UNIVERZA V LJUBLJANI
FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Marko Čelan

Implementacija PCI vmesnika v FPGA

DIPLOMSKO DELO
NA UNIVERZITETNEM ŠTUDIJU

Mentor: doc. dr. Patricio Bulić

Ljubljana, 2009

Univerza
v Ljubljani

Fakulteta za računalništvo
in informatiko

Tržaška 25
1000 Ljubljana, Slovenija
telefon: 01 476 84 11
faks: 01 426 46 47
www.fri.uni-lj.si
e-mail: dekanat@fri.uni-lj.si

Št. naloge: 01570/2009
Datum: 05.05.2009

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

Kandidat: **MARKO ČELAN**

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

Vrsta naloge: Diplomsko delo univerzitetnega študija

Tematika naloge:

V FPGA implementirajte generični PCI vmesnik. Za implementacijo uporabite razvojno ploščico s PCI robnim konektorjem in Spartan 3 vezjem. Vmesnik mora omogočati priklop generične naprave preko PCI vodila in mora vsebovati konfiguracijski prostor. V programskem jeziku C nato implementirajte še gonilnik za PCI vmesnik za operacijski sistem Linux.

Mentor:
doc. dr. Patricio Bulić

Fakulteta za računalništvo in informatiko • UNIVERZA V LJUBLJANI • 2

Dekan:
prof. dr. Franc Solina

Namesto te strani **vstavite** original izdane teme diplomskega dela s podpisom mentorja in dekana ter žigom fakultete, ki ga diplomant dvigne v študentskem referatu, preden odda izdelek v vezavo!

IZJAVA O AVTORSTVU

diplomskega dela

Spodaj podpisani/-a Marko Čelan,

z vpisno številko 63980020,

sem avtor diplomskega dela z naslovom:

Izdelava PCI vmesnika v FPGA

S svojim podpisom zagotavljam, da:

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

V Ljubljani, dne 2. julij 2009 Podpis avtorja:

Zahvala

Zahvaljujem se mentorju doc. dr. Patriciu Buliću za več kot korektno mentorstvo. Zahvaljujem se kolegoma Davidu Pristovniku in Gašperju Kozaku za vse najdene napake v tem delu. Zahvaljujem se svojim staršem za potrpežljivost ter moralno in materialno podporo tekom študija. Posebna zahvala pa gre ženi Mirjam za vso požrtvovalnost, podporo in vzpodbudo, ki jo je izkazala tako tekom študija, kot tudi med nastankom tega dela.

To delo posvečam ženi Mirjam in sinku Matiji

Kazalo

Seznam uporabljenih kratic in simbolov

API	Application Programming Interface
BAR	Base Address Register
BIOS	Basic I/O System
BSD	Berkeley Software Distribution
CPE	Centralna Procesna Enota
DAC	Digital-to-Analog Converter
DIP	Dual Inline Package
EEPROM	Electrically Erasable PROM
FPGA	Field-Programmable Gate Array
GNU	GNU is Not Unix
GPL	General Public Licence
I/O	Input/Output
ISA	Industry Standard Architecture
ID	Identification
JTAG	Joint Test Action Group
LED	Light-Emitting Diode
CPE	Centralna procesna enota
PCI	Peripheral Component Interconnect
PCISIG	PCI Special Interest Group
PROM	Programmable Read-Only Memory
RAM	Random Access Memory
SCSI	Small Computer System Interface
SRAM	Static RAM
USB	Universal Serial Bus
V/I	Vhod/Izhod
VGA	Video Graphic Adapter
VHDL	VHSIC Hardware Description Language
VHSIC	Very-High-Speed Integrated Circuit

Povzetek

Diplomsko delo nas uvede v osnove standarda PCI (Peripheral Component Interconnect) ter popelje skozi različne vidike razvoja in delovanja PCI vmesnika. Po eni strani predstavi PCI vmesnik kot ga vidi snovalec naprave - torej kot različne logične sklope vezij, povezane med sabo, po drugi strani pa ga prikaže skozi oči razvijalca gonilnika - kot naslovni prostor in nabor registrov. Razloži glavne signale, ki so pomembni pri razumevanju tega standarda. Dodobra razdela potek konfiguracije PCI vmesnika ter pomen različnih registrov, ki sodelujejo v tem procesu. Kjer je to potrebno, razloži pomen posameznih bitov v registrih. Podrobneje se spusti v delovanje osnovnih podatkovnih in konfiguracijskih transakcij in pokaže kako potekajo skozi posamezne urine cikle. Za konkretne vrednosti konfiguracijskih registrov opiše obnašanje naprave. Z referenčno implementacijo postavi izhodišča za razvoj strojnega vmesnika.
Na kratko se dotakne delovanja operacijskih sistemov na splošno. Potem se spusti v delovanje operacijskega sistema GNU/Linux. Pove, katere so naloge jedra in katere so naloge gonilnika. Razloži način klasifikacije gonilnikov v operacijskem sistemu GNU/Linux (GNU is Not Unix). Na primeru pokaže kako poteka razvoj enostavnega gonilnika. Na koncu pokaže kako delujeta gonilnik in PCI vmesnik, ki sta nastala tekom študije PCI standarda.

Ključne besede:

PCI, vodilo, strojni vmesnik, gonilnik, operacijski sistem, Linux, VHDL

Abstract

This thesis introduces the basics of the PCI standard and shows us different aspects of development, and how the PCI interface works. First, it introduces the PCI interface as the hardware device developer sees it - as different logical circuits interconnected, and second through the eyes of the driver developer - as an address space and a set of registers.
It explains the main signals that are important for understanding this standard. It shows the configuration procedure of the PCI interface and the meaning of different registers that are involved in the process. Where important, it explains the meaning of certain bits in the registers.
In greater detail, it elaborates the operation of the basic data and configuration transactions, and how they are used within specific clock cycles. It shows us how the device behaves, given some specific values of configuration registers. It sets up a starting point to developing a hardware interface.
In shorter detail this thesis shows how operating systems work in general, and explains how GNU/Linux operates. It explains which are the roles of the kernel, which are the roles of the driver, and how drivers are classified in GNU/Linux operating system. It shows an example of how to develop a simple driver. Finally, it shows how the driver and PCI interface, built for this thesis, operate.

Key words:

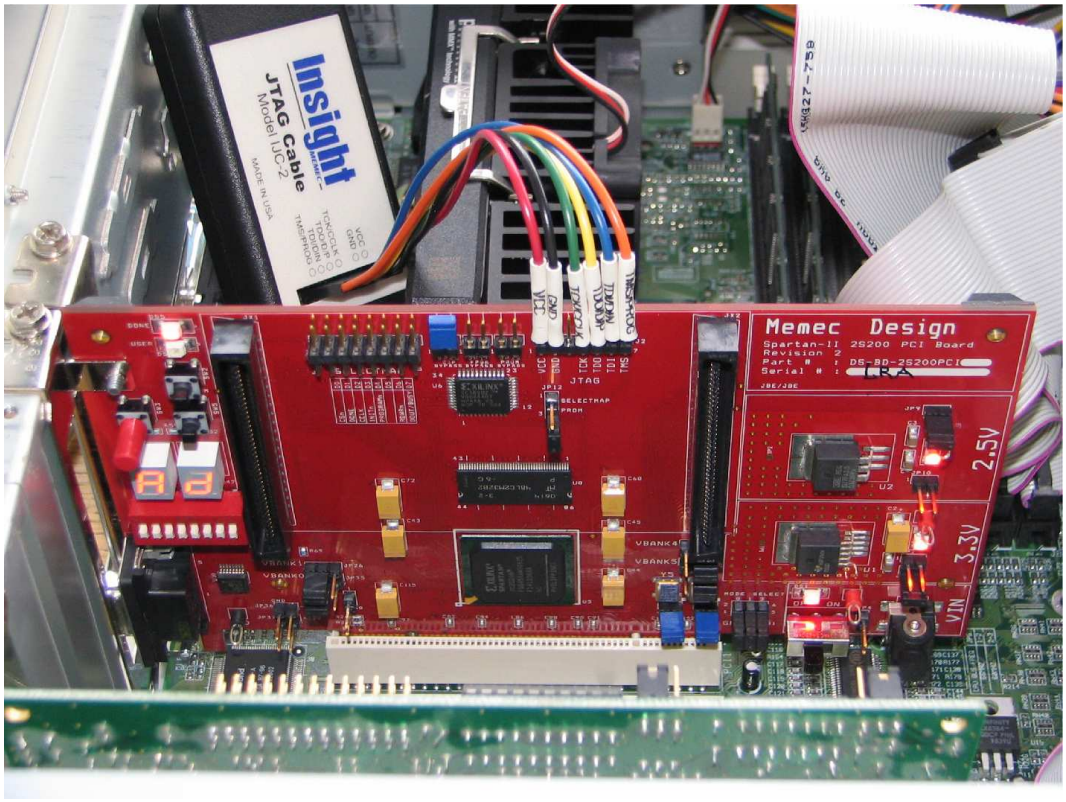
PCI, bus, hardware interface, driver, operating system, Linux, VHDL

Poglavje 1

Uvod

Ob koncu 70. in začetku 80. let prejšnjega stoletja so si računalniki iz laboratorijev utrli pot tudi v domove. Temu je botroval predvsem zelo hiter razvoj na tem področju, ki je povzročil, da so cene računalniških sistemov močno padle. Ker smo si ljudje zelo različni, imamo tudi zelo različne potrebe in računalniki, ki si jih lastimo, niso nobena izjema. Zato je ena ključnih značilnosti današnjega računalnika njegova modularnost - to je zmožnost prilagajanja strojne opreme individualnim potrebam. Na nivoju signalov je to najučinkoviteje uresničiti z uporabo vodil.
Seveda so pojem vodila poznali že prej, vendar se je s tem prebojem računalnikov pokazala potreba po standardizaciji vodila, ki je razširljivo, poceni, energijsko varčno itd. Na trg je prišlo kar nekaj standardov, PCI pa je danes verjetno najbolj razširjen med njimi. Poudariti je treba, da je PCI v računalniški dobi že precej star standard, pa vendar dandanes težko kupite računalnik, ki ne bi vseboval PCI vodila v takšni ali drugačni obliki.

V diplomskem delu sem opisal celovit razvoj PCI vmesnika z gonilnikom.



Slika 1.1: PCI vmesnik v praksi (vir:[1])

Poglavje 2

PCI

2.1 Na kratko o vodilih

PCI vodilo je namenjeno predvsem uporabi v osebnih računalnikih, strežnikih in prenosnikih ter ni namenjen uporabi v vgrajenih sistemih in manjših napravah (npr. mobilnih telefonih).

2.1.1 Kaj je vodilo

Tipičen Von Neumannov računalnik oziroma računalniški sistem je sestavljen iz treh osnovnih komponent:

- CPE (Centralna Procesna Enota),
- glavnega pomnilnika,
- in vhodno-izhodnega sistema.

Seveda je potrebno te komponente med sabo povezati. Da bi čimbolj zmanjšali število potrebnih nožic na čipih (predvsem CPE) in s tem ceno izdelave, so vpeljali koncept vodila. Bistvena značilnost vodil je, da imajo odcepe. Če sta dve napravi med seboj povezani brez odcepov, govorimo o povezavi točka-v-točko in ne o vodilu.
Uporaba vodil je zelo pripravna, saj teoretično omogoča zelo enostavno priklapljanje velikega števila naprav. V praksi se izkaže, da hitro naletimo na zgornjo mejo števila priklopljenih naprav, še posebej takrat, ko želimo omogočati višje hitrosti prenosov po vodilu.

Vse naprave, ki komunicirajo preko istega vodila, uporabljajo iste fizične povezave, kar pomeni, da lahko na vodilo pošilja signale samo ena naprava naenkrat, ostale naprave pa lahko le berejo z vodila oziroma čakajo, da se le-to sprosti. To pomeni, da se morajo naprave zavedati ostalih naprav s katerimi tekmujejo za "lastništvo" nad vodilom. Procesu tekmovanja za vodilo pravimo arbitraža, ki terja svoj čas, med katerim je vodilo neuporabno za prenos podatkov.
Poleg tega se zaradi fizičnih lastnosti vodila (to so odcepi in veliko priklopljenih naprav) pojavljajo različne težave. Zaradi odcepov se na vodilu pojavljajo odboji, ki se izrazijo kot šum ali motnja, ta pa lahko povzroči napačno vrednost signalov. Veliko priklopljenih naprav pa lahko povzroči nizko impedanco vodila, kar vodi k veliki tokovni obremenitvi naprave, ki piše na vodilo, to pa lahko vodi k prepočasnemu spreminjanju napetosti in spet se zgodi, da preberemo napačno vrednost. Pa ne samo to. Lahko se zgodi celo to, da je naprava, ki piše, tokovno preobremenjena in pregori. Seveda pa ne smemo pozabiti, da lahko pride do tega, da naprava, ki je priklopljena na vodilo, hote ali nehote zasede vodilo in ga ne sprosti. S tem pa učinkovito onesposobi vse ostale naprave, ki so priklopljene na isto vodilo.
Vse te težave je treba nasloviti in današnji standardi za vodila se bolj ali manj uspešno spoprijemajo z njimi.

2.2 PCI vodilo

PCI vodilo je računalniško vodilo, namenjeno priključevanju različnih perifernih naprav v računalnik. V takšni ali drugačni obliki je prisotno v veliki večini današnjih osebnih računalnikov, prenosnikov in strežnikov. Prva verzija standarda je izšla leta 1992, vendar so bile specifikacije zelo nepopolne, zato je slabo leto kasneje podjetje Intel izdalo specifikacije za drugo verzijo (PCI 2.0), ki so bila bistveno bolj dodelane. Kasneje so izdali še verzije PCI 2.1, PCI 2.2, PCI 2.3 ter končno verzijo PCI 3.0, ki jo štejemo za zadnjo verzijo standarda PCI. Kljub temu pa je standard doživel še nekaj prevetritev v obliki standarda PCI-X, ki je zmogljivejša različica vodila, namenjena za 64-bitne strežniške sisteme.

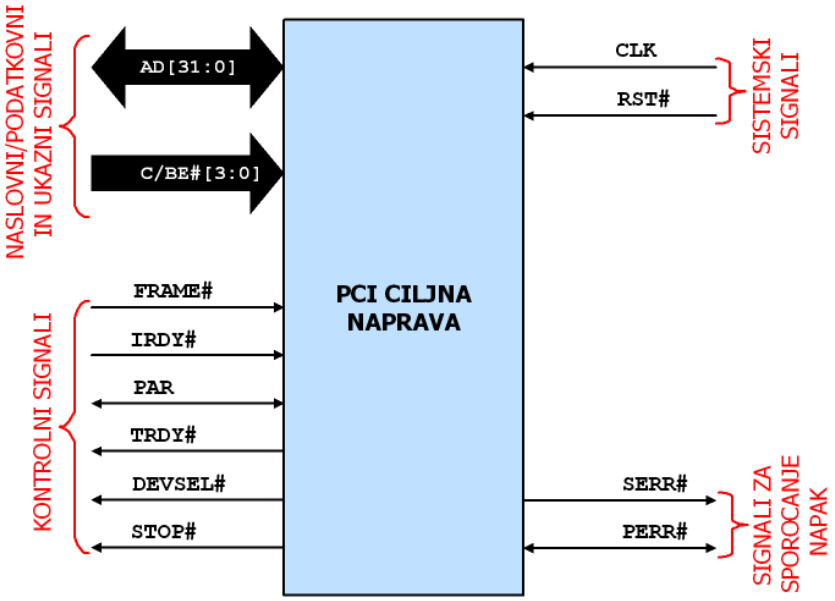
V modernih računalnikih je PCI namenjen priključevanju počasnejših naprav. Za hitre naprave (npr. grafične kartice, diskovni krmilniki, 10Gb Ethernet, Fibre Channel vmesnik, Infiniband vmesnik ipd.), se danes namesto PCI uporablja PCI Express, ki pa je kljub podobnosti v imenu popolnoma drugačen

standard. Ta počasi, a vztrajno, izpodriva svojega predhodnika.

Prenosu podatkov po PCI vodulu pravimo *transakcija*. V transakciji vedno nastopata *iniciator* (initiator), naprava, ki začne prenos ter *ciljna naprava* (target). Na eno PCI vodilo lahko teoretično priključimo 32 naprav. Zaradi električnih omejitev praktično ne najdemo PCI vodila, ki bi omogočalo priklop več kot 10 naprav. Na vsaki PCI napravi lahko implementiramo do osem t.i. *PCI funkcij* (PCI function). PCI vodila se lahko s pomočjo PCI *mostičkov* (PCI bridge) med seboj povezujejo v enoten sistem. To pomeni, da lahko naprave na različnih vodilih med seboj komunicirajo. V tak sistem lahko povežemo kar 256 vodil. Snovalci PCI standarda so s tem dosegli, da lahko v enem sistemu obstaja približno kar 25000 PCI funkcij. V okviru tega dela pa sem razvil le eno.

Pri implementaciji sem se omejil na standard PCI 2.2.

2.2.1 Signali



Slika 2.1: Signali, ki jih ciljna PCI naprava mora implementirati (vir:[1])

Najprej si poglejmo, katere signale PCI predvideva za ciljne naprave. To je

pomembno zato, da bi lažje razumeli delovanje PCI. Ker je PCI namenjen široki množici zelo različnih naprav, je mogoče pri implementaciji izpustiti signale, ki jih vezje ne uporablja. To močno poenostavi realizacijo vezja. Omejil se bom na razlago tistih signalov, ki sem jih tekom izdelave tega dela uporabil.

- Sistemska ura (**CLK**). Frekvenca ure se lahko giblje od 0 do 66MHz. Počasnejša kot je, daljše je lahko vodilo oziroma več naprav lahko priklopimo nanj. Vse operacije so sinhronizirane s prvo urino fronto.

- Naslovno oziroma podatkovno vodilo (**AD**[31:0]). V naslovnih ciklih so prisotni naslovi, sicer pa podatki.

- Ukazno vodilo oziroma bajtna maska (**C/BE#**[3:0]). Kadar je v teku naslovna faza prenosa, je na tem vodilu ukaz (glej tabelo 2.2.1), sicer pa bajtna maska s katero povemo po katerih linijah poteka prenos. To naredimo tako, da na ustrezna mesta C/BE# zapišemo ničle. Na primer: vrednost 1 bita 0 (t.j $C/BE\# = 0001b$) v C/BE# pomeni, da je prenos po AD[7:0] onemogočen. Če pa želimo, da prenos poteka po vseh linijah hkrati, potem postavimo $C/BE\# = 0000b$.

C/BE#	Ukaz
0000	Potrjevanje prekinitev
0001	Poseben cikel
0010	Branje z V/I naslovnega prostora
0011	Pisanje v V/I naslovni prostor
0100 0101	Rezervirano
0110	Branje iz pomnilniškega naslovnega prostora
0111	Pisanje v pomnilniški naslovni prostor
1000 1001	Rezervirano
1010	Branje konfiguracije
1011	Pisanje konfiguracije
1100	Večkratno branje pomnilnika
1101	Dvojni naslovni cikel
1110	Zaporedno branje
1111	"Piši-in-razveljavi"

Tabela 2.1: Vse možne vrednosti C/BE# in pripadajoči pomen

- Pariteta (**PAR**). Soda pariteta preko AD in C/BE# vodila. Pariteta zmeraj zaostaja za eno urino periodo za AD in C/BE#.

- Signal "prenos v teku" (**FRAME#**). Postavi ga iniciator, ko želi začeti transakcijo. To lahko stori le, če sta FRAME# in IRDY# neaktivna. Iniciator ga deaktivira pred zadnjo transakcijo in tako pove, da jo želi zaključiti.

- Signal "ciljna naprava pripravljena" (**TRDY#**). Ta signal aktivira naslovljena naprava, ko je pripravljena sprejeti trenuten podatek na vodilu (kadar je v teku pisalna transakcija), oziroma ima na vodilu pripravljen veljaven podatek za iniciatorja (v primeru pisalne transakcije).

- Signal "iniciator pripravljen" (**IRDY#**). Postavlja ga iniciator, ima pa podobno funkcijo kot TRDY#. Kadar sta oba postavljena ob prvi urini fronti, se vrši prenos.

- Signal "izbrana naprava" (**DEVSEL#**). S tem signalom ciljna naprava sporoči iniciatorju, da je dekodirala naslov in da je ona tista, ki je naslovljena.

- Signal, ki izbere napravo za konfiguracijo (**IDSEL**). Ko naprava še ni konfigurirana, torej nima svojega naslovenga prostora, jo konfiguracijski program izbere s tem signalom za namene konfiguracije. Tega signala si PCI naprave ne delijo.

- Sistemska ponastavitev (**RESET#**). Vse naprave naj gredo v začetno stanje - stanje, kot je bilo ob vklopu računalnika.

- Napaka paritete (**PERR#**). Signal za sporočanje napake tekom vseh transakcij razen v t.i. posebnih ciklih.

- Sistemska napaka (**SERR#**). Signal za sporočanje napak med posebnimi cikli oziroma drugih sistemskih napak z usodnimi posledicami.

Neuporabljeni signali pa so 64-bitne razširitve AD in C/BE#, PAR64, REQ64#, ACK64#, CLKRUN#, PME#, $3.3V_{aux}$, JTAG vmesnik ter INTA# do INTD#.

2.3 PCI konfiguracija

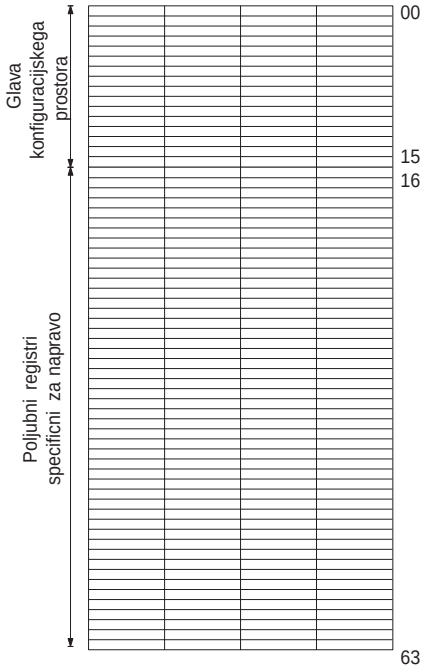
2.3.1 Plug and play

Ena lepih lastnosti PCI standarda je, da se naprave konfigurirajo brez posredovanja uporabnika. Temu v računalništvu rečemo tudi, da deluje v skladu s principom *plug and play*. Le-tega ne gre mešati z zbirko specifikacij podjetja Microsoft. Začetek konfiguracije začne že BIOS (Basic I/O System). PCI standard predpisuje uporabo registrov s posebnimi pomeni, ki imajo svoj naslovni prostor. Ta prostor se imenuje konfiguracijski prostor (configuration space). Poleg tega standard določa še V/I prostor (I/O space) in pomnilniški prostor (memory space). BIOS preda nadzor nad PCI vodilom operacijskemu sistemu, ko se ta zažene. Glede na stanja nekaterih registrov se določijo naslovni prostori, prekinitveni vektorji ipd. Za dostop do naslovnega prostora PCI uporablja pomnilniško preslikan V/I (Vhod/Izhod).
Ko so osnovne stvari nastavljene, se nadzor prepusti gonilniku. Ker ima PCI naprava v svojih registrih dovolj podatkov, lahko gonilnik brez posredovanja uporabnika ugotovi, ali je v sistemu prisotna naprava za katero je zadolžen.
Pridemo do tega, da je naprava najdena, in nastavljena ter da je zanjo zadolžen gonilnik naložen v pomnilniku; torej je pripravljena na uporabo - in vse to brez uporabnikovega posredovanja.

2.3.2 Konfiguracijski prostor (Configuration space)

Ta je velik 64 besed velikosti 32 bitov. Od tega je prva četrtina (16 besed) definirana kot glava (header), ki je obvezna, ostalo pa je na voljo razvijalcu naprave in ni nujno, da je sploh implementirano. Registri v glavi odražajo trenutno stanje naprave. Standard ne zahteva, da so vsi registri v glavi realizirani, vendar morajo nerealizirani vračati vrednost samih ničel.
Nekaj registrov je vseeno obveznih, ker sicer plug and play ne bi deloval. Ti registri so:

- **header type**. Specifikacija določa tri vrste tipe glave glede na funkcijo naprave.
  - Tip glave nič (Header type zero). Ta je namenjen perifernim napravam.
  - Tip glave ena. Namenjen mostu PCI na PCI.
  - Tip glave dva. Dodan v verziji 2.2 specifikacije, namenjen pa je mostu CardBus.



Slika 2.2: Configuration space kot množica registrov

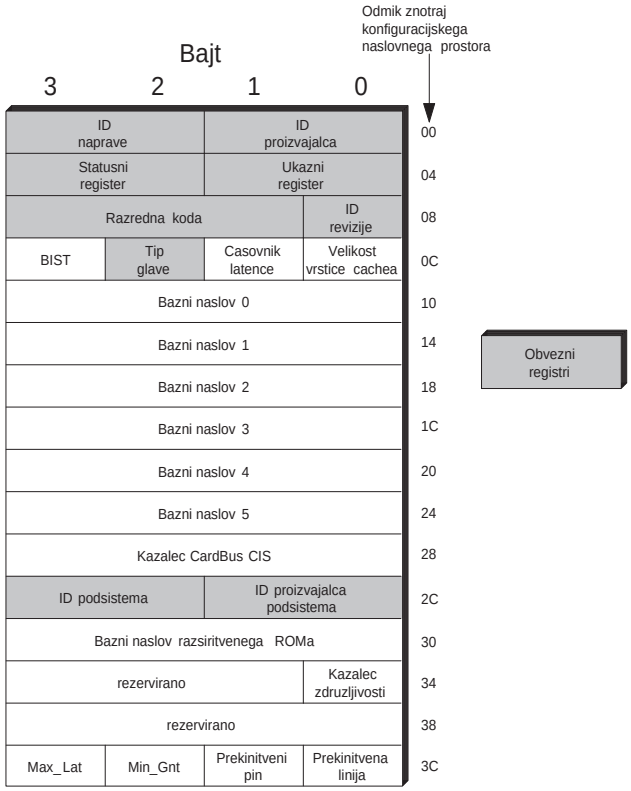
V okviru tega diplomskega dela nas zanima samo tip glave nič. Bit 7 tega registra pove, ali gre za napravo z eno ali več funkcijami. Primer PCI naprave z več funkcijami je zvočna kartica, ki ima poleg funkcije audio vmesnika tudi vmesnik za igralno palico.

- **ID proizvajalca (vendor ID)**. Organizacija, ki bdi nad razvojem in uporabo PCI standarda, vsakemu proizvajalcu dodeli neko številko, po kateri se vidi, za katerega proizvajalca gre.

- **ID naprave (device ID)**. Običajno proizvajalci izdelujejo več kot samo eno napravo. Z vrednostjo tega registra povejo za katero napravo, znotraj njihovega nabora naprave, pravzaprav gre.

- **ID revizije (revision ID)**. Tega nastavi proizvajalec naprave, saj lahko na podlagi vrednosti tega registra naloži pravilen gonilnik, če uporablja gonilnike specifične za verzijo naprave.

- **razredna koda (class code)**. Periferna oprema je razdeljena v kategorije, glede na namembnost naprave. S tem lahko dosežemo, da neka naprava uporablja nek generični gonilnik. Lep primer bi bila mrežna



Slika 2.3: Registri v glavi konfiguracijskega naslovnega prostora

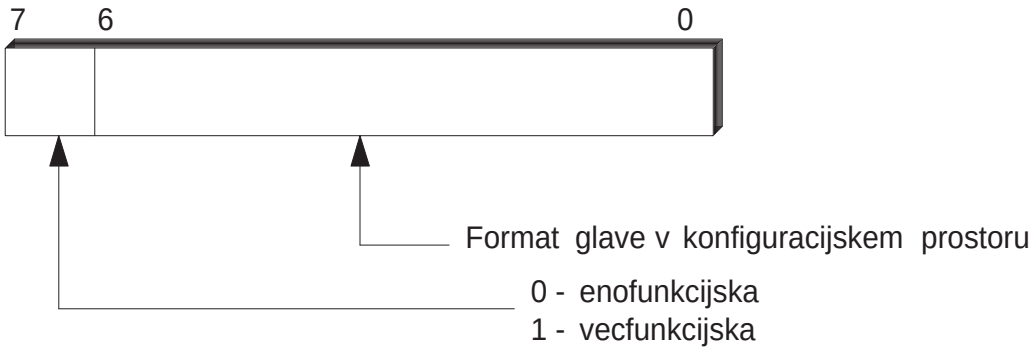
kartica izdelana skladno z ne2000, ki jo je razvilo podjetje Novell (prvotno za vodilo ISA šele kasneje za PCI) in so jo različni izdelovalci radi klonirali.

- **ID proizvajalca podsistema (subsystem vendor ID)**

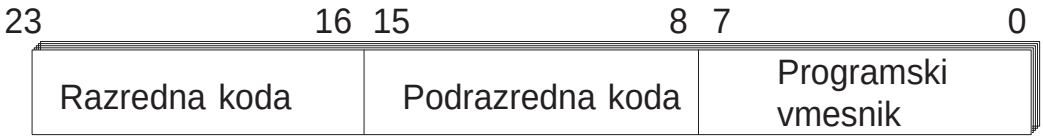
- **ID naprave podsistema (subsystem device ID)**.

- **ukazni register (command register)**. Omogoča osnovni nadzor nad možnostjo odziva oz. generiranjem PCI dostopov.

- **statusni register (status register)**. Odraža stanje naprave. Posebnost tega registra je, da mora biti implementiran tako, da s pisanjem enice na določen bit, ta bit resetiramo na nič. Programsko torej sploh ne moremo pisati vanj. Lahko ga beremo ali resetiramo.



Slika 2.4: Tip glave

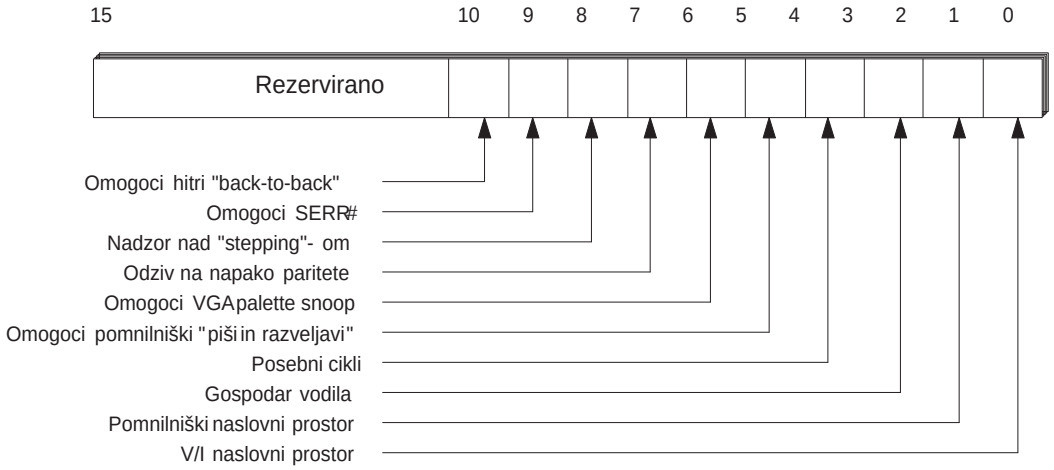


Slika 2.5: Pomen posameznih bajtov v registru razredne kode

Poleg obveznih registrov sem uporabil še bazni register BAR0 (Base Address Register). Naprava ve, da je naslovljena, kadar dobi na naslovno vodilo naslov, ki se sklada s podatkom v tem registru. Na sliki 2.3 vidimo, da je takih registrov šest, zato lahko naprava uporablja več različnih naslovnih prostorov, vendar največ šest. Ti naslovni prostori so lahko v pomnilniškem ali V/I naslovnem prostoru. Priporoča se, da se vedno, ko je to mogoče, uporablja pomnilniški prostor. Le-ta je bistveno večji, medtem ko je V/I prostor manjši in že zelo zaseden s strani drugih protokolov. Poleg tega lahko po specifikaciji V/I bazni register zajema največ 256 V/I lokacij, pomnilniški pa te omejitve nima.
Zaradi bolj učinkovitega dekodiranja naslova se v primeru uporabe pomnilniškega baznega registra priporoča, da je velik vsaj 4KB. Za naprave, ki rabijo manjši prostor, se kjub temu priporoča uporaba 4KB pomnilniškega prostora.

2.3.3 Kako naprava dobi svoj naslovni prostor

Program, čigar naloga je konfiguriranje naprave (naj bo to BIOS ali pa jedro OS), mora za vsako funkcijo znotraj PCI naprave ugotoviti:

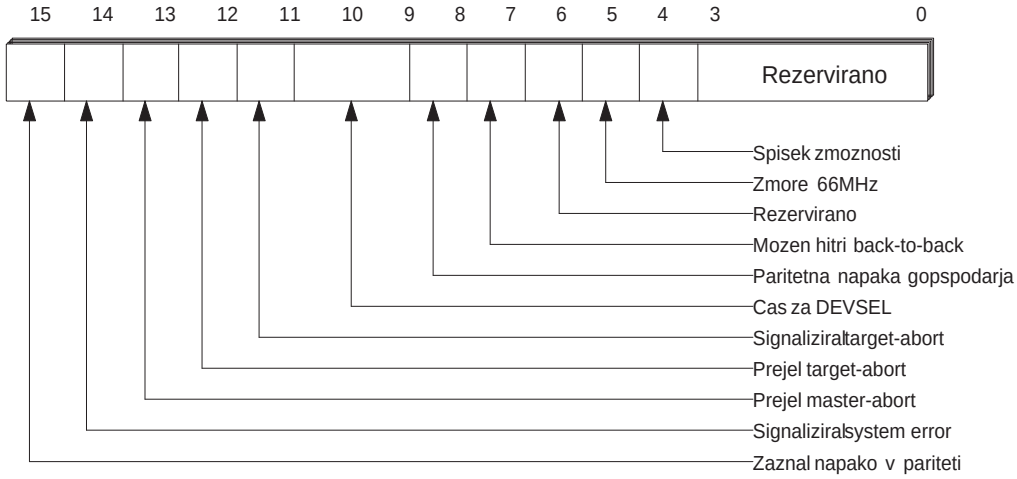


Slika 2.6: Pomen posameznih bitov v ukaznem registru

1. Ali je bazni register implementiran?
2. Ali gre za pomnilniški ali V/I naslovni dekodirnik?
3. So bazni registri 32 ali 64 bitni?
4. V primeru pomnilniškega dekodirnika je narava podatov v naslovnem prostoru primerna za vnaprejšnje branje (prefetchable memory)?
5. Koliko naslovnega prostora potrebuje?

Vse te informacije dobi tako, da v vsak bazni register zapiše same enice in nato ta register prebere. Če prebere same ničle, to pomeni, da register ni implementiran. V nasprotnem primeru pa začne pregledovati vrnjeno vrednost od najmanj pomembnega bita baznega naslova (base address) baznega registra navzgor, dokler ne naleti na bit, postavljen na ena. To je prvi bit fiksnega dela naslova v tem naslovnem prostoru. Na sliki 2.8 se vidi, da je začetek baznega naslova bit 4 za pomnilniški naslovni prostor, za V/I pa je s slike 2.9 razvidno da je to bit 2. Konfiguracijski program nato zapiše njegov naslovni prostor v fiksni del naslova.

Primer naprave s pomnilniškim naslovnim prostorom:
Vzemimo primer, ko konfiguracijski program v bazni register zapiše $FFFFFFFFh$ in prebere $FFF00000h$. Že dejstvo, da je rezultat različen od samih ničel, mu pove, da je bazni register implementiran in da mu je potrebno določiti vrednost. Posamezni biti prebrane vrednosti imajo naslednji pomen:



Slika 2.7: Pomen posameznih bitov v statusnem registru

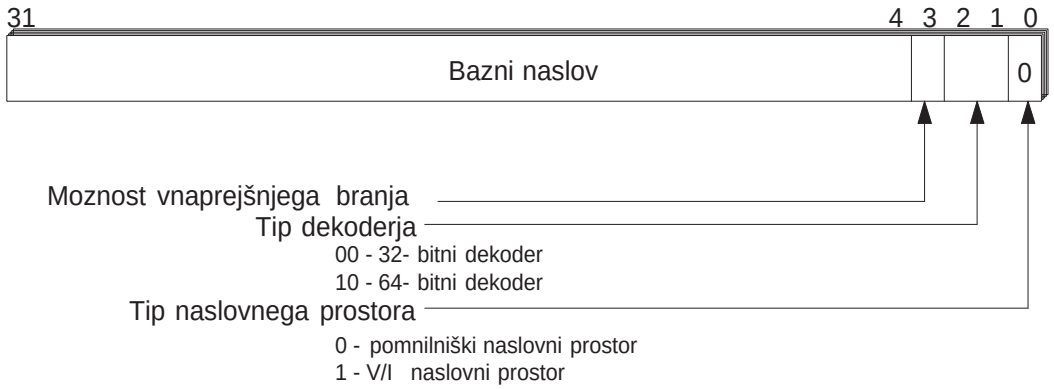
- bit 0 = 0 pove, da gre za pomnilniško dekodiranje
- bita 1,2 = 00b povesta, da gre za 32-bitni dekodirnik
- bit 3 = 0 pove, da je vnaprejšnje branje onemogočeno
- bit 20 je prvi bit, ki je postavljen na ena, znotraj polja za bazni naslov, kar pomeni, da gre za dekodirnik za 1MB pomnilniškega prostora.

Nato program zapiše 32-bitni naslov v bazni register (v našem primeru se zapišejo samo biti od 31 do 20). Tak način dodeljevanja implicitno pomeni, da so naslovni prostori poravnani.

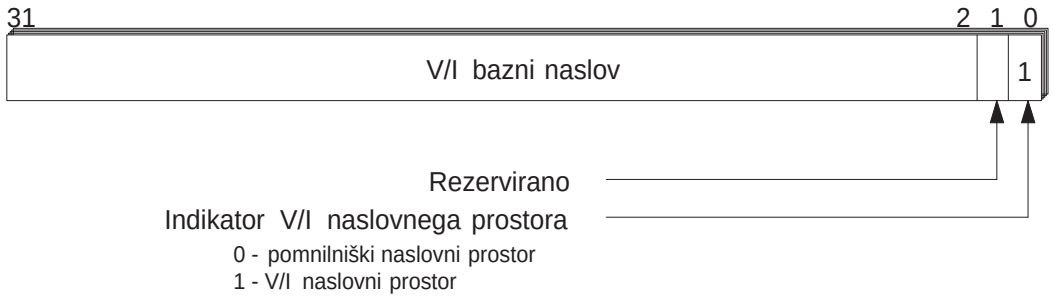
2.3.4 Bralni prenos v pomnilniško preslikanem prostoru

Prenos začne vedno iniciator (*initiator*), ker hoče komunicirati s ciljno napravo (*target*). Kadarkoli želi začeti prenos, mora počakati, da je vodilo prosto. To naredi tako, da opazuje signala FRAME# in IRDY# (oba morata biti deaktivirana - t.j. morata biti v visokem stanju).

prva urina perioda Ko iniciator zazna, da je vodilo prosto (FRAME# in IRDY# deaktivirana), začne prenos ob prvi fronti prve urine periode. Na vodilo postavi vrednosti AD (naslov) in C/BE# (ukaz) ter aktivira signal FRAME#, s čimer ostalim napravam na vodilu pove, da se je začel prenos in da so vrednosti AD in C/BE# veljavne.



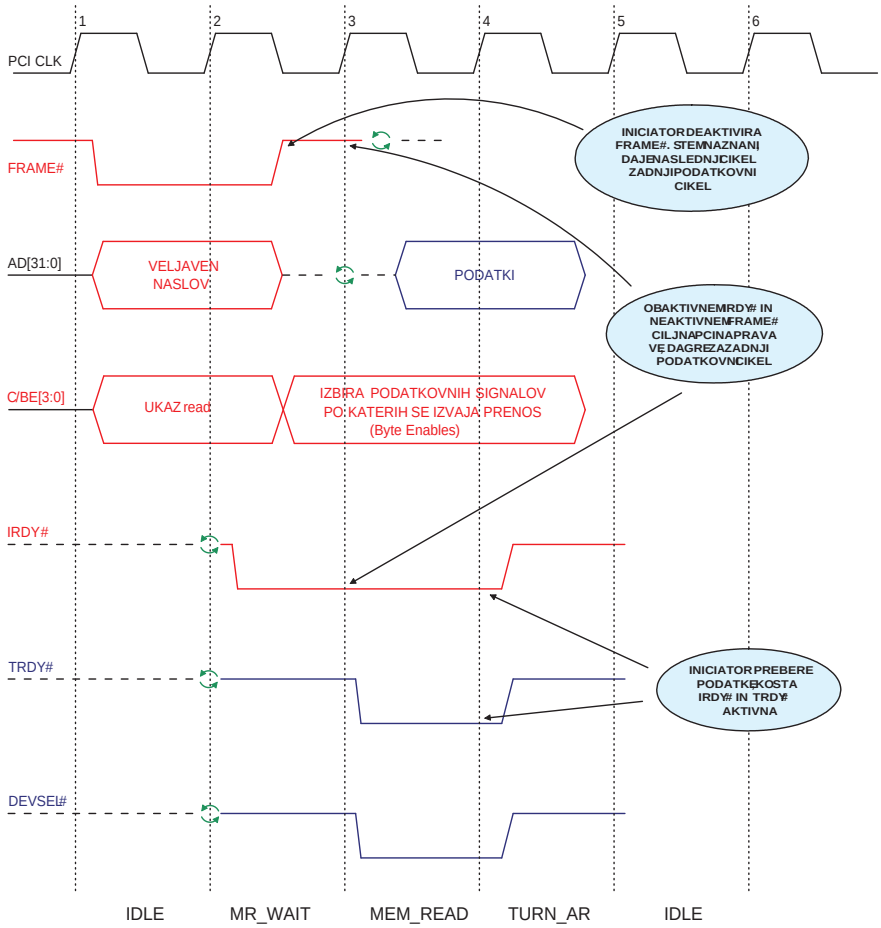
Slika 2.8: Pomen posameznih bitov v pomnilniškem baznem registru



Slika 2.9: Pomen posameznih bitov v V/I baznem registru

druga urina perioda Vse ostale naprave na vodilu zaznajo naslov, ukaz in FRAME# in preidejo v fazo dekodiranja naslova, da bi ugotovile, katera je ciljna naprava. Iniciator aktivira signal IRDY#, s tem pove, da je pripravljen na prvi sprejem. Hkrati še deaktivira FRAME# in s tem nakaže, da je to zadnji prenos. C/BE# preneha biti ukaz - namesto tega iniciator pošlje bajtno masko (*bytes enable*). Ciljna naprava je prebrala naslov, zato lahko v tem ciklu iniciator postavi vodilo AD v stanje visoke impedance, da se spremeni lastništvo vodila AD. Zaradi tega temu ciklu rečemo tudi *obračalni cikel* (turn-around cycle).

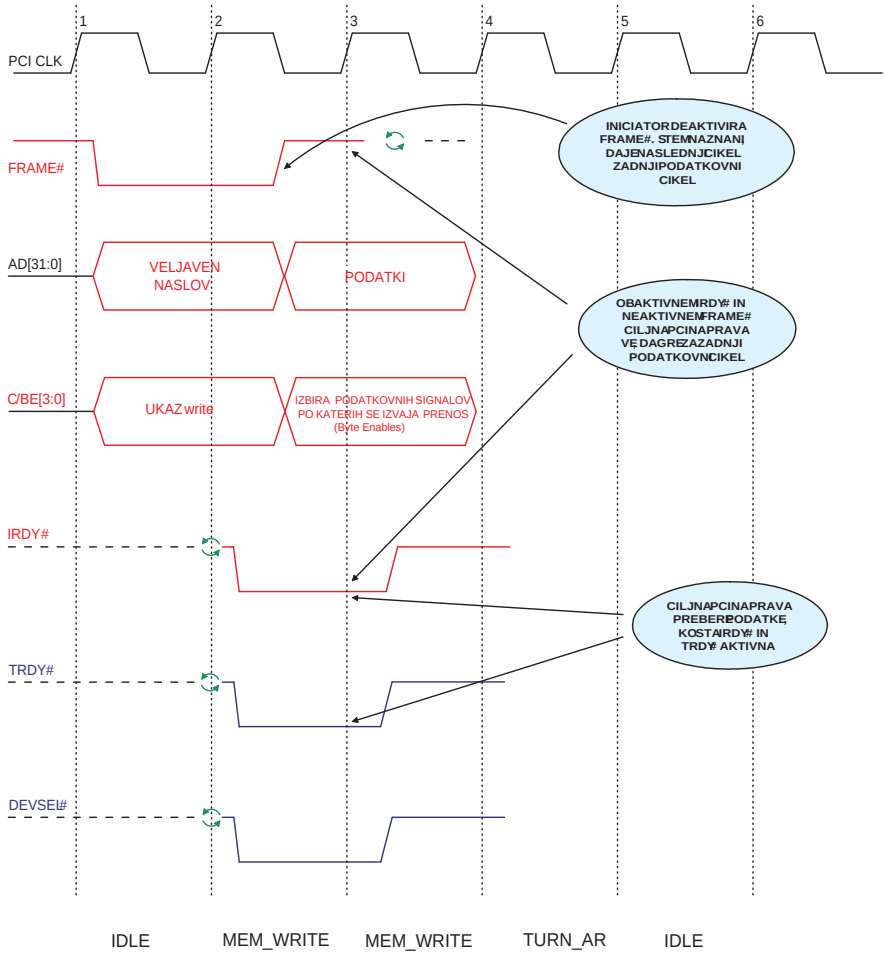
tretja urina perioda Ciljna naprava aktivira signal DEVSEL# in s tem sporoči iniciatorju, da je dekodirala naslov in da bo sodelovala v prenosu. Poleg tega že postavi prvi podatek na AD vodilo in TRDY# označi, da je podatek pripravljen.



Slika 2.10: Časovni diagram bralnega prenosa

četrta urina perioda Ob prvi urini fronti četrte periode iniciator in ciljna naprava zaznata aktivirana signala IRDY# in TRDY#. Iniciator prebere podatek z vodila, saj mu TRDY# pove, da je podatek veljaven. Ciljna naprava zazna deaktiviran signal FRAME#, kar pomeni, da je šlo za zadnji prenos. S tem je branje uspešno zaključeno. Iniciator umakne signale IRDY# in C/BE#, ciljna naprava pa TRDY# in DEVSEL#. Tako se lahko vodilo v naslednji (peti) urini periodi vrne v stanje čakanja.

2.3.5 Pisalni prenos v pomnilniško preslikanem prostoru



Slika 2.11: Časovni diagram pisalnega prenosa

prva urina perioda Ko iniciator zazna, da je vodilo prosto (FRAME# in IRDY# deaktivirana), začne prenos ob prvi fronti prve urine periode. Na vodilo postavi vrednosti AD (naslov) in C/BE# (ukaz), ter postavi signal FRAME#, s čimer ostalim napravam na vodilu pove, da se je začel prenos in da so vrednosti AD in C/BE# veljavne.

druga urina perioda Vse ostale naprave na vodilu zaznajo naslov, ukaz in FRAME# in preidejo v fazo dekodiranja naslova, da bi ugotovile, katera

je ciljna naprava. Initiator postavi signal IRDY#, s tem pove, da je prvi podatek pripravljen na vodilu. Hkrati še deaktivira FRAME# in s tem nakaže, da je to zadnji prenos. C/BE# preneha biti ukaz - namesto tega iniciator pošlje bajtno masko. Ciljna naprava postavi signal DEVSEL# in s tem sporoči iniciatorju, da je dekodirala naslov in da bo sodelovala v prenosu. Hkrati še postavi signal TRDY#, ker je pripravljena na sprejem podatka.

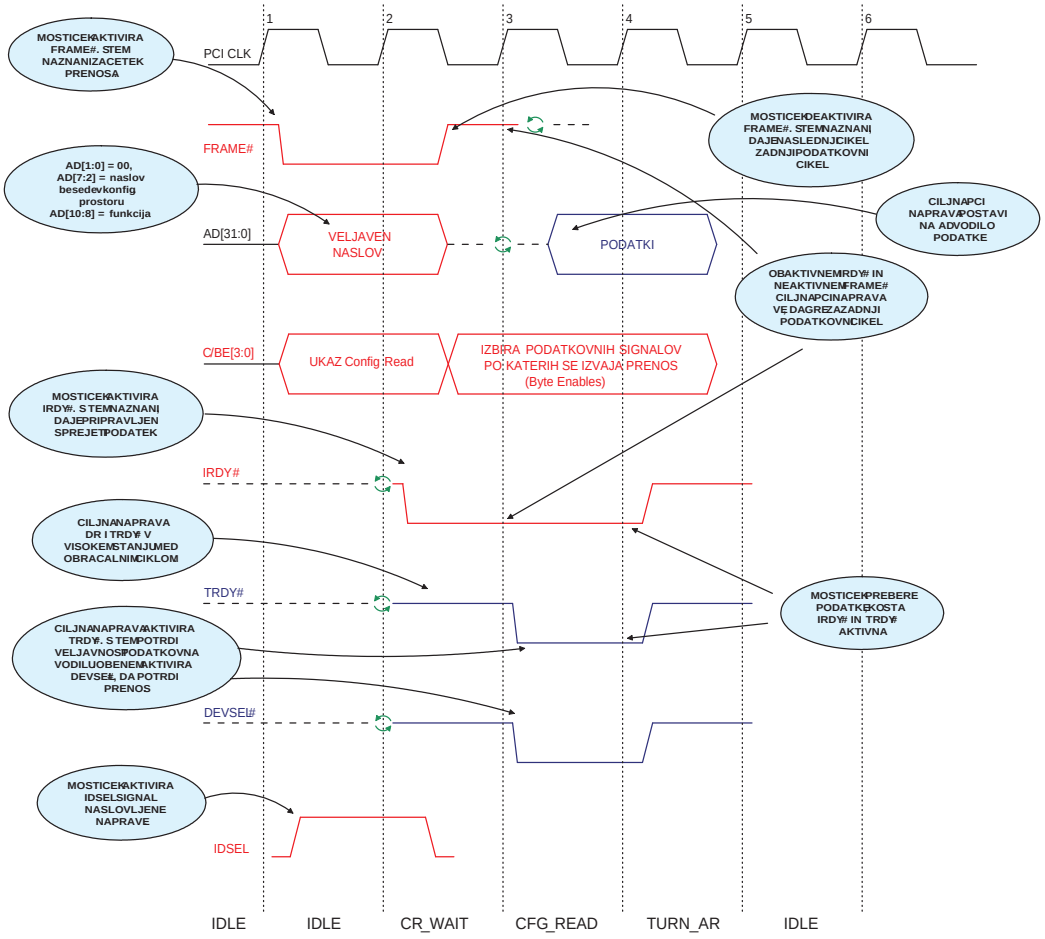
tretja urina perioda Iniciator zazna signal DEVSEL# in tako ve, da je ciljna naprava pripravljena za sprejem. Ciljna naprava zazna signal IRDY#, zato sprejme veljavne podatke z AD vodila. Iniciator in ciljna naprava vesta, da se je prenos enega podatka uspešno zaključil, ker sta bila aktivna signala IRDY# in TRDY#. Ker je signal FRAME# deaktiviran, ciljna naprava ve, da je šlo za zadnji prenos, zato deaktivira signala TRDY# in DEVSEL#, iniciator pa se umakne z vodil AD in C/BE# ter deaktivira signal IRDY#. Ob prvi fronti naslednje urine periode se vodilo vrne v stanje čakanja.

Takoj lahko vidimo, da smo za pisanje porabili eno urino periodo manj, ker ni bilo potrebno menjati lastništva nad vodilom, kar je posledica tega, da pri pisalnih prenosih obračalni cikel ni potreben.

2.3.6 Bralni konfiguracijski prenos

prva urina perioda Ko iniciator zazna, da je vodilo prosto (FRAME# in IRDY# deaktivirana), začne prenos ob prvi fronti prve urine periode. Na vodilo postavi vrednosti AD (naslov znotraj konfiguracijskega prostora) in C/BE# (ukaz), ter aktivira signala FRAME#, s čimer ostalim napravam na vodilu pove, da se je začel prenos in da so vrednosti AD in C/BE# veljavne. Ker se dekoder na naslov, ki je na AD, ne odziva, bodisi, ker še ni nastavljen, ali pa, ker se odziva samo na pomnilniške oziroma V/I dostope, mora dekoder napravi na drug način sporočiti, da se transakcija nanaša nanjo. To naredi s signalom IDSEL.

druga urina perioda Vse ostale naprave na vodilu zaznajo naslov, ukaz in FRAME#. Ker je ukaz konfiguracijski prenos, naprave čakajo na signal IDSEL. Initiator aktivira signal IRDY#, s tem pove, da je pripravljen na prvi sprejem. Hkrati še deaktivira FRAME# in s tem nakaže, da je to zadnji prenos. C/BE# preneha biti ukaz - namesto tega iniciator pošlje bajtno masko (*bytes enable*). Ciljna naprava je zaznala signal IDSEL



Slika 2.12: Časovni diagram bralnega konfiguracijskega prenosa

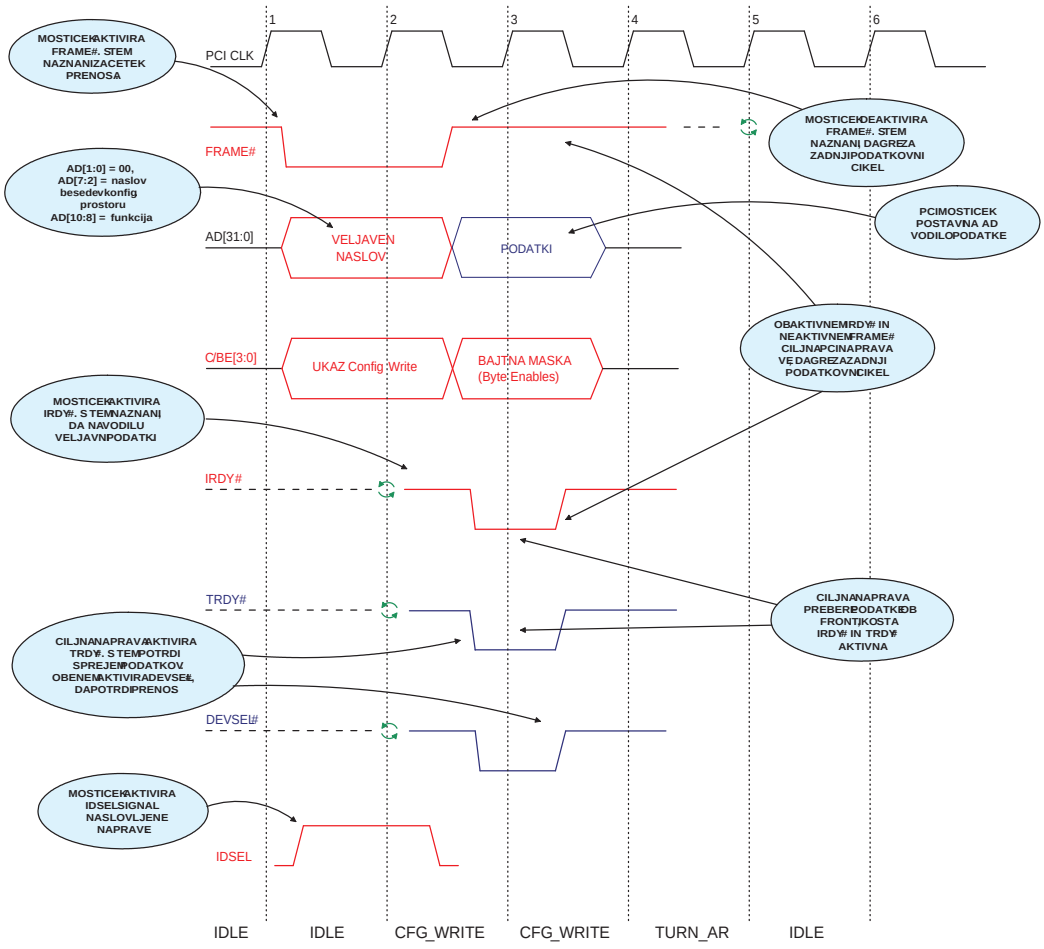
ter prebrala naslov, zato lahko v tem ciklu iniciator postavi vodilo AD v stanje visoke impedance, da se spremeni lastništvo vodila AD.

tretja urina perioda Ciljna naprava aktivira signal DEVSEL# in s tem sporoči iniciatorju, da je zaznala signal IDSEL ter dekodirala naslov in da bo sodelovala v prenosu. Poleg tega že postavi prvi podatek na AD vodilo in s TRDY# označi, da je podatek pripravljen.

četrta urina perioda Ob prvi urini fronti četrte periode iniciator in ciljna naprava zaznata aktivirana signala IRDY# in TRDY#. Iniciator prebere podatek z vodila, saj mu TRDY# pove, da je podatek veljaven.

Ciljna naprava zazna deaktiviran signal FRAME#, kar pomeni, da je šlo za zadnji prenos. S tem je branje uspešno zaključeno. Iniciator umakne signale IRDY# in C/BE#, ciljna naprava pa TRDY# in DEVSEL#. Tako se lahko vodilo v naslednji (peti) urini periodi vrne v stanje čakanja.

2.3.7 Pisalni konfiguracijski prenos



Slika 2.13: Časovni diagram pisalnega konfiguracijskega prenosa

prva urina perioda Ko iniciator zazna, da je vodilo prosto (FRAME# in IRDY# deaktivirana), začne prenos ob prvi fronti prve urine periode.

Na vodilo postavi vrednosti AD (naslov) in C/BE# (ukaz) ter postavi signal FRAME#, s čimer ostalim napravam na vodilu pove, da se je začel prenos in da so vrednosti AD in C/BE# veljavne. Ciljni napravi pošlje še signal IDSEL.

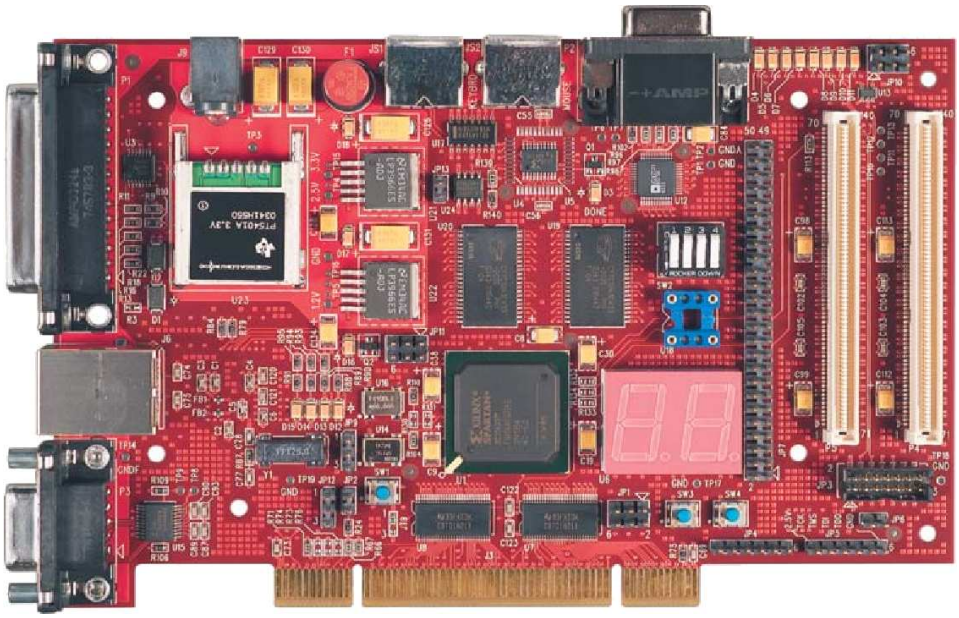
druga urina perioda Vse ostale naprave na vodilu zaznajo naslov, ukaz in FRAME#. Ker ne zaznajo signala IDSEL, vedo, da se ta transakcija njih ne tiče. Initiator postavi signal IRDY#, s tem pove, da je prvi podatek pripravljen na vodilu. Hkrati še deaktivira FRAME# in s tem nakaže, da je to zadnji prenos. C/BE# preneha biti ukaz - namesto tega iniciator pošlje bajtno masko. Ciljna naprava postavi signal DEVSEL# in s tem sporoči iniciatorju, da je dekodirala naslov in da bo sodelovala v prenosu. Hkrati še postavi signal TRDY#, ker je pripravljena na sprejem podatka.

tretja urina perioda Iniciator zazna signal DEVSEL# in tako ve, da je ciljna naprava pripravljena za sprejem. Ciljna naprava zazna signal IRDY#, zato sprejme veljavne podatke z AD vodila. Iniciator in ciljna naprava vesta, da se je prenos enega podatka uspešno zaključil, ker sta bila aktivna signala IRDY# in TRDY#. Ker je signal FRAME# deaktiviran, ciljna naprava ve, da je šlo za zadnji prenos, zato deaktivira signala TRDY# in DEVSEL#, iniciator pa se umakne z vodil AD in C/BE# ter deaktivira signal IRDY#. Ob prvi fronti naslednje urine periode se vodilo vrne v stanje čakanja.

Opazimo, da sta si konfiguracijski in pomnilniški transakciji podobni, tako pri branju, kot tudi pri pisanju. Pravzaprav je bistvena razlika le v vlogi signala IDSEL in v tem, kako ciljna naprava dekodira naslov.

2.4 Implementacija PCI vmesnika

2.4.1 Strojna oprema



Slika 2.14: Razvojna plošča Xilinx Spartan-3 400

Pri implementaciji sem uporabil razvojno ploščo Xilinx Spartan-3 400 Evaluation Kit, kar mi je omogočilo, da sem lahko strojno opremo elegantno razvijal v VHDL. Glavne lastnosti te plošče so:

- Xilinx XC3S1500-FG456 Spartan-3 FPGA (Field-Programmable Gate Array) čip
- Xilinx platform FLASH PROM (Programmable Read-Only Memory) za konfiguracijo
- 2 oscilatorja
- JTAG (Joint Test Action Group) vmesnik za konfiguracijo
- 2 AvBus razširitvena konektorja
- 1, 50-pinski vmesnik za lažji V/I dostop

- univerzalni 32-bitni PCI robni konektor
- 10/100 Ethernet port
- DB15 in video DAC (Digital-to-Analog Converter)
- vmesnik RS-232
- vmesnik PS/2
- 1MB SRAMa (Static RAM)
- 256kb serijskega EEPROM (Electrically Erasable PROM)
- 4 drsna DIP (Dual Inline Package) stikala
- 2 gumba na stik
- 8 samostojnih LED (Light-Emitting Diode) diod
- Dvoznakovni 7-segmentni LED prikazovalnik

Kartica, ki jo vidimo na sliki, 2.14, nam le izpostavi robni konektor. V VHDL dobimo surove signale, s katerimi potem operiramo in razvijamo logično vezje.

2.4.2 Načrtovanje

Naloga je bila realizirati napravo, ki se bo nastavila v skladu s PCI protokolom in bi bilo z njo mogoče komunicirati iz operacijskega sistema. Zraven sem predpostavil še:

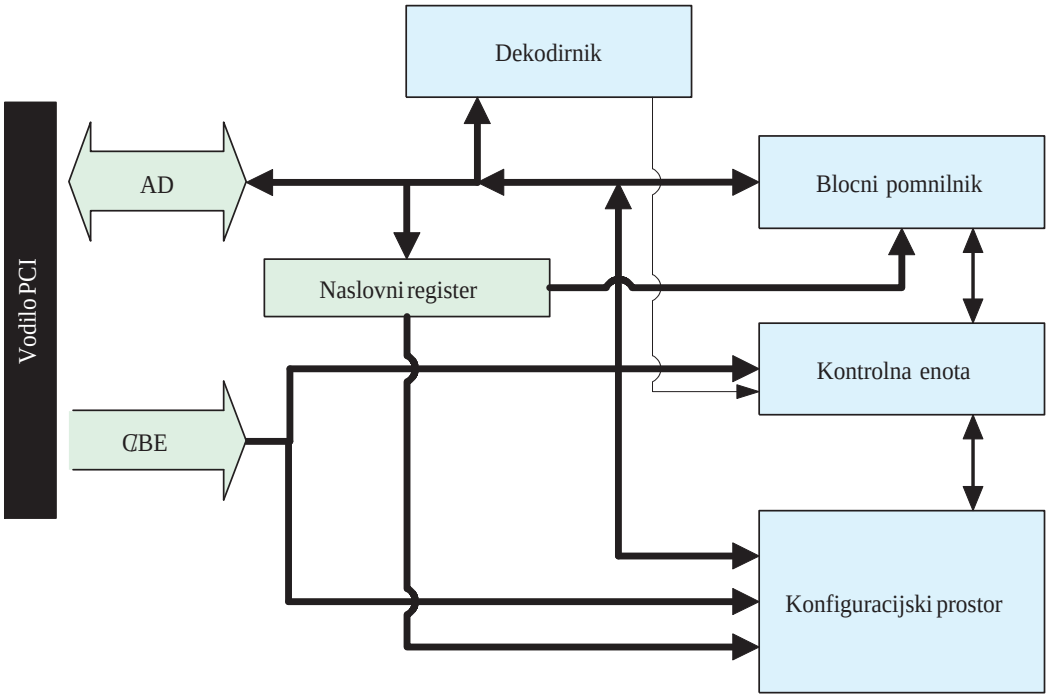
- naprava ne bo nikoli iniciator, torej bo vedno ciljna naprava,
- imela bo samo eno PCI funkcijo,
- imela bo en sam naslovni prostor in sicer pomnilniškega, ki bo velik 4KB,
- ne bo podpirala eksplozijskih prenosov,
- bo delovala z osnovno frekvenco 33MHz.

Preden sem se lotil programiranja v VHDL, sem problem razbil na več manjših podproblemov. Hkrati sem želel narediti vezje karseda modularno, saj le tako lahko omogočim nekomu drugemu, da enostavno uporabi moje delo v lastne namene.
Vezje sem logično razbil na štiri med seboj povezane enote:

Signal	PCI Pin #	PCI Pin #	Signal
TRST#	A1	B1	-12V
+12V	A2	B2	TCK
TMS	A3	B3	GND
TDI	A4	B4	TDO
+5V	A5	B5	+5V
INTA#	A6	B6	+5V
INTC#	A7	B7	INTB#
+5V	A8	B8	INTD#
N/C	A9	B9	PRSNT1#
VIO	A10	B10	N/C
N/C	A11	B11	PRSNT2#
reža	A12	B12	*reža*
reža	A13	B13	*reža*
N/C	A14	B14	N/C
RST#	A15	B15	GND
VIO	A16	B16	CLK
GNT#	A17	B17	GND
GND	A18	B18	REQ#
N/C	A19	B19	VIO
AD[30]	A20	B20	AD[31]
+3.3V	A21	B21	AD[29]
AD[28]	A22	B22	GND
AD[26]	A23	B23	AD[27]
GND	A24	B24	AD[25]
AD[24]	A25	B25	+3.3V
IDSEL	A26	B26	C/BE[3]#
+3.3V	A27	B27	AD[23]
AD[22]	A28	B28	GND
AD[20]	A29	B29	AD[21]
GND	A30	B30	AD[19]
AD[18]	A31	B31	+3.3V
AD[16]	A32	B32	AD[17]
+3.3V	A33	B33	C/BE[2]#
FRAME#	A34	B34	GND
GND	A35	B35	IRDY#
TRDY#	A36	B36	+3.3V
GND	A37	B37	DEVSEL#
STOP#	A38	B38	GND
+3.3V	A39	B39	LOCK#
SDONE	A40	B40	PERR#
SBO#	A41	B41	+3.3V
GND	A42	B42	SERR#
PAR	A43	B43	+3.3V
AD[15]	A44	B44	C/BE[1]#
+3.3V	A45	B45	AD[14]
AD[13]	A46	B46	GND
AD[11]	A47	B47	AD[12]
GND	A48	B48	AD[10]
AD[9]	A49	B49	M66EN
reža	A50	B50	*reža*
reža	A51	B51	*reža*
C/BE[0]#	A52	B52	AD[8]
+3.3V	A53	B53	AD[7]
AD[6]	A54	B54	+3.3V
AD[4]	A55	B55	AD[5]
GND	A56	B56	AD[3]
AD[2]	A57	B57	GND
AD[0]	A58	B58	AD[1]
VIO	A59	B59	VIO
REQ64#	A60	B60	ACK64#
+5V	A61	B61	+5V
+5V	A62	B62	+5V

Tabela 2.2: Pini na kartici in kako so vezani na PCI vodilo

- konfiguracijski prostor,
- dekodirnik,
- kontrolno enoto,
- in uporabniško vezje.



Slika 2.15: Blok shema naše PCI naprave

2.4.3 Konfiguracijski prostor

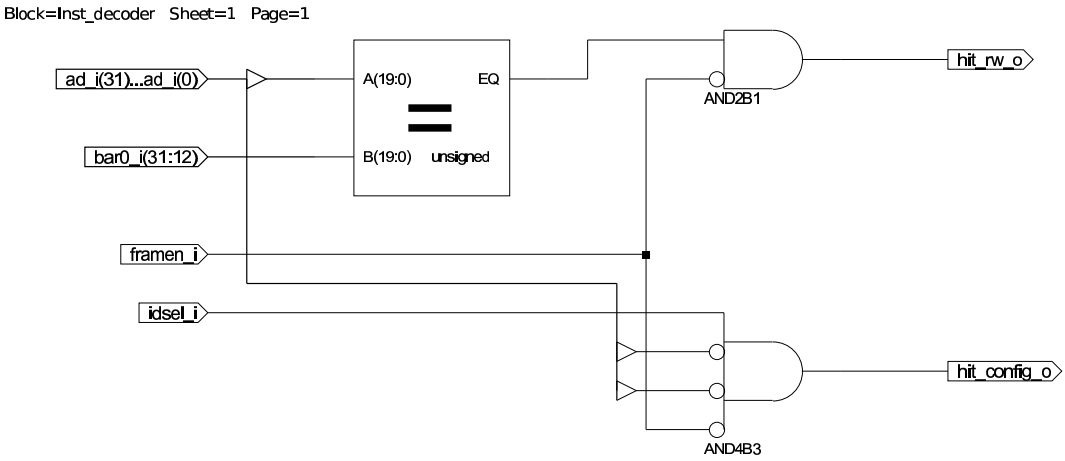
Konfiguracijski prostor je skupek registrov, s katerimi se operira preko enotnega naslovnega prostora. Pomen posameznih registrov vidimo na sliki 2.3. Glede na predpostavke želene naprave sem moral poleg obveznih registrov realizirati še en bazni register. Znotraj konfiguracijskega prostora so lokacije 00h (Vendor ID, Device ID), 02h (Revision ID, Class Code) in 0Ah (Subsystem Vendor ID, Subsystem ID) namenjene samo za branje. Njihove vrednosti sem si izmislil, moral pa sem jih upoštevati pri pisanju gonilnika. Privzete vrednosti za bralno pisalne registre so:

statusni register = $0000h$. Predvsem so pomembne vrednosti bitov 5, 9 in 10. S slike 2.7 je razvidno, da s tem povem, da gre za napravo, ki deluje na največ 33MHz ter da lahko dekoder dekodira naslov v eni urini periodi.

ukazni register = $0000h$. Vsi biti, ki jih opisuje ta register, in jih vidimo na sliki 2.6 so postavljeni na nič. Pomembneje, vrednosti zadnjih treh bitov so nič, kar PCI napravi onemogoča, da bi se odzivala na podatkovne dostope. S tako konfiguracijo bitov se lahko odziva le na konfiguracijske dostope.

bazni register = $FFFFF000h$. Kar pomeni 4KB naslovni prostor. Vrednost nič zadnjih štirih bitov (slika 2.8) pomeni, da naprava ne podpira vnaprejšnega branja, da gre za 32-bitni dekoder in da ta bazni register opisuje pomnilniški naslovni prostor.

2.4.4 Dekodirnik

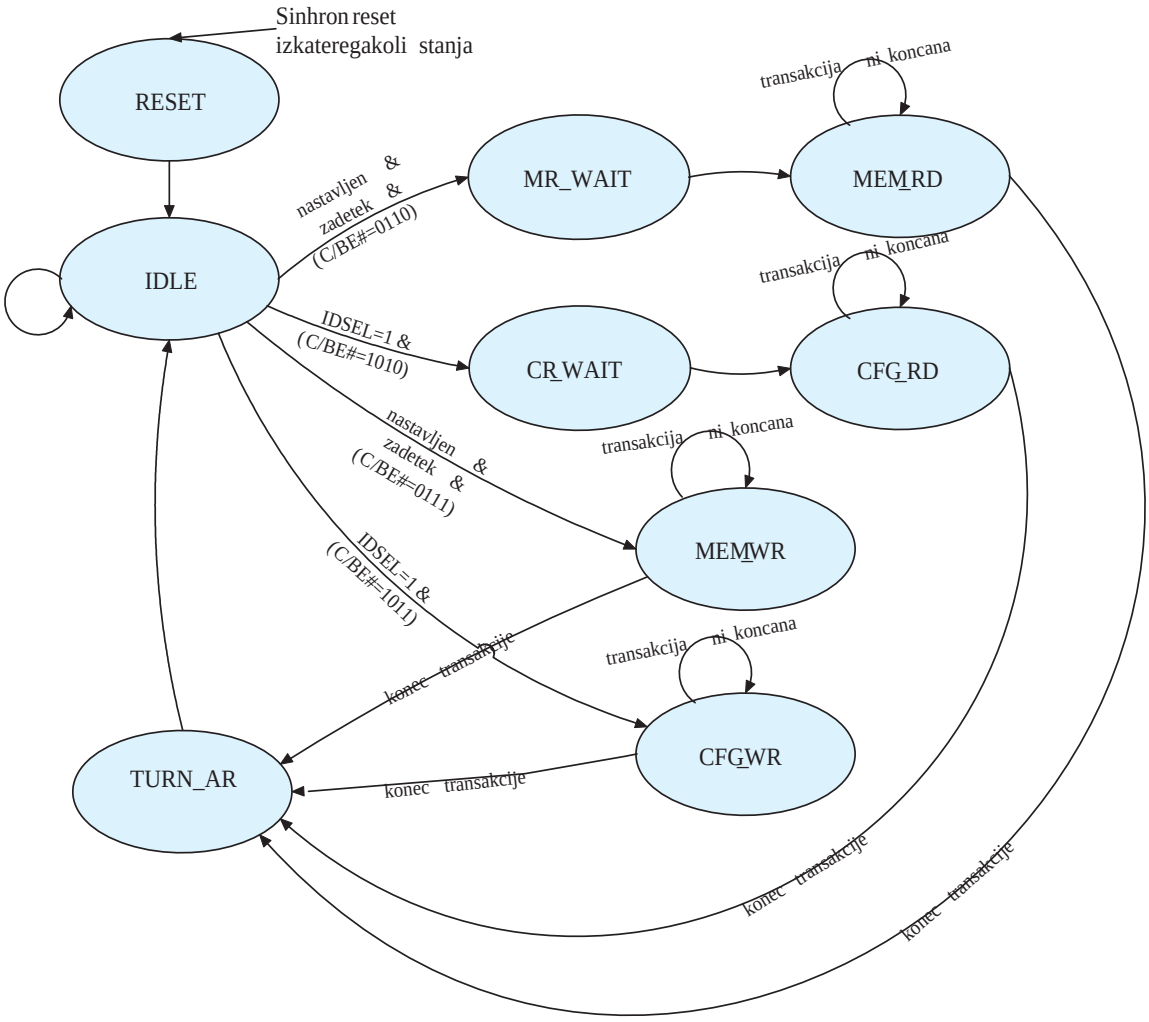


Slika 2.16: Dekodirnik naslova

Dekoder je enostavni primerjalnik, ki v primeru, da zazna signal FRAME#, primerja vrednost vodila AD z baznim registrom iz konfiguracijskega prostora (seveda le relevantne bite - v našem primeru od bita 31 do 23). Če se vrednosti ujemata preko signala HIT sporoči kontrolni enoti, da je na vodilu naslov, ki naslavlja to napravo. Prav tako aktivira signal HIT, če zazna signal IDSEL, ki je namenjen kofiguraciji naprave, ko ta še nima veljavnega naslova v baznem

registru. Ker dekoder nima logike, ki bi spremljala, ali je naprava že konfigurirana, aktivira signal HIT tudi, če pride do zadetka začetne vrednosti v baznem registru. To bi lahko bil problem, vendar sem ta del logike premaknil v kontrolno enoto, da bi bil dekoder kar se da enostaven.

2.4.5 Kontrolna enota



Slika 2.17: Shema končnega avtomata za kontrolno enoto

Kontrolna enota je realizirana kot Mooreov končni avtomat. Stanja bi lahko kategoriziral na:

- Režijska stanja:

  RESET Začetno stanje avtomata. V tem stanju kontrolna enota postavi privzete vrednosti različnih registrov in kontrolnih bitov. V to stanje avtomat zaide, kadar zazna signal RESET#. Reset bi moral biti asinhron, vendar sem zaradi narave realizacije v FPGA rajši naredil sinhronega.

  IDLE Iz stanja RESET gre avtomat takoj v naslednji urini periodi v stanje IDLE. V tem stanju kontrolna enota postavi signale in vodila v stanje visoke impedance. Hkrati "posluša"vodilo AD in čaka na veljaven naslov oziroma na signal IDSEL, ki pomeni, da želi gospodar vodil opraviti konfiguracijski prenos.

  TURN_AR To je zadnje stanje preden se avtomat vrne v stanje IDLE po bralnih oziroma pisalnih ciklih. Namen tega stanja je, da dvigne (pre-charge) napetost na AD in C/BE# vodilih, da vodilo v času mirovanja ne bi osciliralo okoli preklopnih napetosti. To je potrebno zaradi električnih lastnosti CMOS vezij. Torej v tem stanju kontrolna enota postavi vse linije vodil AD in C/BE# ter signal PAR v visoko stanje, preden ga prepusti na razpolago ostalim napravam na vodilu.

- Čakalna stanja (**SDATA_MR, SDATA_CR**) Kadar izvajamo bralne transakcije, bodisi podatkovne ali konfiguracijske, je potrebno po naslovnem ciklu spremeniti lastništvo vodila. Da bi se izognili scenarijem, v katerih bi več naprav hkrati pošiljalo podatke na vodilo, dodamo obračalni cikel (včasih rečemo tudi čakalni cikel).

- Prenosna stanja (**MEM_RD, CFG_RD, MEM_WR, CFG_WR**) V ta stanja avtomat pride potem, ko je že vse pripravljeno za prenose. Ciljna naprava je naslovljena in vodila so ustrezno pripravljena na prenos podatkov. Tukaj se avtomat cikla dokler ne konča prenosa (deaktiviran signal FRAME# in aktivirana TRDY# ter IRDY#). Bodisi čaka na pogoje za prenos (TRDY#, IRDY#), bodisi prenaša podatke. Na tem mestu bi po potrebi lahko implementirali še ostale ukaze iz tabele 2.2.1.

2.4.6 Uporabniško vezje

S tem vezjem realiziramo funkcionalnost naprave, ki s pomočjo ostalih delov vezja lahko komunicira z drugimi napravami na PCI vodilu. Ker sem se želel čimbolj omejiti na generičen del razvoja PCI naprave, je uporabniški del v

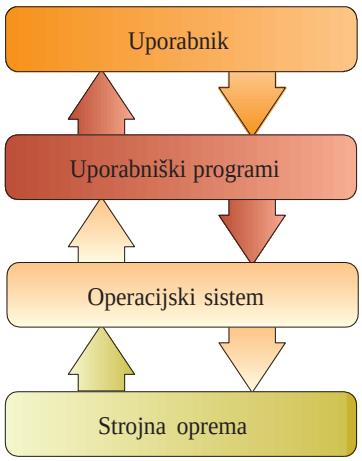
mojem primeru implementacijsko skop. Da bi vseeno lahko nekako komuniciral s PCI napravo, sem uporabniški del zasnoval kot bločni pomnilnik velikosti 4kB (1024 naslovov z 32 bitno širino besede).

Poglavje 3

Gonilnik

3.1 Operacijski sistemi

Operacijski sistem je skupek sistemskih programov, ki leži med strojno opremo in uporabniškimi programi. Njegova naloga je, da čimbolj učinkovito dodeljuje omejena sistemska sredstva (procesorski čas, pomnilnik, dostop do perifernih naprav, ipd.), ki jih ima na razpolago. Z uporabniškimi programi komunicira preko sistemskih klicev. To jim omogoča uporabo sistemskih sredstev na poenoten (abstrakten) način. S tem programe omejuje in jih poenostavlja.



Slika 3.1: Plasti operacijskega sistema

Primer, ko omejuje: vzemimo za primer gonilnik za datotečni sistem. Ta je del operacijskega sistema in nam kot uporabniku, podatke na trdem

disku predstavljeni kot datoteke in mape. Vendar pa uporabnika oziroma uporabniški program omejuje v stvareh, kot so: največja dolžina imena datoteke, najmanjša možna enota na disku ipd. V tem konkretnem primeru vsakdanji uporabnik teh omejitev ne čuti - kar pa še ne pomeni, da jih ni.

Primer, ko poenostavlja: naj nadaljujem zgornji primer. Ker nam gonilnik za dotični datotečni sistem naredi abstrakcijo podatkov na disku v obliki datotek in map, uporabniški program dostopa do le-teh s preprostim sistemskim klicem (navadno po imenu datoteke) in ga ne zanima, kje ti podatki na disku fizično obstajajo. S tem se močno poenostavi kompleksnost programov.

Za izvedbo diplomskega dela sem se odločil uporabiti operacijski sistem GNU/Linux, ki je izdan pod licenco GPL (General Public Licence), kar pomeni, da je izvorna koda zanj prosta (free software) tako za vpogled, kot (pod določenimi pogoji) tudi za spremembe. Poleg tega je izvrstno dokumentiran ter poln konkretnih primerov.
Vse te prednosti operacijskega sistema GNU/Linux imajo tudi različice BSD (Berkeley Software Distribution), vendar sem se na koncu za GNU/Linux odločil, ker imam z njim bistveno več izkušenj.

3.1.1 Gonilniki

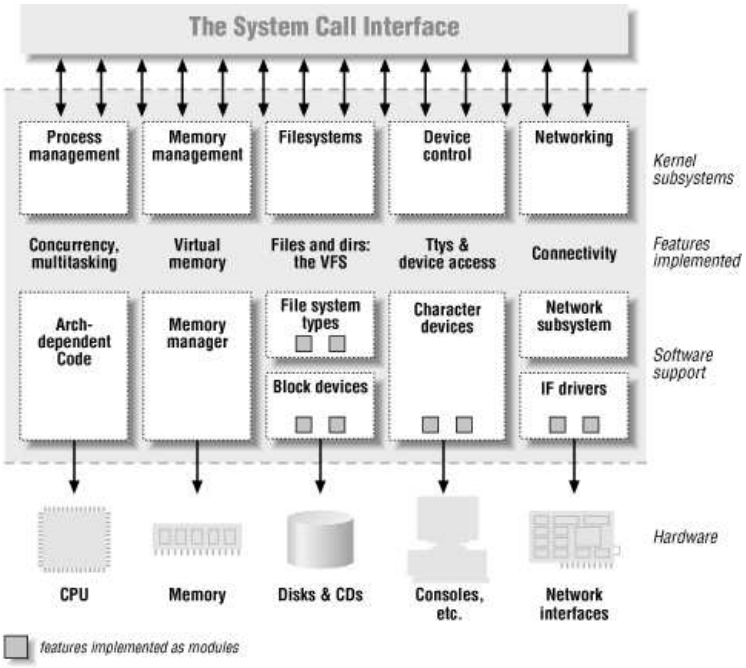
Dele operacijskega sistema, ki skrbijo za komunikacijo s strojno opremo, imenujemo gonilniki. Le-ti poenostavljajo uporabo strojne opreme tako, da ustvarijo abstrakten dostop do naprave. Abstrakcija v tem kontekstu pomeni, da gonilnik skrije podrobnosti o posebnostih naprave tako, da dostop do nje ovije v množico vnaprej dogovorjenih sistemskih klicev (API). Le-ti so vnaprej definirani za posamezne tipe naprav. Npr. vse mrežne kartice imajo isti API (Application Programming Interface), ki omogoča, da se uporabniškemu programu ni treba zavedati za kartico katerega proizvajalce gre, kakšne so njene posebnosti ipd.

3.1.2 Operacijski sistem GNU/Linux

Avtor in lastnik operacijskega sistema GNU/Linux je Linus Torvalds, ki ga je začel kot hobi razvijati leta 1991. Konec naslednjega leta je izdal različico 0.99 pod licenco GNU. Narava te licence je dala projektu neverjeten zagon, ki se

še dandanes nadaljuje ter konkurira močnim komercialnim znamkam, kot so Microsoft Window ali Apple OSX.
GNU/Linux štejemo med klone operacijskega sistema Unix.

Jedro



Slika 3.2: Vmesnik za sistemske klice v GNU/Linux (vir:[3])

V Unix-u in njegovih klonih je hkrati v operacijskem sistemu prisotno več procesov. Vsak proces zahteva sistemske vire, kot so procesorski čas, pomnilnik ali mrežna povezljivost. Jedro je osrednji proces, katerega zadolžitev je upravljanje s temi viri in njihovo dodeljevanje ostalim procesom. Kljub temu, da včasih ne vemo kaj vse je vloga jedra, lahko njegove naloge razdelimo na:

- upravljanje s procesi: jedro je zadolženo za kreiranje in uničevanje procesov, njihovo komunikacijo z zunanjim svetom ter komunikacijo z ostalimi procesi. Hkrati pa opravlja tudi nalogo dodeljevanja procesorskega časa po ustrezni strategiji.
- upravljanje s pomnilnikom: pomnilnik je eden ključnih virov v računalniškem sistemu, zato je upravljanje z njim kritično za hitrost sistema kot celote. Jedro preslikuje fizični naslovni prostor v virtualnega

(in obratno) za vse procese ter jim dodeljuje in sprošča dele virtualnega pomnilnika.

- datotečni sistem: linux je močno zasnovan na konceptu datoteke. Skoraj vse stvari so predstavljene kot datotekae. Jedro gradi strukturiran datotečni sistem, ki predstavlja strojno opremo in ta abstrakcija se uporablja v celem sistemu. Poleg tega podpira veliko množico datotečnih sistemov za organizacijo podatkov na pomnilniških medijih (klasični datotečni sistemi).

- upravljanje z napravami: praktično vsaka sistemska operacija se na koncu odraža na neki napravi. Z izjemo komponent, kot so CPE, delovni pomnilnik, ipd., je izvajanje teh operacij in nadzor nad napravami naloga programov, ki so specifični za vsako napravo. Tem programom pravimo gonilniki. Jedro mora imeti prisoten gonilnik za vse naprave, s katerimi želi razpolagati.

- podpora za mrežo: le-ta mora biti pod nadzorom jedra, ker večina operacij v tem kontekstu ni specifična za proces. Poleg tega pa so paketi, ki prihajajo, asinhron proces. Paketi morajo biti zbrani in identificirani, preden so razposlani ustreznim procesom. Naloga jedra je, da dostavlja podatkovne pakete med programi in mrežnimi vmesniki, hkrati pa skrbi še za usmerjanje.

Gonilniki v operacijskem sistemu GNU/Linux

Kot sem že omenil, sem za našo PCI napravo razvil gonilnik za operacijski sistem GNU/Linux. Ob začetku sem se moral najprej seznaniti s pravili, ki veljajo za razvoj gonilnika. Ker gre za del kode, ki teče v priviligiranem načinu znotraj jedra (kernel), se ne moremo posluževati tehnik programiranja, ki smo jih vajeni, kadar programiramo aplikacije, ki tečejo v uporabniškem prostoru. Prvi problem, na katerega sem naletel je, da nisem mogel uporabljati običajnih ukazov za izpisovanje sporočil ter razhroščevanje. Izpisovanje sporočil poteka striktno preko jedra, sporočila pa končajo v datoteki sistemskega dnevnika (log file). Poleg tega ni možno uporabljati običajnih datotek, kadar programiramo gonilnik.
Da pa le ni vse tako črno, priča dejstvo, da so se razvijalci jedra močno potrudili in napisali makre, s pomočjo katerih je razvoj gonilnikov enostavnejši.
Linux deli gonilnike za naprave na tri temeljne tipe:

1. Znakovne naprave (character devices): znakovna naprava je tista, do katere je dostop podoben toku bajtov (podobno kot datoteke) in gonilnik za te naprave je zadolžen, da implementira takšen vmesnik, ki je primeren za interakcijo s takšnim tipom naprav. Tipično implementira sistemske klice, kot so odpri (*open*), zapri (*close*), beri (*read*), in piši (*write*). Na prvi pogled ni bistvenih razlik od dela z datotekami. Edina razlika je v tem, da se po datoteki lahko vedno sprehajamo naprej in nazaj, medtem ko je znakovna naprava tipično tok podatkov, ki ga beremo zaporedno. Tipični predstavniki so:

   - tekstovna konzola (*/dev/console*),
   - serijski port (*/dev/ttyS0*),
   - tiskalniški vmesnik (*/dev/lp0*).

2. Bločne naprave (block devices): prav tako kot znakovne naprave, so dosegljive preko datotek v mapi */dev*. Bločna naprava je tista, ki lahko gosti datotečni sistem. V tipičnem Unix okolju lahko bločna naprava prenaša le bloke podatkov (enega ali več hkrati), le-ti pa so navadno velikosti 512 (ali večkratnik tega števila) bajtov. Namesto tega Linux omogoča, da bločna naprava prenaša poljubno količino podatkov (podobno kot znakovne naprave) in tako se ta dva tipa naprav ločita le po tem, kako jedro interno obravnava podatke ter tako vpliva na vmesnik gonilnik - program, ki je zato zelo drugačen.

3. Mrežne naprave (network devices): Vsaka mrežna transakcija gre skozi vmesnik - napravo, ki se je sposobna povezati z drugimi računalniki. Običajno je ta vmesnik strojna mrežna naprava, lahko pa je tudi popolnoma programska (npr. vmesnik *loopback*). Mrežni vmesnik je zadolžen za pošiljanje in prejemanje posameznih podatkovnih paketov (z njim upravlja jedro), ne zaveda pa se pojma transakcij. Veliko mrežnih povezav bazira na vzpostavljeni zvezi, mrežni vmesniki pa so navadno osnovani okoli prejemanja in pošiljanja paketov. Tako mrežni gonilnik ne ve ničesar o posameznih povezavah - zadolžen je samo za podatkovne pakete.

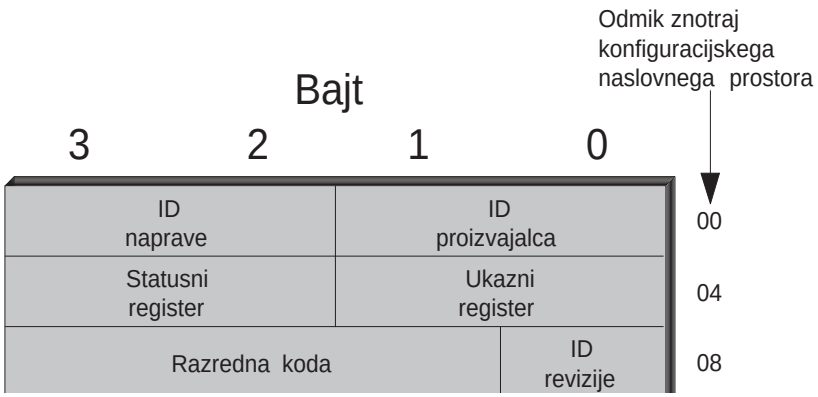
   Mrežni vmesnik ni tokovno orientirana naprava, zato jo je praktično nemogoče predstaviti kot datoteko. Zaradi tega so mrežni vmesniki v Linuxu predstavljeni z unikatnim imenom (npr. eth0, lo), ki nima pripadajoče datoteke v */dev*. Tudi komunikacija med mrežnim gonilnikom in jedrom poteka popolnoma drugače, kot komunikacija z znakovnimi ali

bločnimi napravami. Namesto branja in pisanja jedro uporablja klice za paketni prenos.

Naprave v Linuxu lahko klasificiramo še na drug način, in to ortogonalno na zgornjo klasifikacijo. Tipi naprav kot so USB (Universal Serial Bus), SCSI (Small Computer System Interface), FireWire in drugi ter nenazadnje tudi PCI. Primer: Vzemimo USB ključek - to je USB naprava, ki je hkrati tudi bločna naprava.

3.1.3 Gonilnik *custompci*

Glede na omenjeno sem klasificiral našo PCI kartico kot **znakovno PCI napravo**.
PCI protokol se sklicuje na vrednosti, ki so zapisane v določenih bralnih registrih v konfiguracijskem prostoru. Po pravilih standarda bi bilo treba PCISIG (PCI Special Interest Group) zaprositi za dodelitev unikatne številke, ki bi jo potem vpisovali v register ID proizvajalca. Seveda nismo registriran proizvajalec PCI naprav, zato sem si v namene izdelave tega diplomskega dela vrednosti tega registra izmislil. Treba pa je poudariti, da je to vrednost potrebno konsistentno uporabljati tako v napravi (v našem primeru VHDL), kot tudi v gonilniku.



Slika 3.3: Nekateri registri v glavi konfiguracijskega naslovnega prostora

Vrednosti relevantnih registrov:

- ID proizvajalca: 0x106D - to je ID, ki je bil dejansko dodeljen in sicer podjetju Sequent Computer Systems

- Register razredne kode: 0x058000
- ID naprave: 0x0001
- ID Revizije: 0x01

Razredno kodo sem določil iz tabel, ki so preobširne, da bi jih vključil v to delo. Bralec jih lahko najde v [2]. Vseeno naj povem, kakšen je pomen posameznih bajtov (slika 2.5) v tem registru za naš primer:

- razredna koda: 0x05 - pomnilniški krmilnik
- podrazredna koda: 0x80 - neopredeljen pomnilniški krmilnik
- programski vmesnik: 0x00 - za pomnilniške krmilnike ni definiranih programskih vmesnikov, zato vedno vračamo same ničle.

Sedaj, ko so nam znane vrednosti teh registrov, se lahko lotimo pisanja gonilnika.
Gonilniki za Linux so napisani v programskem jeziku C. Razvoj v kakem drugem (višjem) programskem jeziku ni podprt, kar je po svoje razumljivo, saj je pisanje gonilnikov programiranje na najnižjem programskem nivoju, kar ne ustreza visokim programskim jezikom. V C-ju je mogoče vključiti tudi kodo v zbirnem jeziku (inline assembler), tako da ni mogoče, da česa ne bi bilo mogoče napisati zaradi omejitev v samem jeziku. Včasih pa se programerji poslužujejo zbirnika tudi zato, ker ne zaupajo prevajalniku ali pa, ker so prepričani, da lahko sami napišejo bolj optimalno kodo, kot bi jo naredil prevajalnik iz C-ja. Na začetku pisanja gonilnika določimo, katere knjižnice bomo uporabili. Izbirati je možno samo med knjižnicami, ki so prisotne v izvorni kodi jedra. Torej običajnih knjižnic kot so `stdio.h` in `stdlib.h`, ki smo jih sicer vajeni, tukaj ne moremo uporabljati.

```
#include <linux/kernel.h> // Obicajne knjiznice
#include <linux/init.h>   //   za razvoj gonilnika
#include <linux/module.h> //  ------ || -------
#include <linux/fs.h>     // Strukture/makri za uporabo
                          //  /dev datotecnega sistema
#include <linux/pci.h>    // Strukture/makri za razvoj
                          //   PCI naprav
#include <linux/miscdevice.h>
#include <asm/uaccess.h>  // Ukazi za komunikacijo:
                          //   jedro <- - -> userspace
#define DEVICE_NAME             "custompci"
```

```
MODULE_LICENSE("GPL");
```

Preden nadaljujemo s programiranjem, je potrebno jedru povedati za kakšno licenciranje gre pri našem gonilniku. To naredimo z makrojem `MODULE_LICENSE` (npr. `MODULE_LICENSE("Dual BSD/GPL");`). Če tega ne naredimo, ali pa, če se licenca ne sklada z Linuxovo, jedro vsakokrat, ko gonilnik naloži, potoži, da licenca ni prava.

Določimo statično polje struktur *pci_device_id*.

```
static struct pci_device_id ids[] = {
        { PCI_DEVICE(0x106D, 0x001), },
        { 0, }
};
```

Jedro ga uporablja zato, da ve kateri gonilnik je odgovoren za katero PCI napravo (več kot očitno je ta struktura namenjena za PCI naprave). Makro `PCI_DEVICE` pa nam generira strukturo *pci_device_id*. Parametra tega makra sta ID proizvajalca in ID naprave. Za vrednosti ID podsistema in ID proizvajalca podsistema ta makro določi konstanto `PCI_ANY_ID`. To pomeni, da smo z vrednostmi v tej strukturi jedru povedali, da je to gonilnik zadolžen za našo napravo.

Vsak gonilnik ima najmanj dve statični funkciji: za inicializacijo gonilnika in za primer, ko jedro odstrani gonilnik iz pomnilnika. Prvi je namenjen zaseganju drugi pa sproščanju resursov. Imena funkcij so poljubna, vendar jih je potrebno registrirati z makrojema `module_init` in `module_exit`.

```
module_init(custompci_init_module);
module_exit(custompci_exit_module);
static int __init custompci_init_module(void)
{
        int ret;
// registriramo PCI gonilnik
        ret = pci_register_driver(&pci_driver);
        misc_register(&custompci_dev);
        return ret;
}
```

```
static void __exit custompci_exit_module(void)
{
// sprostimo resurse
        misc_deregister(&custompci_dev);
        pci_unregister_driver(&pci_driver);
}
```

Inicializacija se sklicuje na dve podatkovni strukturi, kateri je še potrebno definirati.

- `pci_driver` je struktura, ki predstavlja PCI napravo kot jo vidi računalnik
- `custompci_dev` je struktura, s katero jedro predstavi napravo v */dev*

```
static struct pci_driver pci_driver = {
        .name = "custompci",
        .id_table = ids,
        .probe = probe,
        .remove = remove,
};

/* Struktura operacij nad datoteko definirana
    v linux/fs.h                                */
static struct file_operations custompci_fops = {
  .owner    =   THIS_MODULE,
  .open     =   custompci_open,
  .release  =   custompci_release,
  .read     =   custompci_read,
  .write    =   custompci_write,
};

static struct miscdevice custompci_dev = {
        MISC_DYNAMIC_MINOR,
        DEVICE_NAME,    // ime, ki je v konstantah
                        // (/dev/custompci)
        &custompci_fops // katere operacije nad
                        // datoteko v /dev  podpira
};
```

Te strukture nam definirajo še šest novih funkcij, ki jih moramo implementirati, da bo naša naprava delovala tako, kot želimo. Implementacija teh funkcij ni posebno težka naloga, bralec pa jih lahko najde v prilogi.
Razvijalci Linuxovega jedra so močno olajšali razvoj gonilnikov. Programer mora paziti le, da se drži pravil, ki so jih postavili.

3.1.4 Delovanje gonilnika *custompci*

Najprej naložimo gonilnik in pogledamo kako napravo vidi OS.

```
root@ubuntu:~/src$  insmod custompci-driver.ko
root@ubuntu:~/src$  lspci -s 00:09.0 -vv
00:09.0 Memory controller [0508]: Sequent Computer
Systems Device 0001 (rev 01)
        Subsystem: Device bebe:0001
        Control: I/O- Mem+ BusMaster- SpecCycle- MemWINV-
                 VGASnoop- ParErr- Stepping- SERR-
                 FastB2B- DisINTx-
        Status: Cap- 66MHz- UDF- FastB2B- ParErr-
                DEVSEL=fast >TAbort- <TAbort- <MAbort-
                >SERR- <PERR- INTx-
        Region 0: Memory at e8009000 (32-bit,
                  non-prefetchable) [size=4K]
        Kernel driver in use: custompci
```

Pravzaprav se tukaj na enem mestu vidijo vse lastnosti PCI naprave, ki smo jih nastavljali preko konfiguracijskih registrov. Poleg ostalega vidimo kje in kolikšen je naslovni prostor ter, da je le-ta v pomnilniško preslikanem prostoru.

Preizkusimo še delovanje branja in pisanja.

```
root@ubuntu:~/src$ echo -n "aabb" > /dev/custompci
root@ubuntu:~/src$ dd if=/dev/custompci count=1 | hd
00000000  61 61 62 62     |aabb|
```

S prvim ukazom sem vpisal vrednost "aabb"v PCI napravo, z drugim pa sem to vrednost tudi uspešno prebral. Iz tega lahko sklepam, da naprava in gonilnik pravilno delujeta.

Poglavje 4

Sklepne ugotovitve

V diplomski nalogi sem predstavil razvoj prototipa razširitvene kartice za PCI vodilo. Proces me je popeljal skozi tri glavne faze razvoja: načrtovanje, razvoj strojne opreme in programiranje gonilnika. Pokazal se je tudi odličen koncept standarda PCI. Kljub zelo obširni specifikaciji in možnostih, ki jih definira ta standard, je mogoče vzeti le majhen del tega, pa bo vseeno zadoščalo specifikaciji. To je za razvoj relativno enostavnih vezij še kako pomembno. Prav takšno je tudi vezje, ki sem ga razvil kot predmet te naloge in je bolj mišljeno kot osnova za nadaljni razvoj oziroma kot opis koncepta delovanja PCI.

Zanimivo bi bilo videti uporabljeno prototipno kartico v vlogi grafične kartice, mrežne kartice ali celo PCI analizatorja. Kartica sama ima možnosti, da se to naredi in menim, da je moje delo lahko zelo dobra osnova za izdelavo česa takšnega.

Ali se bo našel kakšen študent, ki bi mu to predstavljalo dovoljšen izziv, pa bo pokazal čas.

Slike

Tabele

Literatura

[1] P. Bulić, *Načrtovanje in sinteza digitalnih in mikroprocesorskih sistemov v FPGA*, 2007, Dostopno na:
http://www.fri.uni-lj.si/file/84055/vhdl_zapiski.pdf

[2] Tom Shanley, Don Anderson, *PCI System Architecture* četrta izdaja, Addison-Wesley, 1999

[3] Jonathan Corbet, Greg Kroah-Hartman, Alessandro Rubini, *Linux Device Drivers* tretja izdaja, O'Reilly, 2005, Dostopno na:
http://lwn.net/Kernel/LDD3/

[4] Dušan Kodek, *Arhitektura računlniških sistemov* druga popravljena in razširjena izdaja, BI-TIM, 2000

[5] *PCI Local bus specification* revizija 2.2, PCISIG, 1998

[6] *Xilinx Spartan-3 Evaluation Kit User Guide*, AvNet, 2004, Dostopno na:
http://avnetexpress.avnet.com (potrebna predhodna registracija)

[7] *Conventional PCI* Dostopno na:
http://en.wikipedia.org/wiki/PCI_Local_Bus