        fputs("0x1F\n", fp);
        fclose(fp);
        usleep(time);

        fp = fopen("/proc/myled", "w");
        if(fp == NULL) {
                printf("Cannot open /proc/myled for write\n");
```

```
        return -1;
}
fputs("0x3F\n", fp);
fclose(fp);
usleep(time);

fp = fopen("/proc/myled", "w");
if(fp == NULL) {
        printf("Cannot open /proc/myled for write\n");
        return -1;
}
fputs("0x7F\n", fp);
fclose(fp);
usleep(time);

fp = fopen("/proc/myled", "w");
if(fp == NULL) {
        printf("Cannot open /proc/myled for write\n");
        return -1;
}
fputs("0xFF\n", fp);
fclose(fp);
usleep(time);

fp = fopen("/proc/myled", "w");
if(fp == NULL) {
        printf("Cannot open /proc/myled for write\n");
        return -1;
}
fputs("0x7F\n", fp);
fclose(fp);
usleep(time);

fp = fopen("/proc/myled", "w");
if(fp == NULL) {
        printf("Cannot open /proc/myled for write\n");
        return -1;
}
fputs("0x3F\n", fp);
fclose(fp);
usleep(time);

fp = fopen("/proc/myled", "w");
if(fp == NULL) {
        printf("Cannot open /proc/myled for write\n");
        return -1;
}
fputs("0x1F\n", fp);
fclose(fp);
usleep(time);
```

```
                fp = fopen("/proc/myled", "w");
                if(fp == NULL) {
                        printf("Cannot open /proc/myled for write\n");
                        return -1;
                }
                fputs("0x0F\n", fp);
                fclose(fp);
                usleep(time);

                fp = fopen("/proc/myled", "w");
                if(fp == NULL) {
                        printf("Cannot open /proc/myled for write\n");
                        return -1;
                }
                fputs("0x07\n", fp);
                fclose(fp);
                usleep(time);

                fp = fopen("/proc/myled", "w");
                if(fp == NULL) {
                        printf("Cannot open /proc/myled for write\n");
                        return -1;
                }
                fputs("0x03\n", fp);
                fclose(fp);
                usleep(time);

                fp = fopen("/proc/myled", "w");
                if(fp == NULL) {
                        printf("Cannot open /proc/myled for write\n");
                        return -1;
                }
                fputs("0x01\n", fp);
                fclose(fp);
                usleep(time);

                fp = fopen("/proc/myled", "w");
                if(fp == NULL) {
                        printf("Cannot open /proc/myled for write\n");
                        return -1;
                }
                fputs("0x00\n", fp);
                fclose(fp);
                usleep(time);
        }

        return 0;
}
```

Led_test.sh [16]

```
#!/bin/bash
for i in {61..68}
do
        if [ ! -d /sys/class/gpio/gpio$i ]; then
                echo $i > /sys/class/gpio/export
        fi
done

for i in {61..68}
do
        echo out > /sys/class/gpio/gpio$i/direction
        echo 1 > /sys/class/gpio/gpio$i/value
        sleep 1
done

for i in {61..68}
do
        echo 0 > /sys/class/gpio/gpio$i/value
        sleep 1
done
```

Oled_test.sh [16]

```
#!/bin/bash
mknod /dev/zed_oled c 252 0
cat logo.bin > /dev/zed_oled
```