Arhitekture mikrosistema

Goran Lj. Đorđević

PREDGOVOR

Ova knjiga je posvećena tehnikama projektovanja digitalnog hardvera na mikroarhitekturalnom, odnosno RTL (engl. ***R**egister **T**ransfer **L**evel*) nivou apstrakcije. Projektovanje na RTL nivou podrazumeva da se za konstrukciju digitalnih sistema koriste moduli kao što su sabirači, komparatori, multiplekseri i registri. Savremeno RTL projektovanje zasnovano je na upotrebi jezika za opis hardvera i softvera za automatsku sintezu hardvera. U knjizi je obrađen jezik VHDL, kao najrasprostranjeniji i najsveobuhvatniji jezik za opis hardvera danas.

Može se slobodno reći da je značaj koji VHDL ima u digitalnom projektovanju jednak onom koji programski jezik C/C++ ima u programiranju, a da poznavanje VHDL-a spada u obavezna znanja svakog projektanta digitalnog hardvera. VHDL nalazi brojne primene u digitalnom projektovanju. Prvo, u VHDL-u je moguće kreirati strukturne modele digitalnih sistema, odnosno opise koji na formalan način pokazuju kako je sistem podeljen na podsistemi i kako su podsistemi međusobno povezani. Drugo, VHDL takođe omogućava kreiranje funkcionalnih modela korišćenjem jezičkih konstrukcija poput programskih grananja i petlji iz konvencionalnih programskih jezika. Treće, VHDL opis nekog sistema se može simulirati, s ciljem da se pre fizičke realizacije, testiraju i verifikuju njegova funkcionalnost i performanse. Na taj način, projektant dobija mogućnost da brzo ispita različita rešenja, bez potrebe izrade hardverskog prototipa. Četvrto, uz pomoć specijalizovanog softvera, VHDL opis se može sintetizovati, odnosno transformisati u detaljan strukturni opis koji je pogodan za neposrednu fizičku realizaciju. Na taj način se projektant rasterećuje od detaljnog projektovanja na niskom nivou (kao što je npr. minimizacija logičkih funkcija), pa više vremena može da posveti razradi arhitekture sistema i istraživanju alternativnih rešenja na sistemskom nivou.

Ova knjiga ima za cilj da uvode studente u oblast digitalnog projektovanja kroz izučavanje VHDL-a, a pre svega onih aspekata ovog jezika koji su bitni za hardversku sintezu. Uspešno ovladavanje izloženom materijom pretpostavlja poznavanje osnovnih koncepta digitalnog projektovanja, poput onih koja se stiču u prvom kursu digitalne elektronike. Takođe, poznavanje programiranja i bar jednog programskog jezika, zajedno sa osnovama iz organizacije računara može biti od velike pomoću studentima za razumevanje i brzo usvajanje izložene materije. Knjiga je namenjena prevashodno studentima elektronike i računarstva, kao uvod u oblasti mikroprocesora i embeded sistema, s jedne i VLSI projektovanja s druge strane. Nakon uspešnog savladavanja materije obuhvaćene ovom

knjigom, student bi trebalo da bude u mogućnosti da samostalno projektuje RTL sisteme i podsisteme malog i srednjeg nivoa složenosti.

Knjiga nema ambiciju da obuhvati sve aspekte digitalnog projektovanja. Aspekti poput fizičkog projektovanja, simulacije i testiranja, biće razmatrani samo iz perspektive RTL projektovanja i to u meri koja je neophodna za razumevanje principa i tehnika RTL projektovanja i hardverske sinteze. Sveobuhvatno projektovanje digitalnih kola i sistema zahteva kako dodatna teorijska znanja tako i poznavanje specijalizovanih softverskih alata za projektovanje.

Knjiga je podeljena na devet glava. Kao uvod u materiju, u prvoj glavi su predstavljeni osnovni principi, metodologija i tok projektovanja savremenih digitalnih sistema, zajedno s ulogom koju u tom procesu igraju jezici za opis hardvera i softverski alati za projektovanje. Glave 2 – 8 bave se različitim aspektima jezika VHDL. U 2. glavi, dat je pregled osnovnih karakteristika i koncepta ovog u suštini složenog i sveobuhvatnog jezika. Glava 3 posvećena je osnovnim jezičkim konstrukcijama VHDL-a, kao što su tipovi podataka, operatori i atributi. VHDL je konkurentan jezik s podrškom za sekvencijalno programiranje, po čemu se razlikuje od većine konvencionalnih programskih jezika, koji pružaju podršku isključivo za sekvencijalno programiranje. U 4. glavi razmatraju se različiti oblici konkurentne naredbe dodele, kao osnovnog jezičkog mehanizma za kreiranje konkurentnog kôda. Sekvencijalno programiranje u VHDL-u obrađeno je u 5. glavi. Sekvencijalni kôd se koristi za modeliranje sekvencijalnih digitalnih kola, kao što su registarske komponente i konačni automati. Zbog značaja koji imaju u digitalnom projektovanju, konačnim automatima je posvećena posebna glava, glava 6. U glavama 7 i 8, obrađeni su aspekti VHDL-a koji su bitni za projektovanje na sistemskom nivou i organizaciju složenih projekata. Glava 7 posvećena je strukturnom i hijerarhijskom projektovanju, dok se glava 8 bavi jezičkim konstrukcijama i programskim tehnikama za kreiranje parametrizovanih modela. Konačno, 9. glava posvećena je tehnikama projektovanja digitalnih sistema na RTL, odnosno mikroarhitekturalnom nivou apstrakcije.

Knjiga sadrži veliki broj dijagrama i VHDL opisa koji na konkretnim primerima ilustruju primenu izložene materije i doprinesu bržem i potpunijem usvajanju osnovnih koncepta digitalnog projektovanja.

Goran Lj. Đorđević

U Nišu, januara 2009.

SADRŽAJ

1. PROJEKTOVANJE DIGITALNIH SISTEMA

Projektovanje savremenih digitalnih sistema predstavlja fuziju tri podjednako bitna faktora: poluprovodničke tehnologije, metodologije projektovanja i softverskih alata za projektovanje. Zahvaljujući neprestanom razvoju i usavršavanju poluprovodničke tehnologije, od vremena prvih digitalnih kola pa sve do danas, broj tranzistora koji se ugrađuju u integrisana kola (IC) povećavao se eksponencijalnom brzinom. S takvim tempom razvoja, danas srećemo digitalna IC sa više stotina, više hiljada, pa sve do više stotina miliona tranzistora. U prošlosti, digitalna kola su nalazila glavnu primenu u računarskim sistemima. Međutim, vremenom, kako su digitalna IC postajala sve manjih dimenzija i brža, sa sve većim mogućnostima i u isto vreme sve jeftinija, mnogi elektronski, komunikacioni i upravljački sistemi su interno digitalizovani korišćenjem digitalnih kola za memorisanje, obradu i prenos informacija.

Kako su primene digitalnih sistema postajale šire, a IC sve složenija, tako je i zadatak projektovanja digitalnog hardvera postajao sve teži. Praktično je nemoguće neposredno na nivou tranzistora, pa čak ni na nivou logičkih kola projektovati digitalni sistem čija je složenost veća od nekoliko hiljada tranzistora. Projektovanju u toj meri složenih sistema mora se pristupiti na organizovan i sistematičan način. Savremena metodologija projektovanja digitalnih sistema podrazumeva da se u početnoj fazi projektovanja sistem podeli na manje podsisteme. Najpre se za svaki podsistem kreira apstraktni, funkcionalni model, koji se potom postepeno razrađuje sve do nivoa logičkih kola, a onda i dalje do nivoa tranzistora i fotolitografskih maski. Softverski alati za projektovanje igraju nezamenjivu ulogu u ovom složenom procesu. Zadatak projektanta je da osmisli sistem i formalno ga opiše, a zadatak softverskog alata da obavi simulaciju modela radi verifikacije njegovih funkcionalnih karakteristika, a zatim i da transformiše apstraktni opis u detaljan strukturni opis niskog nivoa, koji je pogodan za neposrednu fizičku realizaciju. Potreba za formalnim modeliranjem, koja se nalazi u osnovi savremene metodologije digitalnog projektovanja, dovela je do razvoja i široke primene jezika za opis hardvera. Danas su u praksi najzastupljeniji dva takva jezika sličnih mogućnosti: VHDL ili Verilog. U ovoj knjizi predstavljen je jezik VHDL, s naglaskom na one njegove aspekte koji su bitni za sintezu hardvera.

U ovoj uvodnoj glavi dat je kratak pregled implementacionih tehnologija, toka i metodologije projektovanja digitalnih sistema zajedno s ulogom koju u tom procesu igraju jezici za opis hardvera.

1.1. IC tehnologije

Tehnologija koja se koristi za realizaciju digitalnog hardvera dramatično je evoluirala u toku protekle četiri decenije. Sve do sredine 60' godina prošlog veka, logička kola su realizovana uz pomoć diskretnih komponenti (diskretnih tranzistora i otpornika). Razvojem tehnologije integrisanih kola, postalo je moguće smestiti veći broj tranzistora, a onda i kompletno digitalno kolo na jedan čip. U početku, IC su sadržala tek do nekoliko desetina tranzistora, ali s napretkom tehnologije ovaj broj se brzo povećavao. Početkom 70' godina prošlog veka postalo je moguće na jednom čipu realizovati kompletan mikroprocesor. Iako u mnogim aspektima inferiorni u odnosu na savremene mikroprocesore, prvobitni mikroprocesori su otvorili put ka svojevrsnoj revoluciji u oblasti obrade informacija, koja se ogledala kroz pojavu personalnih računara, a onda i mikroračunarskih sistema specijalne namene. Pre oko 40 godina, istraživač Gordon Mur je publikovao predviđanje po kome će se broj tranzistora koji se mogu integrisati na jednom čipu duplirati svake 1.5 do 2 godine. Ovaj fenomen, neformalno poznat kao Murov zakon, važi i danas. Tako je ranih 1990' bilo moguće proizvesti mikroprocesore s nekoliko miliona, a krajem 1990' s nekoliko desetina miliona tranzistora. Danas, pred kraj prve decenije 21. veka, aktuelni su mikroprocesori sa više stotina miliona tranzistora. Predviđanja govore da će se ovakav eksponencijalni tempo razvoja nastaviti i tokom narednih desetak godina.

Fabrikacija integrisanih kola. Integrisano kolo je sačinjeno od slojeva dopiranog silicijuma, polisilicijuma, metala i silicijum dioksida poređanih jedan iznad drugog na silicijumskoj podlozi. Na nekim od ovih slojeva formiraju se tranzistori, dok se na drugim izvode provodnici za povezivanje tranzistora. Osnovni korak u fabrikaciji IC-a je formiranje jednog sloja, procesom koji se zove *litografija*. Raspored materijala na jednom sloju definisan je *maskom*. Slično fotografskom filmu, maska se koristi da bi se određene oblasti na površini silicijuma izložile dejstvu hemikalija u cilju kreiranja različitih slojevi materijala na površini jednog sloja. Izgled maske se zove *lejaut* (engl. *layout*). Savremene IC tehnologije koriste od 10 do 15 slojeva, što znači da u toku fabrikacije jednog integrisanog kola, litografski proces mora biti ponovljen 10 do 15 puta, uvek s različitom maskom.

Klasifikacija IC tehnologija. Digitalna kola se mogu realizovati u različitim tehnologijama. Da bi se smanjili troškovi proizvodnje, kod nekih IC tehnologija veći broj slojeva se uvek izvodi na isti način, pomoću istih maski, nezavisno od funkcije koju IC treba da realizuje, dok se preostale maske kreiraju u skladu sa zahtevima konkretne primene. Međutim, bilo da se kompletno IC ciljano projektuje i proizvodi za konkretnu primenu, bilo da se IC kreira doradom polu-fabrikovanih IC-ova, proces proizvodnje je složen i skup, i može se obaviti samo u fabrici poluprovodnika. Naručilac IC-a dostavlja fabrici poluprovodnika specifikaciju IC-a na osnovu koje se u fabrici kompletira set maski, a zatim fabrikuju IC i isporučuju naručiocu. Kod drugih IC tehnologija, fabrika poluprovodnika isporučuje naručiocima univerzalna IC, odnosno IC koja nemaju unapred definisanu funkciju, ali se zato mogu *programirati*. Programiranje vrši sam korisnik, ″na terenu″, punjenjem konfiguracionih podataka u internu memoriju IC-a ili sagorevanjem internih silicijumskih osigurača. IC iz prve grupe, tj. ona koja se dorađuju u fabrici, poznata su pod skraćenicom ASIC, koja potiče od pojma aplikaciono-specifična IC (engl. ***A****pplication-****S****pecific* ***I****ntegrated* ***C****ircuit*). Druga grupa IC-ova, tj. ona koje programira krajnji korisnik, poznata je pod nazivom programabilne logičke komponente, ili PLD (engl. ***P****robrammable* ***L****ogic* ***D****evices*). Svaka od ove dve grupe sadrži više specifičnih tehnologije. Tako se ASIC tehnologije dele na:

IC ″potpuno po narudžbi″ (engl. *full-custom*). Ne koriste se prefabrikovane maske ili slojevi, već se IC ″iz početka″, do nivoa pojedinačnih tranzistora i veza, projektuje i proizvodi za tačnu određenu primenu. Na taj način, omogućena je integracija najvećeg broja tranzistora i realizacija IC-ova najveće brzine rada. Međutim, budući da je projektovanje na nivou tranzistora složeno i dugotrajno, projektovanje potpuno po narudžbi se koristi samo za IC male složenosti ili za IC koja će se proizvoditi u masovnim serijama, kako bi se amortizovali visoki jednokratni troškovi projektovanja i pripreme proizvodnje.

Standardne ćelije (engl. *semi-custom*). Deo troškova koji prate projektovanje potpuno po narudžbi može se izbeći ako se koristi tehnologija *standardnih ćelija* (engl. *standard cell*). Standardne ćelije su moduli jednostavne funkcije (tipa NOT, AND, OR, NAND, flip-flop i slična) unapred projektovani do nivoa lejauta koji se u vidu gradivnih blokova koriste za konstrukciju složenijih struktura na čipu. Projektovanje je značajno pojednostavljeno u poređenju sa tehnologijom ″potpuno po narudžbi″, jer se sistem projektuje na nivou logičkih kola umesto na nivou tranzistora. Međutim, iako je lejaut standardnih ćelija unapred definisan, lejaut kompletnog IC zavisi od rasporeda i povezanosti standardnih ćelija. Zbog toga je proces fabrikacija identičan kao kod tehnologije ″potpuno po narudžbi″ - svi slojevi se iznova kreiraju za svako novo IC.

Gejtovska polja (engl. *gate array*). U gejtovskom polju, pojedini delovi čipa su unapred fabrikovani, dok se preostali dorađuju shodno zahtevima konkretne primene. Kod ove tehnologije, proizvođač unapred, nezavisno od ciljne primene IC-a, sprovodi najveći broj koraka, tipično one koji se tiču kreiranja tranzistora. Rezultat je silicijumska pločica nedovršenog čipa. Gejtovsko polje čini veliki broj identičnih, ali nepovezanih gejtova raspoređenih na silicijumskoj pločici u vidu regularne dvodimenzionalne matrice. Gejtovi su obično 3- ili 4-ulazna NI ili NILI logička kola, a na jednoj pločici ih može biti i do više miliona. Naknadnom doradom, koja podrazumeva kreiranje metalnih veza, gejtovi se povezuju na način koji diktira konkretna primena. Ušteda u ceni proizvodnje, u odnosu na tehnologiju standardnih ćelija, postiže se zato što proizvođač može da amortizuje cenu fabrikacije gotovih čipova masovnom proizvodnjom polu-fabrikovanih, identičnih pločica gejtovskih polja.

Programabilne logičke komponente se dele na:

SPLD (engl. ***S**imple **PLD*** - Jednostavne PLD komponente). U ovu grupu PLD komponenti spadaju programabilne strukture tipa PLA, PAL i PROM. Koncept SPLD kola zasnovan je na činjenici da se bilo koja logička funkcija može realizovati u vidu zbira logičkih proizvoda. Stoga, glavni deo SPLD kola čine dve prekidačke mreže: AND mreža u kojoj se formiraju logički proizvodi i OR mreža na čijim izlazima se dobijaju logičke sume proizvoda formiranih u AND mreži. Kod PLA obe mreže su programabilne, kod PAL-a AND mreža je programabilna, a OR fiksna, dok je kod PROM-a AND mreža fiksna, a OR programabilna. Danas se SPLD komponente više ne proizvode u vidu integrisanih kola, ali se zato koriste kao unutrašnji blokovi ASIC IC ili složenijih tipova PLD kola.

CPLD (engl. ***C**omplex **PLD*** - Složene PLD komponente). CPLD kolo sadrži veći broj programabilnih logičkih blokova spregnutih posredstvom centralne programabilne prekidačke matrice. Svaki programabilni logički blok je sličan po strukturi tipičnom

PLA ili PAL kolu. Programiranjem CPLD komponente definiše se kako funkcija logičkih blokova tako i način na koji su logički blokovi međusobno povezani.

FPGA (engl. ***F**ield-**P**rogrammable **G**ate **A**rray*) Arhitektura FPGA kola nije zasnovana na prekidačkim mrežama, kao što je to slučaj sa SPLD i CPLD komponentama. Umesto toga, kod FPGA se za realizaciju logičkih funkcija koriste *logičke ćelije*, koje nalikuju ćelijama gejtovskog polja i tipično sadrže jedan D flip-flop i jednu ili nekoliko *lukap* tabela (engl. *look-up table*). Lukap tabela je RAM-a malog kapaciteta koji se koristi za realizaciju jednostavnih logičkih funkcija. Logičke ćelije su raspoređene u dvodimenzionalno polje. Horizontalni i vertikalni kanali između redova i kolona logičkih ćelija, sadrže veze i programabilne prekidače posredstvom kojih se ostvaruje željeno interno povezivanje logičkih ćelija.

Savremena FPGA kola omogućavaju realizaciju digitalnih sistema složenosti od nekoliko desetina hiljada do nekoliko miliona ekvivalentnih gejtova. S toliko velikim logičkim kapacitetom na raspolaganju, primena FPGA kola nije ograničena samo na realizaciju relativno jednostavnih digitalnih struktura, kao što je to slučaj kod SPLD i CPLD kola, već pružaju mogućnost realizacije kompletnih sistema na jednom čipu zasnovanih na mikroprocesoru.

1.2. Domeni projektovanja

Projektovanje digitalnih sistema se ostvaruje u tri različita *domena*, kojima odgovaraju tri različita načina predstavljanja (reprezentacije) sistema:

- funkcionalni,
- strukturni i
- fizički.

Funkcionalni** (ili **bihejvioralni**) **domen je onaj u kome se na sistem gleda kao na ″crnu kutiju″, a projektant je fokusiran na opis ponašanja sistema u funkciji ulaza i proteklog vremena. Drugim rečima, projektant opisuje **kako** sistem radi (tj. kako treba da radi), ali ne i kako sistem treba realizovati. Cilj projektanta je da definiše odziv sistema na bilo koju kombinaciju vrednosti ulaznih promenljivih. Projektanti se bave razradom algoritma rada sistema i definišu kako sistem interaguje sa svojim okruženjem. Rezultat projektovanja je *funkcionalna reprezentacija* (odnosno *funkcionalna specifikacija*), koja može biti data u obliku dijagrama toka, programa u višem programskom jeziku, opisa u jeziku za opis hardvera, matematičke formule, grafikona i sl.

Strukturni domen je onaj u kome se sistem razlaže na skup komponenti i njihovih veza. Projektant treba da dâ odgovor na pitanje: kako pomoću raspoloživih komponenti realizovati sistem koji će ostvariti zadatu funkciju? Rezultat projektovanja je *strukturna reprezentacija* sistema, koja može biti data u obliku blok dijagrama koji prikazuje komponente sistema i njihove veze, šematskog prikaza mreže logičkih kola, ili mreže tranzistora.

Fizički domen je onaj u kome se definišu fizičke karakteristike sistema koje se odnose na dimenzije i poziciju svake pojedinačne komponente i veze u okviru prostora raspoloživog na štampanoj ploči ili poluprovodničkom čipu. Projektovanje u ovom domenu polazi od strukturne reprezentacije, a bavi se fizičkim raspoređivanjem i povezivanjem komponenti. Rezultat projektovanja je *fizička reprezentacija* sistema u obliku lejauta čipa, mehaničkih

crteža ili crteža štampane ploče. Fizička reprezentacija predstavlja konačni produkt projektovanja u obliku proizvodne dokumentacije na osnovu koje se može direktno realizovati sistem ili fabrikovati čip.

1.3. Nivoi apstrakcije

Projektovanje elektronskih sistema se može obavljati na nekoliko različitih nivoa apstrakcije ili nivoa detaljnosti, gde je svaki nivo definisan tipom gradivnih elemenata (komponenti) koji se koriste za konstrukciju sistema. U opštem slučaju, kod elektronskih sistema mogu se identifikovati sledeća četiri nivoa apstrakcije, počev od najnižeg:

- nivo tranzistora,
- nivo gejtova,
- RTL nivo i
- nivo procesora

Pregled nivoa apstrakcije dat je u tabeli T. 1-1, a grafička ilustracija na Sl. 1-1.

T. 1-1 Nivoi apstrakcje

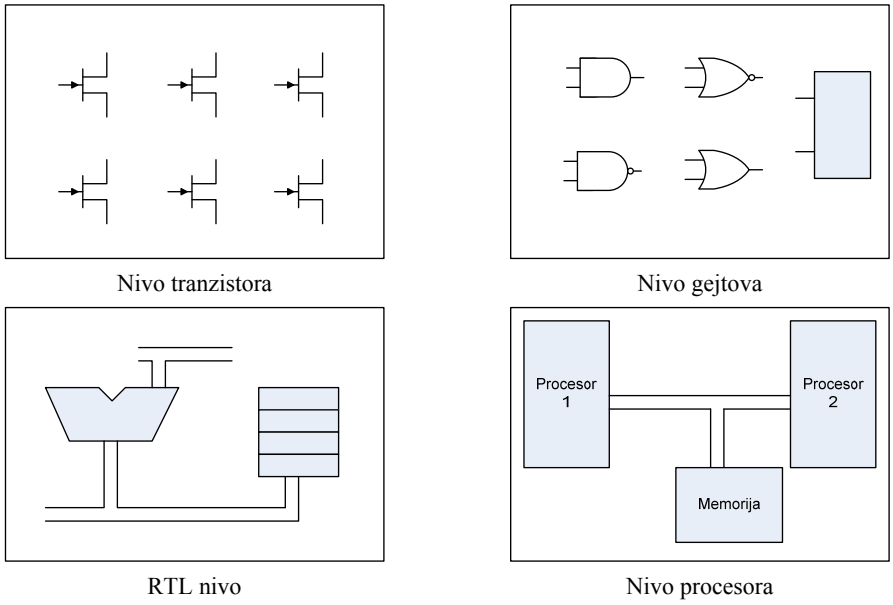
Nivo apstrakcije	**Bihejvioralna reprezentacija**	**Strukturne komponente**	**Fizički objekti**
Nivo tranzistora	Diferencijalne jednačine, U-I dijagrami	Tranzistori, otpornici, kondenzatori	Analogne i digitalne ćelije
Nivo gejtova	Logičke funkcije, konačni automati	Logička kola, flip-flopovi	Moduli, funkcionalne jedinice
RTL nivo	Algoritmi, dijagrami toka, ASM dijagrami	Sabirači, komparatori, registri, brojači, multiplekseri	IC
Nivo procesora	Izvršna specifikacija, programi	Procesori, kontroleri, memorije, ASIC kola	Štampane ploče, mikročip-moduli

Kao što se može uočiti iz tabele T. 1-1, glavne komponente na tranzistorskom nivou su tranzistori, otpornici i kondenzatori. Povezivanjem ovih komponenti formiraju se analogna i elementarna digitalna kola. Na nivou tranzistora, funkcionalnost realizovanih kola obično se opisuje diferencijalnim jednačinama ili nekom vrstom prenosne karakteristike koja pokazuje zavisnost između napona i struja u kolu (U-I dijagrami). Fizička reprezentacija analognih i digitalnih kola se zove ćelija i definiše se u vidu tranzistorskog lejauta.

Glavne gradivne komponente nivoa gejtova su logička kola i flip-flopovi. Logička kola i flip-flopovi su dostupni projektantu u vidu tzv. digitalnih ćelija. Ove ćelije se raspoređuju po silicijumskoj površini i međusobno povezuju tako da čine aritmetičke i memorijske module ili jedinice, koje se dalje koriste kao bazične komponente na RTL nivou apstrakcije. Funkcionalni opis RTL modula, realizovanih pomoću logičkih kola i flip-flopova, predstavlja se u obliku logičkih funkcija ili u obliku dijagrama konačnih automata.

Glavne komponente RTL nivoa su aritmetičke i registarske jedinice, kao sabirači, komparatori, množači, brojači, registri i registarski fajlovi. Registarske komponente se mogu rasporediti i povezati tako da čine IC koja se koriste kao osnovne komponente na sledećem višem nivou apstrakcije. U opštem slučaju, IC se opisuju dijagramima toka, ASM dijagramima, tabelama stanja, ili programom u jeziku za opis hardvera.

Nivo procesora je najviši nivo apstrakcije budući da su osnovne komponente na ovom nivou procesori, memorijska kola i aplikaciono-specifična integrisana kola (ASIC). U opštem slučaju, više ovih komponenti se postavlja na štampanu ploču i povezuju metalnim vezama koje su odštampane na ploči. Sistem koji je sačinjen od procesorskih komponenti obično se opisuje tekstualno, izvršnom specifikacijom napisanom u jeziku za opis hardvera, algoritmom ili programom napisanim u višem programskom jeziku.



Sl. 1-1 Nivoi apstrakcije.

1.4. Proces projektovanja

Put od ideje i polazne funkcionalne specifikacije digitalnog sistema, pa do njegove konačne fizičke realizacije je dug i uključuje više koraka ili faza. Na tom putu, projektantu su na raspolaganju različite tehnike i alati koji mu pomažu da opiše svoju zamisao, proveri njenu funkcionalnu ispravnost i transformiše je u oblik pogodan za direktnu fizičku realizaciju. Uopšteno govoreći, projektovanje digitalnog sistema je iterativan proces razrade i validacije (provere). Sistem se postepeno transformiše, počev od apstraktnog opisa visokog nivoa, pa sve do nivoa lejauta. Neposredno posle svake faze razrade, projektant verifikuje funkcionalnost razrađenog opisa, kako bi pronašao i korigovao eventualne greške u projektovanju i tako osigurao da će konačan proizvod korektno funkcionisati i ispoljiti zahtevane performanse.

Glavne aktivnosti u procesu projektovanja svakog digitalnog sistema su:

- Sinteza
- Fizičko projektovanje
- Verifikacija i
- Testiranje

***Sinteza*.** Sinteza je proces razrade čiji je cilj da se uz pomoć komponenti raspoloživih na nižem nivou apstrakcije realizuje sistem opisan na višem nivou apstrakcije. Polazni opis može biti strukturnog ili funkcionalnog tipa, dok je rezultujući opis uvek strukturni, ali na nižem apstraktnom nivou. Kako proces sinteze napreduje, tako opis sistema postaje sve detaljniji. Konačni rezultat sinteze je strukturna reprezentacija gejtovskog nivoa koja sadrži bazične ćelije raspoložive u ciljnoj implementacionoj tehnologiji. Proces sinteze se obično deli na nekoliko manjih koraka, od kojih svaki ima za cilj da obavi neku specifičnu vrstu transformacije. Glavni koraci u sintezi digitalnog sistema su:

Sinteza visokog nivoa - transformacija algoritma u funkcionalni RTL opis. Algoritam, koji može biti dat u vidu programa napisanog u višem programskom jeziku, kao što je C, transformiše se u funkcionalni opis prilagođen hardverskoj realizaciji, tipično u vidu programa napisanog u jeziku za opis hardvera, kao što je VHDL.

RTL sinteza - transformacija funkcionalnog RTL opisa u strukturni opis korišćenjem komponenti sa RTL nivoa, praćena izvesnim optimizacijama sa ciljem da se smanji broj upotrebljenih komponenti.

Logička sinteza - transformacija strukturnog RTL opis u mrežu logičkih kola (tj. u kombinacionu ili sekvencijalnu mrežu). Inicijalna transformacija je praćena optimizacijom mreže kako bi se minimizovala njena veličina i/ili propagaciono kašnjenje. Za sintezu se koriste standardne logičke komponente (I, ILI, NE, flip-flopovi), a optimizacija se vrši metodama kao što su Karnoove mape, Mek Klaskijeva metoda i sl. Logička sinteza je nezavisna od ciljne implementacione tehnologije.

Tehnološko mapiranje - transformacija logičke mreže u mrežu koja sadrži isključivo ćelije raspoložive u ciljnoj tehnologiji, kao što su standardne ćelije, za tehnologiju standardnih ćelija, NI ili NILI kola, za tehnologiju gejtovskih polja, odnosno logičke ćelije, za FPGA tehnologiju. Ovo je poslednji korak u sintezi i, očigledno, zavistan je od ciljne tehnologije.

***Fizičko projektovanje*.** Fizičko projektovanje uključuje dva aspekta. Prvi se odnosi na transformaciju strukturne u fizičku reprezentaciju, npr. konverzija mreže koja je dobijena nakon tehnološkog mapiranja u lejaut. Drugi aspekt se tiče analize i optimizacije električnih karakteristika kola koje se projektuje.

Glavni zadaci fizičkog projektovanja su: prostorno planiranje, raspoređivanje i povezivanje. U fazi prostornog planiranja, projektant deli sistem na manje funkcionalne celine koje raspoređuje na određene lokacije na čipu (npr. procesor u gornji levi ugao čipa, memorija u donji levi ugao, ostali moduli u preostali deo čipa). Na ovaj način, obezbeđuje se da sve komponente koje pripadaju istoj celini budu fizički raspoređene u definisanu oblast na čipu. U fazi raspoređivanja, određuju se tačne fizičke pozicije svih komponenti, dok se u fazi povezivanja kreiraju sve potrebne veze, pri čemu se za svaku vezu pronalazi optimalna (obično najkraća) putanja. Na taj način se kreira lejaut IC-a.

Nakon obavljenog raspoređivanja i povezivanja, poznate su tačne pozicije i dužine svih veza u kolu, što omogućava proračun pratećih parazitnih kapacitivnosti i otpornosti. Ovaj proces se naziva *ekstrakcijom parametara*. Podaci ekstrahovani iz lejauta, uz poznavanje električnih karakteristika upotrebljenih komponenti, potom se koriste za proračun propagacionih kašnjenja u kolu i procenu disipacije energije.

***Verifikacija*.** Verifikacija se odnosi na proveru da li ostvareni dizajn (projekat) zadovoljava postavljene funkcionalne zahteve i zadate performanse (npr., brzina rada). Sprovodi se

neposredno nakon svakog koraka razrade (sinteze), kao i nakon obavljenog fizičkog projektovanja. Postoje dva aspekta verifikacije: funkcionalni i performansni. *Funkcionalna verifikacija* daje odgovor na pitanje: da li sistem generiše očekivani odziv na zadatu pobudu? Performanse se zadaju izvesnim vremenskim ograničenjima (npr., maksimalno kašnjenje signala od ulaza od izlaza). *Vremenska verifikacija*, ili *verifikacija tajminga*, znači proveru da li sistem generiše odziv u granicama postavljenih vremenskih ograničenja. U praksi, verifikacija se sprovodi primenom ili kombinacijom sledećih metoda:

Simulacija je najčešće korišćeni metod za verifikaciju i podrazumeva: konstrukciju funkcionalnog modela sistema, izvršenje ovog modela na računaru i analizu odziva. Na primer, model može biti opis sistema u jeziku za opis hardvera. Jedan takav opis, zajedno sa opisom ulazne pobude, učitava se u simulator i izvršava. Rezultat izvršenja je odziv modela na zadatu pobudu, predstavljen npr. u vidu talasnih oblika izlaznih signala. Simulacija nam omogućava da ispitamo rad sistema i uočimo eventualne greške pre nego što pristupimo njegovoj fizičkoj realizaciji. Međutim, ispravan rad modela u simulatoru ne znači 100%-nu garanciju korektnost realnog sistema. Model može da dâ ispravan odziv na sve stimulanse kojima je pobuđen, ali se može desiti da ni jedan stimulans nije doveo do pobude nekog specifičnog dela modela koji eventualno sadrži grešku. S druge strane, naročito u slučajevima kad model sadrži veliki broj komponenti, broj stimulansa (kaže se i test-vektora) potrebnih da bi se postigla potpuna provera može biti toliko veliki da računarsko vreme potrebno za simulaciju postane enormno veliko.

Vremenska analiza je metod verifikacije fokusiran na tajming. Vremenska analiza podrazumeva: analizu strukture kola kako bi se odredile sve moguće putanje signala od ulaza do izlaza, proračunala propagaciono kašnjenja duž ovih putanja i odredili relevantni vremenski parametri kola, kao što su maksimalno propagaciono kašnjenje i maksimalna frekvencija taktnog signala.

Formalna verifikacija podrazumeva određivanje osobina kola primenom matematičkih tehnika. Jedan takav metod je *provera ekvivalentnosti*, koji se koristi za upoređivanje dve formalne reprezentacije sistema kako bi se ustanovilo da li imaju istu funkciju. Ovaj metod se često primenjuje nakon sinteze, da bi se proverilo da li je funkcija sintetizovanog kola identična funkciji polaznog funkcionalnog opisa. Za razliku od simulacije, formalna verifikacija može garantovati korektnost sintetizovane strukture budući da je zasnovana na rigoroznom matematičkom aparatu. Međutim, njena primena je ograničena samo na sisteme male složenosti s obzirom na veliku složenost matematičkih izračunavanja koje podrazumeva.

Hardverska emulacija podrazumeva fizičku realizaciju prototipa koji će oponašati rad sistema koji se projektuje. Primer hardverske emulacije je implementacija dizajna u FPGA kolu radi emulacije rad ASIC kola.

***Testiranje*.** Reči ″verifikacija″ i ″testiranje″ znače približno isto, u jezičkom smislu. Međutim, u procesu razvoja digitalnih sistema, verifikacija i testiranje su dva različita zadatka. Kao što je već rečeno, verifikacija se odnosi na proveru funkcionalne korektnost i ostvarenosti zadatih performansi dizajna. S druge strane, *testiranje* je proces otkrivanja fizičkih defekta (kvarova) u gotovom proizvodu koji nastaju tokom proizvodnje. Testiranje se obavlja nakon fabrikacije proizvoda s ciljem da se dobre jedinice razdvoje od loših.

Testiranje se sprovodi tako što se na jedinicu koja se testira deluje odgovarajućom test-sekvencom (niz ulaznih kombinacija, ili test-vektora) i odziv jedinice na svaku ulaznu kombinaciju poredi sa odzivom koji se očekuje od jedinice koja radi ispravno. Međutim,

zbog ogromnog broja ulaznih kombinacija, ovakav pristup je praktično neizvodljiv. Umesto toga, projektanti koriste posebne algoritme kako bi za dato kolo kreirali što manji skup test-vektora kojim će kolo biti što potpunije testirano. Ova aktivnost je poznata kao *generisanje test-sekvenci*.

***CAD alati*.** Razvoj većih digitalnih kola i sistema zahteva upotrebu složenih postupaka projektovanja i manipulaciju velikom količinom informacija. Zato se za automatizaciju pojedinih zadataka u procesu projektovanja koristi računarski softver, tj. alati za projektovanje pomoću računara ili CAD (engl. ***C**omputer-**A**ided **D**esign*) alati. Shodno ulozi koju igraju u procesu projektovanja, savremeni CAD alati se mogu podeliti u sledećih pet kategorija: (1) alati za opis i modeliranje; (2) alati za sintezu; (3) alati za verifikaciju i simulaciju; (4) alati za raspoređivanje i povezivanje i (5) alati za generisanje test sekvenci.

U idealnom scenariju, zadatak projektanta bi bio da osmisli i kreira funkcionalni opis visokog nivoa apstrakcije, a zadatak CAD alata da automatski i na optimalan način obavi sintezu i fizičko projektovanje kompletnog sistema. Nažalost ovako nešto nije izvodljivo, ne zbog neadekvatne moći izračunavanja sadašnjih i budućih računara, već zbog kombinatorne prirode problema koji su karakteristični za sintezu hardvera i fizičko projektovanje. Bez obzira na to, CAD alati su nezamenljiva podrška projektovanju i koriste se u svim fazama projektovanja. Međutim, alat za sintezu ne može da promeni prvobitno osmišljenu arhitekturu sistema, niti može loše osmišljeni dizajn da konvertuje u dobar. Kvalitet krajnjeg rešenja prevashodno zavisi od kvaliteta inicijalnog opisa (zamisli), što u svakom slučaju predstavlja odgovornost projektanta.

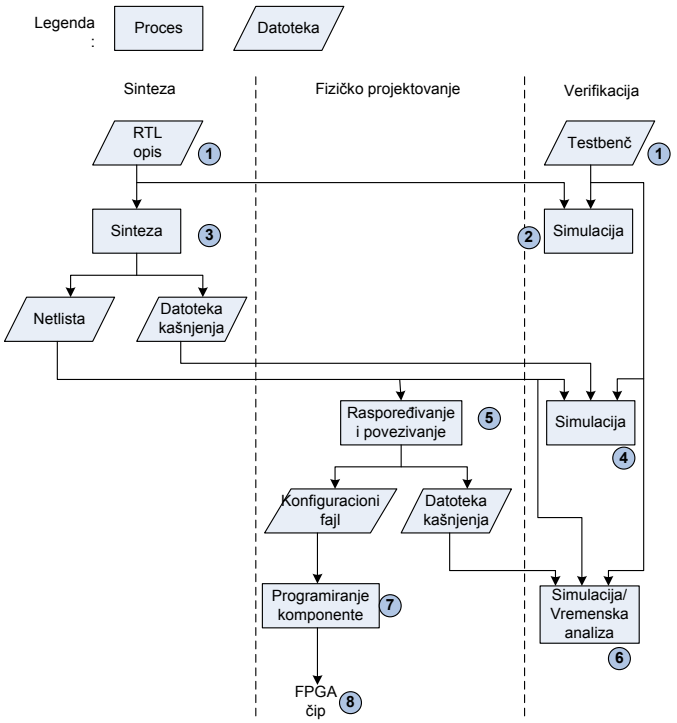
***Tok projektovanja*.** Kao što je već rečeno, projektovanje digitalnih kola i sistema predstavlja iterativni proces razrade i verifikacije, tokom kojeg se polazni apstraktni opis visokog nivoa postepeno transformiše u detaljan strukturni opis niskog nivoa. Konkretne faze i aktivnosti koje čine ovaj proces zavise kako od veličine i složenosti sistema koji se projektuje tako i od karakteristika ciljne implementacione tehnologije.

Algoritmi za optimizaciju koji se koriste u softverima za sintezu su veoma složeni. Moć izračunavanja i memorijski prostor koji su neophodni za njihovo izvršenje drastično rastu s povećanjem veličine sistema. Savremeni CAD alati za sintezu su najefikasniji za digitalne sisteme srednjeg nivoa složenosti (2000 - 50000 gejtova). Za veće sisteme, neophodno je podeliti sistema na manje blokove i nezavisno procesirati svaki blok.

Ciljna implementaciona tehnologija je drugi faktor koji utiče na tok projektovanja. Na primer, postupci implementacije dizajna u FPGA i ASIC tehnologijama se značajno razlikuju. FPGA je unapred projektovano i testirano IC. S druge strane, ASIC mora proći kroz dugotrajan i složen proces fabrikacije, što zahteva dodatne korake u projektovanju.

Ilustracije radi, u nastavku će biti predstavljen tok projektovanja digitalnog sistema srednje složenosti u FPGA tehnologiji. Projektovanje sistema srednje složenosti (do 50000 gejtova) ne zahteva prethodnu podelu sistema na manje blokove. Primeri ovakvih sistema su jednostavni mikroprocesori ili moduli za neka složenija aritmetička izračunavanja. Tok projektovanja je ilustrovan dijagramom sa Sl. 1-2. Dijagram je podeljen na tri kolone, koje odgovaraju ″pod-tokovima″ sinteze, fizičkog projektovanja i verifikacije.

Projektovanje započinje kreiranjem tzv. *dizajn datoteke*, koja najčešće sadrži funkcionalni opisa RTL nivoa u jeziku za opis hardvera (tačka 1 na Sl. 1-2). Posebna datoteka, poznata pod nazivom *testbenč*, sadrži opis virtuelnog eksperimentalnog okruženja za simulaciju i verifikaciju dizajna. Testbenč objedinjuje kôd (program) koji generiše stimulanse, model sistema koji se testira i kôd koji nadgleda i analizira odzive generisane u toku simulacije.



Sl. 1-2 Tok projektovanja sistema srednje veličine za implementaciju na FPGA.

Tok projektovanja (u idealnom slučaju) podrazumeva sledeće korake:

1. Razvoj dizajn-datoteke i testbenča.
2. Funkcionalna simulacija, čiji je cilj verifikacija polaznog RTL opisa.
3. Sinteza. Rezultat sinteze je strukturni opis na nivou gejtova. Rezultujuća logička mreža je opisana tekstualno u obliku tzv. *netliste*.
4. Datoteka koja sadrži netlistu koristi se za simulaciju i vremensku analizu, kako bi se verifikovala funkcionalna korektnost sintetizovane logičke mreže i obavila preliminarna provera tajminga.
5. Raspoređivanje i povezivanje. Gejtovi i veze iz netliste se preslikavaju na fizičke elemente i fizičke veze u FPGA kolu.
6. Ekstrakcija preciznih podataka o tajmingu. Netlista, proširena ovim podacima koristi se za simulaciju i vremensku analizu u cilju verifikacije korektnost raspoređivanja i povezivanja i provere da li kolo zadovoljava postavljena vremenska ograničenja.
7. Generisanje konfiguracione datoteke i programiranje FPGA kola.
8. Verifikacija rada fizičke komponente.

Opisani tok projektovanja odgovara idealnom scenariju, kad polazni RTL opis uspešno prolazi funkcionalnu simulaciju i sintetizovana struktura zadovoljava unapred postavljena vremenska ograničenja. U realnosti, tok projektovanja može sadržati više iteracija radi korekcije grešaka ili problema u tajmingu. To obično zahteva reviziju polaznog RTL opisa i fino podešavanje parametara u softverima za sintezu, raspoređivanje i rutiranje.

Veći i složeniji digitalni sistemi mogu sadržati nekoliko stotina ili čak nekoliko miliona gejtova. CAD alati za sintezu ne mogu efikasno da obavljaju transformacije i optimizacije nad u toj meri složenim sistemima. Zato je neophodno da se sistem podeli na manje blokove adekvatne veličine i funkcionalnosti i svaki pojedinačni blok nezavisno projektuje, pre nego što se ″konstruiše″ kompletan sistem. Vreme projektovanja se može skratiti ako se za realizaciju koriste ranije projektovani i verifikovani podsistemi ili komercijalno dostupna IP jezgara (engl. ***I**ntelectualy **P**roperty Cores*). Drugi moduli se moraju projektovati ″iz početka″, na prethodno opisani način.

1.5. Jezici za opis hardvera

Digitalni sistem se projektuje na različitim nivoima apstrakcije i u različitim domenima (funkcionalni, strukturni, fizički). U toku projektovanja, projektant je u neprekidnoj interakciji s različitim CAD alatima. Kako proces projektovanja napreduje, tako se nivoi i domeni menjaju, bilo od strane projektanta, bilo od strane CAD alata. Poželjno je da postoji jedan zajednički okvir za razmenu informacija između projektanta i CAD alata. Jezici za opis hardvera, ili HDL (engl. ***H**ardware **D**escription **L**anguage*), upravo služe ovoj svrsi. HDL treba da omogući verno i precizno modeliranje kola i sistema, bilo da se radi o postojećem kolu odnosno sistemu ili o kolu/sistemu koji se projektuje i to na željenom nivou apstrakciju, kako u funkcionalnom tako i u strukturnom domenu. HDL-ovi nalikuju po sintaksi tradicionalnim programskim jezicima. Međutim, budući da se HDL-ovi koriste za modeliranje hardvera, njihova semantika i način korišćenja se značajno razlikuju od semantike i načina korišćenja programskih jezika. VHDL i Verilog su danas dva najšire korišćena HDL-a. Iako su ova dva jezika veoma različita po sintaksi i ″izgledu″, njihove mogućnosti i oblasti primene su približno iste. Oba jezika su industrijski standardi i podržani su u većini alata za projektovanje. Prednost VHDL-a je nešto bolja podrška za parametrizovano projektovanje.

***Ograničenja programskih jezika*.** Programski jezik se karakteriše sintaksom i semantikom. Sintaksa se odnosi na gramatička pravila za pisanje programa, dok semantika definiše značenje jezičkih konstrukcija. Danas postoji veliki broj računarskih, programskih jezika, od Fortrana do C-a i Java-e. Većina programskih jezika opšte namene, kao što je C, namenjeni su za pisanje sekvencijalnih programa. Naredbe sekvencijalnog programa se obavljaju ″jedna po jedna″, po strogo sekvencijalnom redosledu. Redosled operacija se ne može proizvoljno menjati, jer svaka operacija u nizu obično zavisi od rezultata prethodnih operacija. Na apstraktnom nivou, model sekvencijalnih programa omogućava programeru da opisuje algoritam ″korak po korak″. Na implementacionom nivou, ovaj model je usklađen sa načinom rada računara i omogućava efikasno prevođenje programa u mašinske instrukcije.

Međutim, karakteristike digitalnog hardvera su značajno drugačije od karakteristika modela sekvencijalnih programa. Tipičan digitalni sistem je sačinjen od više međusobno povezanih delova. Kad god se neki signal u kolu promeni, svi delovi koji su u vezi s ovim signalom reaguju tako što izvršavaju odgovarajuću operaciju. Sve ove operacije se izvršavaju konkurentno (istovremeno, u paraleli), a trajanje svake pojedinačne operacije zavisi od iznosa propagacionog kašnjenja kroz konkretan deo. Nakon okončanja operacije, svaki deo postavlja rezultat na svoj izlaz. Ako su se izlazne vrednosti promenile, izlazni signal će aktivirati sve povezane delove i inicirati novi ciklus izvršenja operacija. Na osnovu ovog opisa, uočavaju se nekoliko jedinstvenih karakteristika digitalnih sistema, kao što su povezanost delova, konkurentne operacije i koncepti propagacionog kašnjenja i tajminga.

Sekvencijalni model, koji se koristi kod tradicionalnih programskih jezika, previše je restriktivan da bi mogao da obuhvati navedene karakteristike hardvera, što nameće potrebu za specijalizovanim jezicima koji su prevashodno namenjeni za modeliranje digitalnog hardvera.

Upotreba HDL programa. Tradicionalni programski jezici se koriste za pisanje programa koji treba da reše neki konkretan problem. Program prihvata ulazne vrednosti, sprovodi izračunavanja i generiše odgovarajuće izlazne vrednosti. Program se najpre kompajlira u mašinske instrukcije, a zatim izvršava na računaru. S druge strane, primena HDL programa je značajno drugačija. Naime, HDL programi imaju trostruku namenu:

Formalna dokumentacija. Rad na razvoju digitalnog sistema obično počinje tekstualnim opisom njegove funkcije. Nažalost, govorni jezik nije dovoljno precizan, a tekstualni opis je često nekompletan, dvosmislen i podložan različitim tumačenjima. S obzirom na to što HDL-ovi poseduju strogu sintaksu, opis sistema u obliku HDL programa je tačan i precizan. Dakle HDL program se može koristiti za formalnu specifikaciju sistema, koja se u obliku razumljive i nedvosmislene dokumentacije može razmenjivati među projektantima i projektantskim grupama.

Simulacija. Kao što je napomenuto u prethodnom poglavlju, simulacija se koristi za analizu i verifikaciju rada sistema pre nego što se on i fizički realizuje. U suštini, HDL simulator predstavlja okvir za izvršenje konkurentnih operacija na sekvencijalnom računaru. Simulator za konkretan HDL u stanju je da tumači semantičko značenje jezičkih konstrukcija i interpretira program napisan u tom HDL-u. U toku izvršenja, simulator interpretira HDL program i generiše odgovarajuće odzive.

Sinteza. Savremeni tok projektovanja zasnovan je procesu razrade kojim se funkcionalni opis visokog nivoa postepeno transformiše u strukturni opis niskog nivoa. Pojedini koraci razrade se mogu obaviti uz pomoć CAD alata za sintezu. Softver za sintezu, kao ulaz prihvata HDL program i generiše odgovarajuće kolo sačinjeno od komponenti iz raspoložive biblioteke komponenti. Tipično, izlaz iz alata za sintezu je novi HDL program, koji sada predstavlja strukturni opis sintetizovanog kola.

Karakteristike savremenih HDL-ova. Fundamentalne karakteristike digitalnih kola i sistema se definišu konceptima: entitet, povezanost, konkurentnost i tajming. *Entitet* je bazični gradivni blok, modeliran po ugledu na realno kolo. Entitet je samodovoljan i nezavisan i ne poseduje bilo kakve informacije o drugim entitetima. *Povezanost* modelira veze koje povezuju delove sistema i omogućava interakciju između entiteta. Tipično, više entiteta su aktivni u isto vreme, a mnoge operacije se izvršavaju u paraleli. *Konkurentnost* opisuje ovakav način ponašanja. *Tajming* je u vezi sa konkurentnošću; definiše početak i završetak svake operacije, kao i redosled izvršenja operacija.

Kao što je opisano u prethodnom poglavlju, digitalni sistem se može modelirati na četiri različita nivoa apstraktnosti i u tri različita domena. Idealno bi bilo posedovati HDL koji bi pokrio sve ove nivoe i domene. Međutim, to je teško ostvarljivo zbog velikih razlika koje postoje između različitih nivoa apstrakcije, kao i između različitih domena. Savremeni HDL-ovi omogućavaju kreiranje opisa u strukturnom i bihejvoralnom (funkcionalnom) domenu, ali ne i u fizičkom. Takođe, oni pružaju podršku za modeliranje na nivou gejtova i RTL nivou i u izvesnoj meri na procesorskom i tranzistorskom nivou. Zajedničke karakteristike savremenih HDL-ova su:

- Semantika jezika pruža podršku za koncepte: entitet, povezanost, konkurentnost i tajming.

- U jeziku se mogu predstaviti propagaciona kašnjenja i informacije o tajmingu
- Jezik poseduje konstrukcije kojima se na direktan način može opisati strukturna implementacija kola (npr. blok dijagram).
- Jezik sadrži konstrukcije kojim se na bihejvioralni način može opisati rad kola, uključujući i konstrukcije koje nalikuju onim iz tradicionalnih programskih jezika, kao što su *if-then-else*, *for*, *while*, itd.
- Jezik može efikasno da modelira operacije i strukture na gejtovskom i RTL nivou.
- Jezik poseduje konstrukcije za hijerarhijsko i parametrizovano projektovanje.

2. UVOD U VHDL

VHDL je skraćenica od "***VHSIC Hardware Description Language***" (u prevodu, VHSIC jezik za opis hardvera) gde je VHSIC takođe skraćenica od "***Very High Speed Integrated Circuits***" (u prevodu, integrisana kola veoma velike brzine rada). Prvobitan VHDL standard usvojen je 1987. godine od strane međunarodne organizacije za standardizaciju IEEE (***I**nstitute of **E**lectrical and **E**lectronics **E**ngineers*) pod oznakom IEEE 1076. Revidiran i trenutno aktuelan VHDL standard, IEEE 1164, usvojen je 1993. godine.

Iako prevashodno namenjen za modeliranje hardvera, VHDL je po mnogim svojim karakteristikama sličan konvencionalnim, računarskim programskim jezicima, kao što je to npr. C. U VHDL-u, kao u C-u, postoje promenljive, programska grananja, petlje, funkcije itd. Međutim, između VHDL-a i programskih jezika postoje i značajne razlike, uslovljene njihovom specifičnom namenom. Program napisan na programskom jeziku se kompilira (uz pomoć kompajlera) i tako prevodi u mašinski kôd, koji se može direktno *izvršiti* na računaru opšte namene. S druge strane, VHDL kôd je moguće *sintetizovati*, pomoću specijalizovanih softvera za hardversku sintezu, odnosno prevesti ga u oblik netliste na osnovu koje je moguće automatski realizovati fizički sistem. Takođe, VHDL opis nekog sistema se može funkcionalno i vremenski simulirati, uz pomoć HDL simulatora, a u cilju verifikacije projekta pre njegove fizičke realizacije.

Prvobitno, VHDL je imao dvostruku primenu. Prvo, bio je korišćen kao dokumentacioni jezik za precizno i unificirano opisivanje strukture složenih digitalnih sistem s ciljem da se olakša razmena projekata između projektantskih organizacija. Drugo, budući da poseduje jezičke konstrukcije za modeliranje ponašanja digitalnih kola, VHDL je korišćen i kao ulaz u softvere koji su se u to vreme koristili za simulaciju rada digitalnih kola. Počev od sredine 90' godina prošlog veka, pored korišćenja za dokumentovanje projekta i simulaciju, VHDL je takođe postao popularno sredstvo za unos dizajna u CAD sisteme za sintezu hardvera. Jednom napisan i verifikovan VHDL kôd može se iskoristiti za realizaciju opisanog kola u različitim implementacionim tehnologijama, kao što su PLD ili ASIC kola.

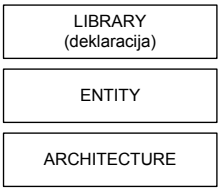
VHDL je izuzetno obiman i složen jezik. Učenje svih detalja ovog jezika zahteva dosta napora i vremena, kao i dobro poznavanje naprednijih tehnika digitalnog projektovanja. Međutim, za osnovnu upotrebu u sintezi, bitan je samo jedan manji deo mogućnosti VHDL jezika. U ovoj knjizi biće razmatrani samo oni aspekti VHDL-a koji su od ključne važnosti za sintezu hardvera.

Napomenimo da je za razliku od programskih jezika, koji su *sekvencijalni*, VHDL u osnovi *konkurentan* (paralelan) jezik. Da bi se naglasila ova razlika, uobičajeno je da se u kontekstu VHDL-a govori o *kôdu* ili o *opisu*, a ne o *programu*, kao što je to slučaj kod programskih jezika.

2.1. Organizacija VHDL kôda

Svaki celovit VHDL opis sastoji se iz sledeće tri sekcije (Sl. 2-1):

- Deklaracija LIBRARY: sadrži spisak biblioteka i/ili delova biblioteka koji se koriste u projektu, npr. *ieee*, *std*, *work* itd.
- ENTITY: definiše ulazne i izlazne signale (pinove ili portove) kola.
- ARCHITECTURE: sadrži *pravi* VHDL kôd koji opisuje ponašanje (tj. funkciju) ili unutrašnju organizaciju (tj. strukturu) kola.

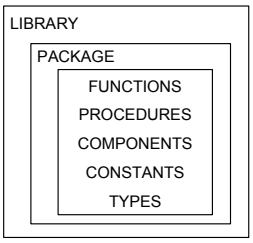


Sl. 2-1 Struktura VHDL kôda.

2.1.1. LIBRARY

Library (u prevodu, biblioteka) je kolekcija često korišćenih delova VHDL kôda. Biblioteka se sastoji iz jednog ili više paketa (*package*) koji sadrže funkcije (*functions*), procedure (*procedures*), komponente (*components*), konstante (*constants*) i definicije tipova podataka (*types*) (Sl. 2-2). Jednom kreirana, biblioteka se može koristiti u drugim projektima ili razmenjivati s drugim projektantima.

Biblioteke *std* i *work* su sastavni deo jezika VHDL. Biblioteka *std* sadrži definicije standardnih tipova podataka, dok je *work* tzv. podrazumevana radna biblioteka, tj. biblioteka u koju će nakon kompajliranja biti smeštene sve programske celine sadržane u konkretnom VHDL projektu.



Sl. 2-2 Struktura biblioteke.

Da bi se u VHDL projektu omogućio pristup sadržaju neke konkretne biblioteke, potrebne su dve linije kôda sledeće sintakse:

```
LIBRARY ime_biblioteke;
USE ime_biblioteke.ime_paketa.delovi_paketa;
```

Prva linija definiše ime biblioteke, dok druga, koja sadrži naredbu USE, definiše pakete i delove pakete iz izabrane biblioteke koji se žele koristiti. U svakom VHDL projektu koji je namenjen sintezi neophodna su barem tri paketa iz tri različite biblioteke:

- paket *std_logic_1164* iz biblioteke *ieee*
- paket *standard* iz biblioteke *std* i
- paket *work* iz biblioteke *work*

Odgovarajuće deklaracije su:

```
LIBRARY IEEE;                       -- tačka-zarez (;) označava
USE IEEE.STD_LOGIC_1164.ALL;        -- kraj naredbe ili deklaracije
LIBRARY STD;                        -- dupla crta označava komentar
USE STD.STANDARD.ALL;
LIBRARY WORK;
USE WORK.ALL;
```

Reč *all* na kraju naredbe *use* ukazuje na to da se koristi celokupan sadržaj navedenog paketa. Biblioteke *std* i *work* su po automatizmu uključene u svaki projekat, pa ih nije neophodno eksplicitno deklarisati. Međutim, to nije slučaj s bibliotekom *ieee*, pa njena deklaracija mora uvek da bude navedena. Treba napomenuti da je biblioteka *ieee* neophodna samo ako se u projektu koristi tip podataka *std_logic*. U suprotnom, i ona se može izostaviti. Ali, budući da se signali i varijable tipa *std_logic* često koriste u sintezi, VHDL kôd namenjen sintezi po pravilu počinje sledećim dvema linijama:

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
```

Međunarodna organizacija za standardizaciju IEEE razvila je i standardizovala nekoliko paketa s ciljem da olakša i unificira pisanje VHDL kôda za sintezu. Svi ovi paketi su sastavni deo biblioteke *ieee*. Pored paketa *std_logic_vector_1164*, najznačajniji paket iz ove biblioteke je *numeric_std*, koji se koristi za modeliranje aritmetičkih kola. Detaljniji opis navedenih paketa biće dat u poglavlju 3.3.

2.1.2. ENTITY

Sekcija *entity* definiše ime kola koje se projektuje i imena i osnovne karakteristike njegovih ulaza i izlaza, tzv. pinova, odnosno *portova* (*ports*). Sintaksa deklaracije entiteta je sledećeg oblika:

```
ENTITY ime_entiteta IS
PORT (
  ime_porta : smer_signala tip_signala;
  ime_porta : smer_signala tip_signala;
  . . . );
END ime_entiteta;
```

Ime entiteta (tj. ime kola koje se opisuje) se navodi u prvoj i poslednoj liniji sekcije *entity*. Deklaracija portova se nalazi u delu ove sekcije koji počinje službenom rečju *port*. Za svaki port se navodi njegovo ime, smer i tip. Posredstvom portova kolo razmenjuje signale sa drugim kolima (entitetima) iz svog okruženja. Portovi ne moraju biti samo binarni signali (tipa *bit*), već to mogu biti signali bilo kog drugog tipa koji je podržan od strane VHDL-a (npr. *std_logic*, *integer* itd.). U kôdu za sintezu najčešće se koristi tip *std_logic* koji omogućava da signali pored ´0´ i ´1´ mogu imati još nekoliko različitih vrednosti (ili

nivoa). Pregled osnovnih tipova podataka u VHDL-u dat je u glavi 3. Smer porta može biti: *in*, *out*, *inout* ili *buffer* (Sl. 2-3(a)). Smer *in* označava ulazne, a *out* izlazne portove kola. Smer *inout* se koristi za deklaraciju dvosmernih (ili bidirekcionih) portova, tj. portova koji se po potrebi mogu koristiti bilo kao ulazi, bilo kao izlazi. Smer *buffer* označava izlazne portove koji se mogu koristiti i kao interni signali unutar samog kola.

(a) (b)

Sl. 2-3 (a) Smer porta; (b) NI kolo.

Pr. 2-1 Entitet dvoulaznog NI kola

NI kolu sa Sl. 2-3(b) odgovara entitet:

```
ENTITY ni_kolo IS
  PORT (a, b : IN BIT;
        c : OUT BIT);
END ni_kolo;
```

Značenje gornje deklaracije entiteta je sledeće. Opisuje se kolo koje ima dva ulazna, *a* i *b*, i jedan izlazni port, *c*. Sva tri porta su tipa *bit*. Kolu je dato ime *ni_kolo*. Dakle, *entity* ne govori ništa o funkciji kola, već samo opisuje spoljni pogled na kolo, tj. **interfejs** kola. Treba obratiti pažnju na sledeće pojedinosti: više portova istog smera i tipa mogu biti deklarisani u istoj liniji (kao što je to slučaj sa portovima *a* i *b*). Svaka linija iz konstrukcije *port* završava se znakom ´;´, osim poslednje, posle koje sledi zagrada koja zatvara deklaraciju portova.

2.1.3. ARHITECTURE

Da bi VHDL opis nekog kola bio potpun, osim entiteta, koji definiše ulaze i izlaze kola, neophodno je napisati još jednu programsku celinu koja se zove arhitektura (*architecture*), a koja će sadržati opis funkcionisanja (*ponašanja*) ili opis unutrašnje strukture kola. Njena sintaksa je sledeća:

```
ARCHITECTURE ime_arhitekture OF ime_entiteta IS
 [deklaracije]
BEGIN
 [kôd]
END ime_arhitekture;
```

Kao i za entitet, ime arhitekture može biti proizvoljno izabrano. Arhitektura je uvek pridružena jednom konkretnom entitetu, čije se ime navodi posle službene reči *of*. Sekcija *architecture* se sastoji iz dva dela: deklarativnog (koji je opcion) gde se, između ostalog, deklarišu interni signali i konstante i dela za kôd (počev od službene reči *begin*, pa do kraja). Za pisanje kôda arhitekture koriste se tzv. *konkurentne* naredbe.

Pr. 2-2 Arhitektura dvoulaznog NI kola

Razmotrimo ponovo NI kolo sa Sl. 2-3(b). Funkcionisanje ovog kola se može opisati sledećom arhitekturom:

```
ARCHITECTURE ni_funkcija OF ni_kolo IS
BEGIN
  c <= a NAND b;
END ni_funkcija;
```

Gornji kôd se tumači na sledeći način. Kolo obavlja NI operaciju (*nand*) nad dva ulazna signala, *a* i *b*, i rezultat dodeljuje (″<=″) izlaznom signalu *c*. U ovom primeru ne postoji deklarativni deo arhitekture, a kôd sadrži samo jednu naredbu dodele. Treba razumeti da se ova naredba ne izvršava samo jedanput, već uvek kad se na nekom od signala *a* ili *b* desi događaj. *Događaj na signalu* znači da se desila promena vrednosti signala. Kaže se da je naredba dodele osetljiva (ili *senzitivna*) na promenu vrednosti bilo kog signala s desne strane znaka ″<=″. U konkretnom primeru, naredba je senzitivna na signale *a* i *b*.

Iako je svaka arhitektura pridružena tačno jednom entitetu, entitetu se, s druge strane, može pridružiti jedna od eventualno nekoliko raspoloživih, različitih arhitektura, od kojih svaka na neki specifičan način opisuje isto kolo. Na primer, jedan, apstraktniji opis, može se koristiti za simulaciju, a drugi, detaljniji, za sintezu. Spoj entiteta i arhitekture se naziva *konfiguracijom* (v. 2.5).

2.2. Stilovi projektovanja

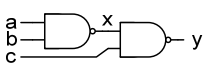
Digitalni sistem se može opisati u VHDL-u na dva načina: funkcionalno ili strukturno. Tehnika (ili stil) programiranja u VHDL-u kojom se opisuje funkcija kola naziva se *bihejvioralno* ili *funkcionalno* modeliranje. Osnovu funkcionalnog modeliranja čine konkurentne i sekvencijalne naredbe. Međutim, nekada je lakše, efikasnije ili čak neophodno digitalni sistem opisati na strukturan način. Za razliku od funkcionalnog modeliranja, koje je usredsređeno na funkciju kola, *strukturno* modeliranje opisuje složenije kolo kao skup povezanih kola manje složenosti. Takođe, podržana je, i često se koristi, kombinacija stilova. Nije neuobičajeno da su u istoj arhitekturu sadržane sekcije konkurentnog, sekvencijalnog i strukturnog kôda.

U ovom poglavlju biće predstavljeno nekoliko jednostavnih primera VHDL opisa. Iako još uvek nismo proučili sve jezičke konstrukcije koje će se pojaviti u primerima koji slede, oni će nam pomoći da bolje razumemo osnovne principe funkcionalnog i strukturnog modeliranja u VHDL-u.

2.2.1. Funkcionalni opis

***Konkurentne naredbe*.** Za razliku od standardnih programskih jezika koji su sekvencijalni (naredbe se izvršavaju jedna za drugom), VHDL je u osnovi konkurentan jezik. To znači da prilikom izvršenja VHDL kôda, simulator pravi iluziju istovremenog (konkurentnog) izvršavanja različitih delova kôda. Ovakav jedan model je pogodan za opis hardvera, gde različiti podsistemi jednog složenog sistema rade u paraleli.

Pr. 2-3 Konkurentno izvršenje naredbi



Sl. 2-4 Jednostavna kombinaciona mreža.

Ispod je dat VHDL opis kombinacione mreže sa Sl. 2-4. Interfejs ove mreže čine tri ulazna, *a*, *b* i *c*, i jedan izlazni port, *y*. Osim signala koji odgovaraju portovima, u kôdu arhitekture se koristi još jedan interni (unutrašnji) signal, *x*, koji odgovara istoimenoj vezi u mreži sa Sl. 2-4 kojom se izlaz prvog povezuje sa ulazom drugog NI kola. Interni signali se deklarišu u deklarativnom delu arhitekture (linija 8). Primetimo da deklaracija signala *x* ne sadrži definiciju smera (smer se definiše samo za portove). Primetimo i to da dati kôd ne sadrži deklaraciju biblioteka. Razlog za to je što su u ovom primeru svi signali (portovi plus interni signal) tipa *bit*. Ovaj tip podataka je definisan u biblioteci *std*, koja je po automatizmu uključena u svaki VHDL projekat. U kôdu arhitekture, prva naredba dodele (linija 10) postavlja izlaz mreže, *y*, u zavisnosti od vrednosti signala *x* i *c*, a druga (linija 11) signal *x* u zavisnosti od *a* i *b*.

```
1  ----------------------------------------------------------------
2  ENTITY nand_mreza IS
3    PORT(a, b, c: IN BIT;
4         y : OUT BIT);
5  END nand_mreza;
6  ----------------------------------------------------------------
7  ARCHITECTURE dataflow OF nand_mreza IS
8     SIGNAL x : BIT;
9  BEGIN
10    y <= x NAND c;
11    x <= a NAND b;
12 END dataflow;
13 ----------------------------------------------------------------
```

Izvršenje naredbe u konkurentnom kôdu inicirano je pojavom događaja na nekom signalu s desne strane znaka ″<=″, a ne redosledom po kome su naredbe napisane, kao što je to slučaj kod sekvencijalnih programskih jezika. Zbog toga, redosled naredbi u konkurentnom kôdu nije od značaja. Identičnu funkciju bi imao i kôd u kome bi dve naredbe iz gornje arhitekture zamenile mesta:

```
x <= a NAND b;
y <= x NAND c;
```

Pretpostavimo da su ulazi mreže sa Sl. 2-4 postavljeni na *a*=*b*=*c*=´1´. S ovakvom ulaznom pobudom, na izlazu mreže je *y* = (´1´ *nand* ´1´) *nand* ´1´ = ´1´. Neka se u nekom momentu vrednost ulaza *a* promeni na *a*=´0´. Ovaj događaj utiče na naredbu iz linije 11 (*x* <= *a nand b*), zato što je ona senzitivna na signal *a*, ali ne i na naredbu iz linije 10 (*y* <= *x nand c*), jer kod nje signal *a* ne postoji kao operand. Kao rezultat izvršenja naredbe iz linije 11, signal *x* dobija novu vrednost *x*= (´0´ *nand* ´1´)= ´1´. Ova promena vrednosti signala *x* ima za posledicu aktiviranje naredbe iz linije 10, nakon čijeg izvršenja izlaz *y* dobija vrednost *y* = ´1´ *nand* ´1´ = ´0´. Ovakav način izvršenja naredbi se naziva modelom *toka podataka* (eng. *dataflow*) budući da je redosled izvršenja naredbi uslovljen propagacijom (protokom) ulazne promene kroz naredbe kôda.

Konkurentne naredbe nalikuju elementima nekog digitalnog kola koji rade u paraleli. Signal s leve strane naredbe dodele predstavlja izlaz, signali s desne strane ulaze, a operacija funkciju elementa.

Pr. 2-4 Modeliranje propagacionog kašnjenja

Propagaciono kašnjenje predstavlja jednu od ključnih osobina digitalnih kola, a odnosi se

na vreme koje protekne od trenutka promene vrednosti ulaznog signala do trenutka odgovarajuće promene vrednosti izlaznog signala. Ispod je dat VHDL opis kombinacione mreže sa Sl. 2-4 koji osim funkcije modelira i propagaciono kašnjenje kroz pojedine delove kola. Dve konkurentne naredbe dodele su sada proširene klauzulom **after** posle koje je naveden iznos kašnjenja u nanosekundama. S ovim proširenjem, naredbe dodele se izvršava, kao ranije, uvek kad se desi događaj na nekom signalu s desne stane znaka ″<=″, ali sada je rezultat dostupan, tj. signal sa desne strane znaka ″<=″ dobija novu vrednost, tek nakon navedenog iznosa propagacionog kašnjenja. Dakle ako pretpostavimo da su ulazni signali mreže sa Sl. 2-4 postavljeni na a=b=c=´1´ i da se u trenutku t = 0 ns signal a promeni na ´0´, signal x će dobiti vrednost x = ´1´ u trenutku t = 20 ns. Ova promena pokrenuće naredbu y <= x *nand* c, čiji se rezultat ´1´ = ´0´ *nand* ´1´ prenosi na signal y, ali ne trenutno već tek nakon 20 ns, tj. u trenutku t = 40 ns.

```
1  ----------------------------------------------------------------
2  ENTITY nand_mreza IS
3    PORT(a, b, c: IN BIT;
4         y : OUT BIT);
5  END simple_circuit;
6  ----------------------------------------------------------------
7  ARCHITECTURE dataflow OF nand_mreza IS
8     SIGNAL x : BIT;
9  BEGIN
10    y <= x NAND c after 20 ns;
11    x <= a NAND b after 20 ns;
12 END dataflow;
13 ----------------------------------------------------------------
```

Modeliranje propagacionog kašnjenja je bitno kad se VHDL opis kreira u cilju simulacije. U kôdu za sintezu, navođenje propagacionog kašnjenja nije dozvoljeno. Međutim, čak iako u naredbi dodele propagaciono kašnjenje nije eksplicitno navedeno, ono se podrazumeva, ali ima beskonačno malu vrednost, tzv. *delata-kašnjenje*. Delta-kašnjenje je veće od nule, ali manje od bilo kog konačnog fizičkog kašnjenja. Drugim rečima, naredba:

```
x <= a NAND b;
```

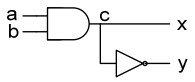
je identična naredbi:

```
x <= a NAND b AFTER 0 ns;
```

Pr. 2-5 Port smera OUT se ne može koristiti kao ulazni signal!

Portovi su signali koji se mogu koristiti u arhitekturi ravnopravno s internim signalima, uz poštovanje smera porta. Pri tom, ulazni port (smer *in*) se može naći u naredbi dodele samo s desne, a izlazni port (smer *out*) samo s leve strane znaka ″<=″. Nepoštovanje ovih ograničenja predstavlja sintaksnu grešku. Razmotrimo kombinacionu mrežu sa Sl. 2-5(a). Težnja ka što kompaktnijim opisom nas može lako dovesti do pogrešnog kôda sa Sl. 2-5(b). Arhitekturu čine dve naredbe dodele koje striktno prate strukturu mreže: x je ″*a and b*″, a y komplement od x. Međutim, greška postoji u liniji 12 gde se x, deklarisan kao izlazni port (u liniji 7), koristi kao ulazni signal (naveden je s desne strane znaka ″<=″). Ova greška se može ispraviti ako se u entitetu port x deklariše sa smerom *buffer*. Međutim, korišćenje smera *buffer* se ne preporučuje, jer može dovesti do suptilnih grešaka koje se teško otkrivaju. Druga mogućnost je da se port x deklariše sa smerom *inout*. Međutim, ni ovo nije dobro rešenje, jer x nije bidirekcioni već izlazni signal. Rešenje koje se preporučuje je ono

koje je dato na Sl. 2-5(c). U ovom rešenju, uveden je pomoćni, interni signal *c* koji se u liniji 5 koristi za čuvanje međurezultata *"a and b"*, a na osnovu kojeg se formiraju oba izlaza: *x* je isto što i *c* (linija 6), dok je *y* komplement od *c* (linija 7). Kôd sa Sl. 2-5(c) je ispravan jer se, za razliku od *in* i *out* portova, interni signali u naredbama dodele mogu koristiti bilo kao ulazi, bilo kao izlazi. Takođe, uočimo da su svi signali u ovom primeru tipa *std_logic*. Kao što je već rečeno, ovaj tip podataka se standardno koristi u kôdu za sintezu za predstavljanje binarnih signala.



(a)

```
1  ------------------------------
2  LIBRARY IEEE;
3  IEEE.STD_LOGIC_1164.ALL;
4  ------------------------------
5  ENTITY pr5 IS
6    PORT(a,b : IN STD_LOGIC;
7         x,y : OUT STD_LOGIC);
8  END r5;
9  ARCHITECTURE pogresno OF pr5 IS
10 BEGIN
11   x <= a AND b;
12   y <= NOT x;
13 END pogresno;
14 ------------------------------
```

(b)

```
1 ------------------------------
2 ARCHITECTURE ispravno OF pr5 IS
3   SIGNAL c : STD_LOGIC;
4 BEGIN
5   c <= a AND b;
6   x <= c;
7   y <= NOT c;
8 END ispravno;
9 ------------------------------
```

(c)

Sl. 2-5 Portovi kao signali: (a) primer kombinacione mreže sa dva izlaza; (b) pogrešan VHDL opis; (c) ispravan VHDL opis.

Pr. 2-6 Opis tabele istinitosti

Iako VHDL opisi iz prethodnih primera striktno prate strukturu polaznih logičkih mreža, u opštem slučaju to ne mora biti tako. VHDL, pre svega, omogućava modeliranje (opisivanje) funkcije ili ponašanja kola bez ulaženja u strukturne detalje. Na primer, funkcija mreže sa Sl. 2-4 može se predstaviti u obliku tabele istinitosti sa Sl. 2-6(a). Odgovarajući VHDL opis prikazan je na Sl. 2-6(b).

a	b	c	y
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

(a)

```
ARCHITECTURE funct OF nand_mreza IS
BEGIN
  WITH (a & b & c) SELECT
    y <= '1' WHEN "000",
         '0' WHEN "001",
         '1' WHEN "010",
         '0' WHEN "011",
         '1' WHEN "100",
         '0' WHEN "101",
         '1' WHEN "110",
         '1' WHEN "111";
END funct;
```

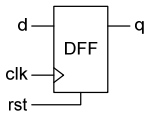
(b)

Sl. 2-6 Opis tabele istinitosti: (a) tabela istinitosti funkcije tri promenljive; (b) VHDL opis.

Kao što se može videti, sada kôd nema direktne veze sa strukturom polazne mreže, već samo opisuje njenu funkciju. Naredba *select* je senzitivna na sva tri ulaza, *a*, *b* i *c*, koji su operatorom & povezana u trobitni vektor. Naredba *select* se jedanput izvrši uvek kad se desi događaj na bilo kom od ovih signala, a izlazu *y* se dodeli vrednost iz *when* grane koja odgovara trenutnoj vrednosti trobitnog ulaza. Osim naredbe dodele i naredbe *select*, u VHDL-u postoji još nekoliko konkurentnih naredbi. Konkurentnim naredbama je posvećena glava 4.

Sekvencijalne naredbe. Konkurentne naredbe su pogodne za opis hardvera gde različiti podsistemi jednog složenog sistema rade u paraleli. Međutim, ponašanje pojedinih podsistema, ili kola, često je lakše opisati sekvencijalnim kôdom. Konstrukcija *process* predstavlja jedan od načina kako se u VHDL-u može nametnuti sekvencijalno izvršenje naredbi. Naime, naredbe obuhvaćene procesom izvršavaju se sekvencijalno (jedna za drugom), kao kod bilo kog programskog jezika. Drugim rečima, kao što arhitektura predstavlja okvir za konkurentni, tako proces predstavlja okvir za sekvencijalni kôd.

Pr. 2-7 Sekvencijalni proces - D flip-flop sa sinhronim resetom



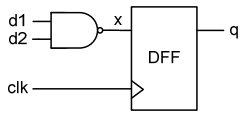
Sl. 2-7 D flip-flop sa sinhronim resetom.

Na Sl. 2-7 je prikazan grafički simbol D flip-flopa sa okidanjem na rastuću ivicu taktnog signala (*clk*) i sinhronim resetom (*rst*). Filip-flop je bistabilno kolo, tj. kolo sa dva stabilna stanja: stanje 0 ili resetovano stanje i stanje 1 ili setovano stanje. Trenutno stanje flip-flopa dostupno je na izlazu *q*. D flip-flop ostaje u zatečenom stanju sve do trenutka delovanja aktivne ivice taktnog signala (u konkretnom primeru aktivna je rastuća ivica takta), a onda prelazi u novo stanje shodno trenutnim vrednostima ulaza *d* i *rst*. Ako je *rst*=´1´, novo stanje flip-flopa biće ´0´ (kaže se, flip-flop se *resetuje*). Ako je *rst* = ´0´, novo stanje flip-flopa biće identično vrednosti ulaznog signala *d* (kaže se, u flip-flop se *upisuje* vrednost sa ulaza *d*). Zapazimo da promene signala *d* i *rst*, same po sebi, ne utiču na flip-flop, već je njihova vrednost od značaja samo u trenucima delovanja rastuće ivice takta.

U VHDL kôdu koji sledi, D flip-flop je opisan u vidu procesa. Kao što su konkurentne naredbe dodele osetljive na primenu signala s desne strane znaka dodele, tako je proces osetljiv na promenu signala iz njegove liste senzitivnosti, koja, u konkretnom primeru sadrži samo jedan signal, *clk* (linija 11). Svaka promena ovog signala ″pokreće″ proces. Naredbe sadržane u procesu izvršavaju se na sekvencijalan način, tj. jedna za drugom. Kad se izvrši i poslednja sekvencijalna naredba, proces se deaktivira i prelazi u stanje čekanja na novu promenu signala *clk*. Kôd procesa čine dve ugnježdene *if* naredbe. Spoljna *if* naredba detektuje rastuću ivicu signala *clk*, a unutrašnja ispituje vrednost signala *rst*. Uslov za rastuću ivicu signala *clk* (linija 13) je kombinacija dva uslova: signal *clk* se promenio (*clk'event*) i njegova nova vrednost je *clk* = ´1´ . Ako u momentu rastuće ivice signala *clk*, signal *rst* ima vrednost ´1´ novo stanje flip-flopa je ´0´ (linija 15), inače u flip-flop se upisuje vrednost ulaza *d* (linija 17). Signal *q* zadržava (memoriše) dodeljenu vrednost sve do sledeće rastuće ivice signala *clk*, kad se na osnovu vrednosti ulaza *rst* i *d* ponovo određuje njegova nova vrednost. Napomenimo da se procesi u VHDL-u koriste za modeliranje sekvencijalnih digitalnih kola, tj. kola s memorijom, kao što su to flip-flopovi, registri, brojači i konačni automati. Procesi i sekvencijalno programiranje u VHDL-u obrađeni su u glavi 5.

```
1  ------------------------------------------------------------------
2  LIBRARY IEEE;
2  USE IEEE.STD_LOGIC_1164.ALL;
3  ------------------------------------------------------------------
4  ENTITY dff IS
5     PORT (d, clk, rst: IN STD_LOGIC;
6           q: OUT STD_LOGIC);
7  END dff;
8  ------------------------------------------------------------------
9  ARCHITECTURE behavior OF dff IS
10 BEGIN
11   PROCESS (clk)
12   BEGIN
13    IF(clk'EVENT AND clk='1') THEN
14       IF(rst='1') THEN
15         q <= '0';
16       ELSE
17         q <= d;
18       END IF;
19    END IF;
20   END PROCESS;
21 END behavior;
```

Pr. 2-8 D flip-flop i NI kolo – proces i konkurentna naredba dodele



Sl. 2-8 D flip-flop i NI kolo.

VHDL opisi NI kolo sa Sl. 2-3(b) i kombinacione mreže sa Sl. 2-4, s jedne, i D flip-flopa sa Sl. 2-7 s druge strane primeri su čisto konkurentnog i čisto sekvencijalnog opisa, respektivno. VHDL opis kola sa Sl. 2-8 (dat u ovom primeru) predstavlja kombinaciju ova dva osnovna stila funkcionalnog modeliranja. Kôd arhitekture (linije 10 – 15) sadrži jedan procesa koji opisuje D flip-flop (linije 11 – 15) i jednu konkurentnu naredbu dodele koja obavlja NI operaciju (linija 10). Iako se naredbe unutar procesa izvršavaju sekvencijalno, proces kao celina je konkurentan u odnosu na druge naredbe sadržane u istoj arhitekturi. Tako, možemo smatrati da se naredba dodele (linija 10) i proces (linije 11 - 15) izvršavaju konkurentno, slično kao što bi u fizičkom kolu, NI kolo i D flip-flop radili istovremeno i nezavisno. Interni signal *x* se koristi za spregu ove dve sekcije kôda. Uočimo da za razliku od D flip-flopa iz Pr. 2-7, D flip-flop iz ovog primera ne poseduje funkciju resetovanja. Proces se pokreće pri svakoj promeni signala *clk*. Ako ova promena odgovara rastućoj ivici, signalu *q* se dodeljuje vrednost signala *x* (linija 13), što odgovara upisu u flip-flop. Primetimo da konkurentna naredba dodele iz linije 10 nije neophodna i da se može izostaviti ako se u procesu naredba *q* <= *x* (linija 13) zameni naredbom *q* <= *d1 nand d2*.

```
1  ------------------------------------------------------------------
2  ENTITY pr8 IS
3     PORT(d1, d2, clk: IN BIT;
4          q : OUT BIT);
5  END pr8;
6  ------------------------------------------------------------------
```

```
7  ARCHITECTURE primer OF pr8IS
8     SIGNAL x : BIT;
9  BEGIN
10    x <= d1 NAND d2;
11    PROCESS(clk)
12    BEGIN
13       IF(clk'EVENT AND clk='1') THEN q <= x;
14       END IF;
15    END PROCESS;
16 END primer;
```

Svi primeri iz ovog poglavlja sadrže funkcionalne VHDL modele. To se odnosi i na primere kod kojih se lako može uspostaviti veza između naredbi i logičkih kola. Međutim, operatori *and*, *not*, *nand* itd. nisu logička kola, već su to logičke operacije koje, kao i složenije konstrukcije poput naredbi *select* i *if*, služe za opisivanje funkcije. Pored operacija i naredbi viđenih u prethodnim primerima, VHDL poseduje i brojne druge operacije i naredbe koje se mogu koristiti za funkcionalno opisivanje, a kojima će biti posvećeno nekoliko narednih glava ove knjige.

2.2.2. Strukturni opis

Strukturno modeliranje u VHDL-u zasnovano je na konceptu komponenti. Komponenta je celoviti VHDL opisi (entitet plus arhitektura) koji se jednom piše, a onda koristi za konstrukciju složenijih modela. Tipično, više komponenti se pakuje u jedan paket; paket se smešta u biblioteku (Sl. 2-2), a biblioteka uključuje u one delove projekata gde želimo da koristimo ranije projektovane komponente.

Pr. 2-9 D flip-flop i NI kolo – strukturni opis

Ispod je dat strukturni opis kola sa Sl. 2-8. U kôdu se koriste dva komponente, *ni_kolo* i *dff*. Pretpostavka je da se funkcionalni opisi ovih komponenti nalaze u paketu *nas_paket* iz biblioteke *nasa_biblioteka*. U samom kôdu, najpre su uključeni biblioteka i paket (linije 3 i 5), zatim su komponente koje želimo da koristimo deklarisane u deklarativnom delu arhitekture (linije 13-22) i konačno *instancirane* i na odgovarajući način povezane u telu arhitekture (linije 25 i 26). Deklaracija komponente, slično entitetu, definiše ulazne i izlazne portove komponente. Naredba iz linije 25 instancira (kreira) jedan primerak komponente *ni_kolo* i povezuje ga sa portovima i internim signalima kola koje se projektuje. Ime ove instance je *K*1. Konstrukcija *port map* povezuje spoljne portove *d*1 i *d*2 i interni signal *x* na portove *a*, *b* i *c* instance *K*1 komponente *ni_kolo*, respektivno. Slično, naredba iz linije 26 kreira jedan primerak D flip-flopa *dff* i njegove portove *d*, *clk* i *q* povezuje redom sa signalom *x* i spoljnim portovima *clk* i *q*.

```
1  ----------------------------------------------------------------
2  LIBRARY IEEE;
3  LIBRARY nasa_biblioteka;
4  USE IEEE.STD_LOGIC_1164.ALL;
5  USE nasa_bibloteka.nas_paket.all;
6  ----------------------------------------------------------------
7  ENTITY pr9 IS
8    PORT(d1, d2, clk: IN BIT;
9         q : OUT BIT);
10 END pr9;
11 ----------------------------------------------------------------
```

```
ARCHITECTURE struct OF pr9 IS
-- komponenta dff -----------
   COMPONENT dff IS
      PORT (d, clk: IN STD_LOGIC;
            q: OUT STD_LOGIC);
   END COMPONENT;
-- komponenta ni_kolo --------
   COMPONENT ni_kolo IS
      PORT (a, b : IN STD_LOGIC;
            c: OUT STD_LOGIC);
   END COMPONENT;
   SIGNAL x : STD_LOGIC;
BEGIN
   K1: ni_kolo PORT MAP(d1, d2, x);
   K2: dff PORT MAP(x, clk, q);
END struct;
-------------------------------------------------------------------
```

Arhitektura koja se sastoji samo iz instanciranih komponenti, kao što je to slučaj u ovom primeru, zapravo predstavlja netlistu, tj. tekstualni opis šematskog ili blok dijagrama. Šta više, postoje softveri za projektovanje koji su u stanju da automatski konvertuju grafičke šematske dijagrame u strukturni VHDL kôd i obrnuto.

Pr. 2-10 D flip-flop i NI kolo – kombinovan opis

Naredba za instanciranje komponenti je takođe konkurentna naredba koja se u istoj arhitekturi može kombinovati s drugim tipovima konkurentnih naredbi i procesima. U VHDL kôdu koji sledi, a koji opisuje mrežu sa Sl. 2-8, iskorišćena je samo jedna komponenta, *dff*, dok je *ni* funkcija realizovana u vidu konkurentne naredbe dodele.

```
-------------------------------------------------------------------
 ARCHITECTURE struct OF pr9 IS
-- komponenta dff -----------
   COMPONENT dff IS
      PORT (d, clk: IN STD_LOGIC;
            q: OUT STD_LOGIC);
   END COMPONENT;
   SIGNAL x : STD_LOGIC;
BEGIN
   x <= d1 NAND d2;
   K2: dff PORT MAP(x, clk, q);
END struct;
```

Iako predstavlja najniži nivo opisa, strukturno modeliranje ima višestruku ulogu u projektovanju digitalnih sistema. Prvo, strukturno modeliranje predstavlja osnovu za hijerarhijsko projektovanje. Po pravilu, složen sistem se deli na nekoliko manjih podsistema od kojih se svaki predstavlja jednom komponentom koja se projektuje nezavisno od drugih komponenti. Ako je to potrebno, podsistem se može dalje podeliti na još manje celine. Drugo, strukturno modeliranje predstavlja način za ponovno korišćenje ranije projektovanih kola. Takva kola, dostupna u obliku VHDL komponenti, jednostavno se instanciraju u novom projektu i tretiraju kao ″crne kutije″. Komponente predviđene za korišćenje u različitim projektima poznate su pod nazivom *IP jezgra* (ili IP korovi od *IP cores*, gde je IP skraćenica od *Intellectual Propriety* – intelektualna svojina). Treće, strukturni VHDL opis se može koristiti za reprezentaciju konačnog rezultata sinteze.

Tipično, ulaz u alati za sintezu je funkcionalni VHDL opis, a izlaz funkcionalno ekvivalentan strukturni VHDL opis u kome se kako komponente koriste elementarna logička kola. Takav strukturni opis se dalje koristi kao ulaz u alate za fizičko projektovanje.

2.3. Projektne jedinice

Projektne jedinice (engl. *design units*) su osnovni gradivni blokovi VHDL opisa. Projektna jedinica je nedeljiva sekcija VHDL kôda koja u potpunosti mora biti sadržana u jednoj projektnoj datoteci. S druge strane, projektna datoteka može sadržati proizvoljan broj projektnih jedinica. Kad se projektna datoteka analizira od strane VHDL simulatora ili softvera za sintezu, ona se zapravo razlaže na projektne jedinice. U VHDL-u postoji pet tipova projektnih jedinica:

- Entitet
- Arhitektura
- Deklaracija paketa
- Telo paketa
- Konfiguracija

Projektne jedinca se dalje dele na primarne i sekundarne. Primarna projektna jedinica može samostalno da egzistira, dok je sekundarna uvek pridružena jednoj primarnoj jedinici. Tako je entitet primarna, a arhitektura sekundarna jedinica. U VHDL projektu, za isti entitet može postojati više različitih arhitektura. S druge strane, svaka arhitektura je pridružena samo jednom entitetu. Sličan odnos takođe važi između deklaracije paketa i tela paketa. Deklaracija paketa je primarna jedinica koja sadrži definicije tipova podataka, potprograma, operacija, komponenata itd. koje se mogu koristiti u različitim delovima projekta. Telo paketa je odgovarajuća sekundarna jedinica koja sadrži implementaciju potprograma i operacija deklarisanih u paketu. Konačno, konfiguracija je projektna jedinica koja definiše spoj primarne i jedne od raspoloživih sekundarnih jedinica, tj. spoj entiteta i arhitekture, odnosno spoj deklaracije paketa i tela paketa.

2.4. Procesiranje VHDL kôda

VHDL projekat se procesira u tri koraka ili faze:

- Analiza
- Elaboracija
- Izvršenje

Tokom faze *analize*, softver najpre provera sintaksu VHDL kôda. Ukoliko ne postoje sintaksne greške, softver razlaže kôd na projektne jedinice koje potom nezavisno prevodi (tj. kompajlira) u tzv. *međukôd*, tj. u oblik koji više nije čitljiv (tekstualan), ali je zato lako ″razumljiv″ simulatorima i alatima za sintezu. Međukôdovi projektnih jedinica se smeštaju u podrazumevanu projektnu biblioteku (*work*) ili u biblioteku koju je projektant naznačio kao odredišnu biblioteku projekta.

Složeni projekti su obično hijerarhijski organizovani. Vršni nivo hijerarhije može sadržati, u vidu instanciranih komponenti, više strukturno povezanih podsistema (kao u Pr. 2-9). Tokom faze *elaboracije*, softver polazi od deklaracije entiteta vršnog nivoa i spaja ga sa odgovarajućom arhitekturom (shodno specifikaciji konfiguracije). Ako u arhitekturi postoje

instancirane komponente, softver zamenjuje svaku instancu komponentu odgovarajućom arhitekturom. Ovaj proces se rekurzivno ponavlja sve dok se sve instancirane komponente ne zamene odgovarajućim arhitekturama. Tako, elaboracija, kroz proces izbora i kombinovanja arhitektura, dovodi do kreiranja jednog ″ravnog″ (ne više hijerarhijskog, već razvijenog) opisa celokupnog sistema.

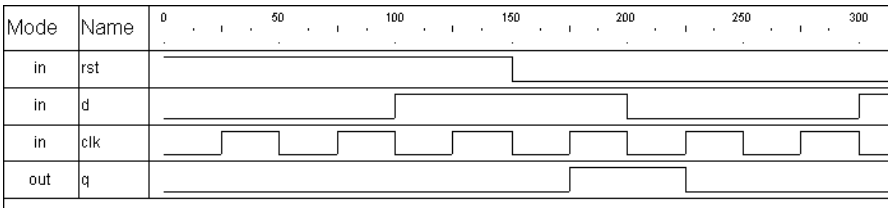
Konačno, u fazi izvršenja, analiziran i elaboriran opis se prosleđuje softveru za simulaciju ili softveru za sintezu. Prvi simulira i ″izvršava″ VHDL opis na računaru, dok drugi sintetiše (realizuje) fizički opis kola.

2.4.1. Simulacija VHDL kôda

Značaj VHDL modeliranja leži u činjenici da se jedan takav opis digitalnog kola može simulirati kako bi se na taj način verifikovala njegova funkcija. Za simulaciju se koriste namenski softveri, tzv. VHDL simulatori. VHDL kôd se najpre procesira, na način opisan u prethodnom odeljku, a zatim učitava u simulator koji ima mogućnost da ″izvršava″ VHDL. Kao rezultat izvršenja, simulator generiše vremenske dijagrame koji pokazuju kako se signali iz VHDL kôda menjaju u vremenu.

Pr. 2-11 Simulacija D flip-flopa

Na Sl. 2-9 prikazan je rezultat simulacije VHDL opisa D flip-flopa iz Pr. 2-7. Grafikon se može lako interpretirati. U prvoj koloni navedeni su smerovi signala, a u drugoj njihova imena. Preostali deo dijagrama sadrži talasne oblike posmatranih signala. Duž dela za talasne oblike naznačeni su vremenski trenuci (izraženi u nanosekundama, ns). U ovom primeru, svi signali su binarni i stoga su njihove vrednosti ′0′ ili ′1′ (odnosno nizak ili visok logički nivo). Kad se koriste vektori, tj. višebitni signali, njihove trenutne vrednosti biće prikazane u obliku binarnog, decimalnog ili heksadecimalnog broja. Oblik ulaznih signala (*rst*, *d* i *clk*) može se proizvoljno zadati u simulatoru. Na osnovu oblika ulaznih signala i VHDL kôda, simulator određuje oblik izlaznih signala (u ovom slučaju postoji samo jedan izlazni signal, *q*). Poređenjem rezultata sa Sl. 2-9 sa očekivanim ponašanjem D flip-flopa sa sinhronim resetom, možemo zaključiti da je simulirani VHDL opis korektan.



Sl. 2-9 Rezultati simulacije VHDL kôda iz Pr. 2-7

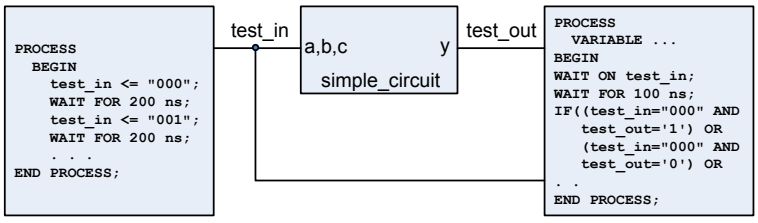
Testbenč. Simulacija VHDL modela je slična sprovođenju eksperimenta nad realnim, fizičkim kolom. U jednom realnom eksperimentu, na ulaze fizičkog kola se deluje stimulansima (pobudinim signalima koje generiše uređaj poput generatora signala) i pri tom se nadgledaju izlazi kola (npr. uređajem kao što je osciloskop ili logički analizator). Simulacija VHDL kôda je neka vrsta virtuelnog eksperimenta u kome je fizičko kolo zamenjeno odgovarajućim VHDL opisom. Šta više, u mogućnosti smo da razvijemo specijalizovane VHDL rutine koje će imitirati rad generatora signala, s jedne i prikupljati i analizirati odzive VHDL modela koji se simulira, s druge strane. Ovakav jedan koncept se naziva *testbenč*.

Pr. 2-12 Testbenč

U ovom primeru je predstavljen testbenč za VHDL kôd iz Pr. 2-3 koji opisuje kombinacionu mrežu sa Sl. 2-4 (*nand_mreza*). Na Sl. 2-10 je prikazana grafička interpretacije testbenča. Testbenč sadrži: generator stimulansa, kolo koje se testira i verifikator koji proverava ispravnost odziva kola. Testbenč je kompletan VHDL modul sa entitetom i arhitekturom. Međutim, pošto testbenč ne poseduje spoljne ulaze i izlaze, njegov entitet je ″prazan″, tj. bez portova (linije 2 i 3). Arhitektura testbenča sadrži tri konkurentne naredbe: jednu naredbu za instanciranje komponente i dva procesa. Naredba za instanciranje komponente (linije 14 i 15) instancira jedan primerak kola koje se testira i posredstvom internih signala, *test_in* i *test_out*, povezuje ga sa generatorom stimulansa i verfikatorom. Prvi od dva procesa (linije 17-35) igra ulogu generatora stimulansa. Ovaj proces generiše sve moguće test-vektore, tj. binarne kombinacije koje se dovode na ulaz kola koje se testira, počev od ″000″ pa sve redom do ″111″. Svaki test-vektor traje 200 ns. Drugi proces (linije 37-59) igra ulogu verifikatora. On čeka da se pojavi novi test-vektor (linija 40), zatim čeka još 100 ns (linija 41), a onda proverava da je vrednost koja je u tom momentu prisutna na izlazu kola koje se testira identična očekivanoj, poznatoj vrednosti. U slučaju odstupanja, verifikator prijavljuje grešku (linije 56-58). (VHDL kôd iz ovog primera je dat radi ilustracije koncepta testbenča i za sada ne treba brinuti oko sintaksnih detalja.)

```
1  ----------------------------------------------------------------
2  ENTITY nand_mreza_testbench IS
3  END nand_mreza_testbench;
4  ----------------------------------------------------------------
5  ARCHITECTURE tb_arch OF nand_mreza_testbench IS
6    COMPONENT nand_mreza
7         PORT(a, b, c: IN BIT;
8              y : OUT BIT);
9    END COMPONENT;
10   SIGNAL test_in : BIT_VECTOR(2 DOWNTO 0);
11   SIGNAL test_out : BIT;
12 BEGIN
13 -- Instanciranje kola koje se testira ----------------------------
14   uut: nand_mreza
15   PORT MAP(a=>test_in(2),b=>test_in(1),c=>test_in(0),y=>test_out);
16 -- Generator stimulansa (test vektora) ---------------------------
17   PROCESS
18   BEGIN
19     test_in <= "000";
20     WAIT FOR 200 ns;
21     test_in <= "001";
22     WAIT FOR 200 ns;
23     test_in <= "010";
24     WAIT FOR 200 ns;
25     test_in <= "011";
26     WAIT FOR 200 ns;
27     test_in <= "100";
28     WAIT FOR 200 ns;
29     test_in <= "101";
30     WAIT FOR 200 ns;
31     test_in <= "110";
32     WAIT FOR 200 ns;
```

```
33     test_in <= "111";
34     WAIT FOR 200 ns;
35  END PROCESS;
36 -- Verifikator ----------------------------------------------------
37  PROCESS
38    VARIABLE error_status : BOOLEAN;
39  BEGIN
40    WAIT ON test_in;
41    WAIT FOR 100 ns;
42    IF((test_in="000" AND test_out='1') OR
43       (test_in="001" AND test_out='0') OR
44       (test_in="010" AND test_out='1') OR
45       (test_in="011" AND test_out='0') OR
46       (test_in="100" AND test_out='1') OR
47       (test_in="101" AND test_out='0') OR
48       (test_in="110" AND test_out='1') OR
49       (test_in="111" AND test_out='1'))
50    THEN
51      error_status := FALSE;
52    ELSE
53      error_status := TRUE;
54    END IF;
55 -- prijavljivanje greske ---------------------------------------
56    ASSERT NOT error_status
57      REPORT "greska!"
58      SEVERITY note;
59  END PROCESS;
60 END tb_arch;
61 ---------------------------------------------------------------------
```



Sl. 2-10 Konceptualni prikaz testbenča.

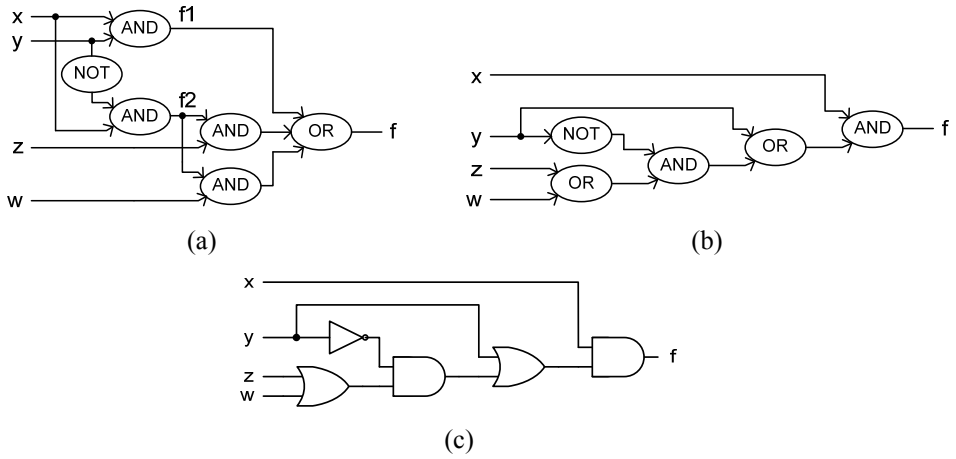
2.4.2. Sinteza VHDL kôda

VHDL je osmišljen i projektovan prevashodno kao jezik za simulaciju. Potreba za sintezom se javila naknadno. Softver za sintezu radi tako što pronalazi i preslikava jezičke konstrukcije iz VHDL kôda na hardverske elemente identičnog ponašanja (funkcije). Ovo preslikavanje nije uvek jednostavan zadatak. Za pojedine konstrukcije iz VHDL jezika postoje direktni hardverski ekvivalenti, pred neke druge se postavljaju posebni zahtevi i ograničenja kako bi mogle da se sintetišu, dok za mnoge jezičke konstrukcije hardverska interpretacija jednostavno ne postoji. Iz tog razloga, VHDL koji se koristi za pisanje kôda za sintezu je tek podskup ″punog″ VHDL-a. Poznavanje ovih ograničenja je od velike važnosti, jer ona diktiraju način na koji treba pisati kôd da bi on mogao da se sintetiše. Preporuka je da VHDL opise treba u startu pisati za sintezu, jer kôd koji je uspešno prošao funkcionalnu simulaciju, ne znači da se može i sintetizovati.

Radi ilustracije toka hardverske sinteze, razmotrićemo sledeći segment VHDL kôda:

```
. . .
f1 <= x AND y;
f2 <= x AND NOT y;
f <= f1 OR (f2 AND z) OR (f2 AND w);
. . .
```

Sinteza VHDL kôda se obavlja u nekoliko uzastopnih koraka. U prvom koraku, softver za sintezu transformiše polazni VHDL opis u funkcionalnu mrežu, koja za razmatrani VHDL kôd izgleda kao na Sl. 2-11(a). Kao što vidimo, funkcionalna mreža sadrži logičke operatore koji su povezani na način kao u naredbama VHDL kôda. U drugom koraku, funkcionalna mreža se pojednostavljuje primenom različitih metoda za automatsku optimizaciju logičkih funkcija (Sl. 2-11(b)). Konačno, u trećem koraku, operatori iz optimizovane funkcionalne mreže se preslikavaju na hardverske elemente (tj., u ovom jednostavnom primeru, na logička kola).

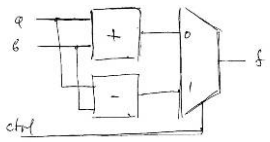


Sl. 2-11 Sinteza VHDL kôda: (a) inicijalna funkcionalna mreža; (b) funkcionalna mreža nakon optimizacije. (c) konačni rezultat sinteze u obliku logičke mreže.

VHDL opis može biti značajno složeniji od onog sa Sl. 2-11 i može da sadrži ne samo logičke već i aritmetičke operacije, pa i složenije programske konstrukcije (npr. *if*, *select*, itd.). Međutim, da bi sinteza bila moguća, kôd mora biti tako napisan da se može transformisati u funkcionalnu mrežu. Otuda je dobra praksa da pisanju VHDL kôda za sintezu prethodi crtanje skice konceptualnog dijagrama koji će poslužiti kao model za pisanje kôda (slično kao što dijagram toka služi kao model za pisanje programa). Ovaj koncept je ilustrovan na Sl. 2-12.

Na Sl. 2-12(a) vidimo crtež konceptualnog dijagrama digitalnog kola koje u zavisnosti od vrednosti ulaza *s* na izlazu *f* daje *a*+*b* ili *a*-*b*. Na Sl. 2-12(b) je dat segment VHDL kôda koji je napisan po ugledu na dijagram sa Sl. 2-12(a). Cilj crtanja konceptualnog dijagrama nije precizno definisanje strukture sintetizovanog hardvera, već kreiranje vizuelne, funkcionalne predstave kola. U prvom koraku sinteze, VHDL kôd sa Sl. 2-12(b) biće interno transformisan u funkcionalnu mrežu koja će po obliku biti identična ili barem slična skici konceptualnog dijagrama (Sl. 2-12(c)), ali će potom uslediti niz optimizacija tokom kojih funkcionalna mreža može biti transformisana u drugačiji oblik, koji će biti funkcionalno identičan polaznom, ali jednostavniji, kao npr. mreža sa Sl. 2-12(d).

Projektant:

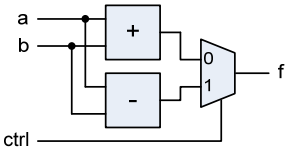


(a)

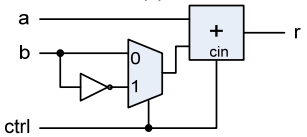
```
. . .
f <= a + b WHEN ctrl='0' ELSE
     a - b;
. . .
```

(b)

Softver za sintezu:



(c)

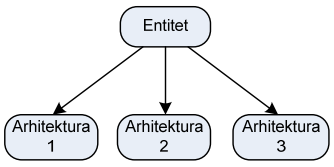


(d)

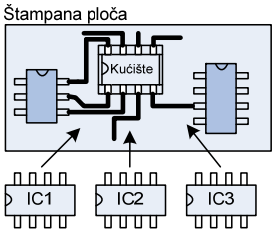
Sl. 2-12 VHDL sinteza. *Aktivnosti projektanta*: (a) crtanje skice konceptualnog dijagrama; (b) pisanje VHDL kôd. *Aktivnosti softvera za sintezu*: (c) transformacija VHDL kôda u funkcionalnu mrežu; (d) optimizacija funkcionalne mreže.

2.5. CONFIGURATION

Entitet i arhitektura su ključni koncepti VHDL-a. Razdvajanje specifikacije interfejsa (entitet) od specifikacije funkcije/strukture kola (arhitektura), omogućava projektantu da zadrži isti entitet, a zameni arhitekturu novom, eventualno detaljnijom ili unapređenom verzijom. Kao što je već rečeno, istom entitetu se može pridružiti više različitih arhitektura (Sl. 2-13(a)). Može se uspostaviti analogija sa integrisanim kolom (IC) i kućištem za IC. Entitet se može zamisliti kao kućište na štampanoj ploči predviđeno za IC neke konkretne funkcije. Ovo kućište je prazno, ali sa definisanim rasporedom pinova. Arhitekture koje su pisane za taj entitet mogu se zamisliti kao različita IC sa istim rasporedom pinova kao kućište (Sl. 2-13(b)). Iako, funkcionalno i spolja gledano identična, ova IC se mogu razlikovati po načinu kako je njihova funkcija interno realizovana ili u performansama ili po nekim drugim npr. električnim karakteristikama. Shodno konkretnim zahtevima, u mogućnosti smo da odaberemo jedno od raspoloživih IC-ova i ugradimo ga u kućište, a da pri tom ne moramo bilo šta da menjamo na štampanoj ploči. VHDL poseduje mehanizam, podržan naredbom *configuration*, koji na sličan način omogućava povezivanje entiteta i arhitekture.



(a)



(b)

Sl. 2-13 Konfiguracija: (a) entitet i arhitekture; (b) kućište za IC i IC-ovi.

Pr. 2-13 Konfiguracija

U prethodnim primerima predstavljene su dve arhitekture za kombinacionu mrežu *nand_mreza* sa Sl. 2-4. Prva, nazvana *dataflow*, koristi logičke operacije (Pr. 2-3), a druga,

nazvana *funct*, opisuje funkciju mreže na nešto višem nivou apstrakcije, korišćenjem naredbe *select* (Pr. 2-6). U Pr. 2-12 je dat testbenč za ovo kolo. Međutim, ako pažljivo pogledamo VHDL kôd ovog testbenča, uočićemo da ni u deklaraciji komponente *nand_mreza* (linije 6-9 kôda iz Pr. 2-12) ni u naredbi za instanciranje ove komponente (linije 14-15) nije navedeno koju od dve arhitekture treba koristiti, *dataflow* ili *funct*. Testbenč je poput štampane ploče sa praznim kućištem za komponentu koja se testira, dok su alternativne arhitekture ove komponente poput različitih IC-ova koja se mogu umetnuti u kućište. Ispod je dat kôd konfiguracije u kome je za korišćenje u testbenču komponenta *nand_mreza* eksplicitno povezana sa arhitekturom *dataflow*. Tumačenje kôda je sledeće: Konfiguracija, nazvana *demo_config*, odnosi se na entitet *nand_mreza_testbanch*, tj. na njegovu arhitekturu *tb_arch* u kojoj se za instancu *uut* entiteta *nand_mreza* koristi arhitektura *dataflow* koja se može naći u podrazumevanoj radnoj biblioteci (*work*). Treba napomenuti da deklaracija konfiguracije nije neophodna ako u projektu postoji samo jedna arhitektura za dati entitet. U tom slučaju, povezivanje entiteta i arhitekture se obavlja po automatizmu.

```
1 ----------------------------------------------------------------
2 CONFIGURATION demo_config OF nand_mreza_testbench IS
3  FOR tb_arch
4    FOR uut : nand_mreza
5      USE ENTITY work. nand_mreza(dataflow);
6    END FOR;
7  END FOR;
8 END demo_config;
```

2.6. Uloga VHDL-a u procesu projektovanja

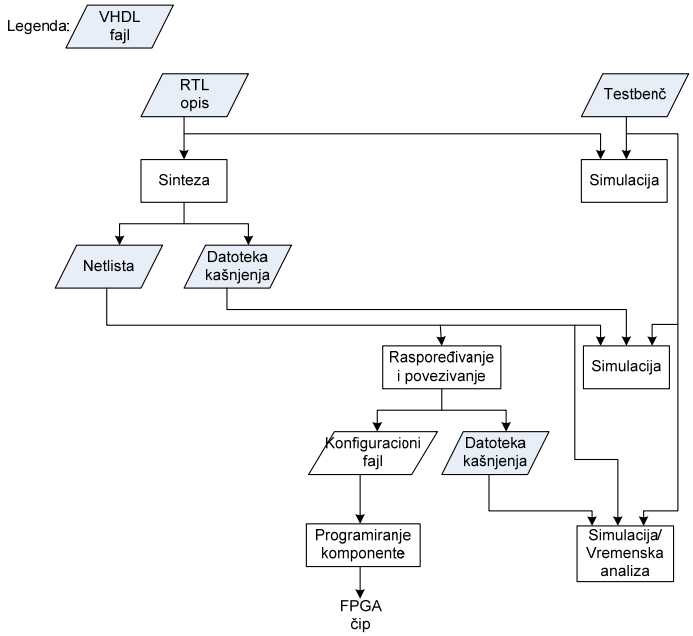
Primeri iz prethodnih poglavlja ukazali su nam na osnovne jezičke konstrukcije i mogućnosti VHDL-a. Izbor ovih konstrukcija nije slučajan, već su one pažljivo odabrane kako bi se istakla glavna svrha VHDL-a – podrška projektovanju digitalnih sistema. VHDL pruža unificirano okruženje za celokupan proces projektovanja digitalnih sistema. Ne samo da sadrži konstrukcije za modeliranje sistema na različitim nivoima apstrakcije, od apstraktno-funkcionalnog do strukturnog, već takođe predstavlja i okvir za simulaciju i verifikaciju. Radi sticanja potpunijeg uvida u ulogu koju VHDL igra u savremenom projektovanju, na Sl. 2-14 je prikazan dijagram toka projektovanja digitalnih sistema za implementaciju u FPGA tehnologiji.

Projektovanje složenijeg digitalnog sistema tipično počinje kreiranjem njegovog funkcionalnog modela višeg nivoa apstrakcije i odgovarajućeg testbenča koji sadrži test vektore za testiranje opisane funkcionalnosti. Ovaj opis i testbenč služe projektantu da detaljno analizira model, otkrije njegove eventualne nedostatke, razjasni i finalizira polaznu specifikaciju i konačno, precizno definiše ulazno-izlazno ponašanje budućeg digitalnog sistema. Arhitektura *dataflow* (iz Pr. 2-3) i odgovarajući testbenč (iz Pr. 2-12) nalikuju ovakvoj vrsti kôda. Apstraktni opisi složenih sistema obično nisu pogodni za sintezu, bilo zato što bi se njegovom sintezom kreirao nepotrebno složen hardver, bilo zato što se uopšte i ne mogu sintetizovati.

Nakon što je u potpunosti razjasnio sve aspekte funkcionisanja sistema i kreirao testbenč, projektant kreira VHDL kôd za sintezu. Po pravilu, ovaj kôd je opis RTL nova i zapravo daje ″skicu″ hardverske organizacije sistema, na osnovu koje softverski alat može da izvrši efikasnu sintezu hardvera. Arhitektura *dataflow* za kolo *nand_mreza* nalikuje ovakvom

kôdu. Međutim, kôd predviđen za sintezu mora biti verifikovan pre nego što se sintetiše u hardver. Za verifikaciju se može koristiti isti testbenč koji je razvijen tokom polaznog, funkcionalnog modeliranja sistema – dovoljno je, uz pomoć VHDL konfiguracije, postojeću ″apstraktnu″ arhitekturu zameniti novom, detaljnijom. Nova, RTL arhitektura mora da dâ identičan odziv kao apstraktna.

Nakon uspešne funkcionalne verifikacije, RTL opis je spreman za sintezu. Rezultat sinteze je netlista u obliku strukturnog VHDL opisa (poput VHDL kôda iz Pr. 2-9). Ovaj opis je obično nesrazmerno obimniji od polaznog RTL opisa, jer je sada sistem razgrađen do nivoa elementarnih logičkih kola i flip-flopova. Zbog obilja detalja koje sadrži, netlista nije pogodna za interpretaciji i tumačenju od strane čoveka, ali se zato lako može, kao nova konfiguraciona jedinica, umetnuti u postojeći testbenč. Simulacijom testbenča može se verifikovati korektnost sinteze i preliminarno analizirati tajming sistema. Nakon uspešne verifikacije, netlista je spremna za dalje procesiranje, koje se obavlja uz pomoć softverskih alata za raspoređivanje i rutiranje. Ovi alati generišu lejaut, koji više nije u obliku VHDL-a. S jedne strane, lejaut se prevodi u konfiguracionu datoteku, koja se koristi za programiranje FPGA komponente. S druge strane, iz generisanog lejauta se ekstrahuje informacija o tajmingu (propagaciona kašnjenja kroz komponente) koja se u obliku datoteke kašnjenja može pridodati postojećem strukturnom opisu (netlisti). Ovaj novi opis se ponovo uključuje u testbenč i simulira radi završne verifikacije tajminga.



Sl. 2-14 Uloga VHDL-a u procesu projektovanja.

3. OSNOVNE JEZIČKE KONSTRUKCIJE

U ovoj glavi biće dat pregled osnovnih jezičkih konstrukcija VHDL-a, što uključuje leksičke elemente, objekte, tipove podatka, operatore i atribute. VHDL je strogo-tipiziran jezik, što znači da nameće stroga pravila u načinu korišćenja tipova podataka i operatora. Posebna pažnja biće posvećena tipovima podataka i pratećim operacijama koje je dozvoljeno koristiti u kôdu namenjenom za sintezu.

3.1. Leksički elementi

Leksički elementi su osnovne sintaksne jedinice VHDL-a. To su: komentari, identifikatori, rezervisane reči, brojevi, karakteri i stringovi.

Komentari. Komentar počinje sa dve crtice ″--″ posle kojih sledi tekst komentara. Komentari se koriste radi dokumentacije i nemaju uticaja na procesiranje kôd. (Prilikom kompajliranja, celokupan tekst počev od ″--″ pa do kraja tekuće linije se ignoriše.)

Identifikatori. Identifikatori su imena objekata u VHDL-u (signala, promenljivih, entiteta, arhitektura itd.). Pravila za pisanje identifikatora su:

- Identifikator može sadržati samo slova, decimalne cifre i crtu za podvlačenje.
- Prvi karakter mora biti slovo.
- Poslednji karakter ne može biti crta za podvlačenje.
- Dve uzastopne crte za podvlačenje nisu dozvoljene.

Na primer, sledeći identifikatori su ispravni:

```
A11, sledece_stanje, NextState, addr_en
```

Sledeći identifikatori krše neko od navedenih pravila usled čega izazivaju sintaksnu grešku prilikom analize programa:

```
x#8, _x3, 5linija, a10_, ab__cd
```

VHDL ne pravi razliku između malih i velikih slova, a to važi kako za identifikatore, tako i za službene reči. Tako *a* i *A*, *ni_kolo* i *NI_Kolo*, kao i *Port*, *port* i *PORT* znače isto.

Rezervisane reči. U jeziku VHDL, neke reči imaju posebno značenje i ne mogu se koristiti kao identifikatori. Ove reči su poznate kao *službene* ili *rezervisane* reči. Sledi njihov spisak:

```
abs access after alias all and architecture array
assert attribute begin block body buffer bus case
component configuration constant disconnect downto
else elsif end entity exit file for function generate
generic guarded if impure in inertial inout is label
library linkage literal loop map mod nand new next
nor not null of on open or others out package port
postponed procedure process pure range record
register reject rem report return rol ror select
severity shared signal sla sll sra srl subtype then
to transport type unaffected units until use variable
wait when while with xnor xor
```

Brojevi, karakteri i stringovi. U VHDL-u, brojevi mogu biti celi, kao npr. 0, 123456 i 98E7 (= $98 \cdot 10^7$) ili realni, kao npr. 0.0, 1.2345 ili 6.82E6 (= $6.82 \cdot 10^6$). Osim kao decimalni, brojevi se mogu predstaviti i u drugim brojnim osnovama. Na primer, 45 se može napisati kao 2#101101# u osnovi 2, odnosno 16#2D# u osnovi 16 (vrednost na početku zapisa ukazuje na brojnu osnovu, dok se vrednost broja zapisuje između znakova #). Radi poboljšanja čitljivosti, dozvoljeno je korišćenje crte za podvlačenje. Na primer, 12_3456 je isti što i 123456, a 2#0011_1010_1101# isto što i 2#001110101101#.

U VHDL-u, karakteri (znakovi) se pišu pod jednostrukim navodnicima, npr. 'A', 'a' ili '7'. Uočimo da 1 nije isto što i '1' (1 je broj, a '1' karakter).

String je niz karaktera pod dvostrukim navodnicima, kao npr. ″Alo″ i ″1000111″. Ponovo, uočimo da 2#10110010# i ″10110010″ nisu isto, budući da je 2#10110010# broj, a ″10110010″ string. Za razliku od brojeva, unutar stringova nije dozvoljeno koristiti crtu za podvlačenje. Na primer, ″10100010″ i ″1011_0010″ su dva različita stringa.

3.2. Objekti

U VHDL-u, pod objektom se smatra imenovana stavka koja sadrži (čuva) vrednost podatka određenog tipa. Za reprezentaciju podatka u VHDL-u se koriste tri tipa objekata: signali, varijable i konstante.

***Signali*.** Signali su najčešće korišćeni objekti u VHDL-u. Signali se koriste za povezivanje komponenti i razmenu podataka između entiteta. Mogu se zamisliti, zavisno od konteksta, bilo kao električne veze u fizičkom kolu, bilo kao ″veze sa memorijom″, tj. registri ili leč kola. Portovi entiteta su takođe signali.

Signal se deklariše u deklarativnoj sekciji arhitekture shodno sledećoj sintaksi:

```
SIGNAL ime_signala,..., ime_signala : TIP_PODATAKA;
```

Na primer, u sledećoj liniji deklarisana su tri signala tipa *std_logic*:

```
SIGNAL x, y, z : STD_LOGIC;
```

Prilikom deklaracije, signalu se može dodeliti početna, tj. inicijalna vrednost:

```
SIGNAL x, y, z : STD_LOGIC := ´0´;
```

Međutim, inicijalne vrednosti signala podržane su samo u simulaciji VHDL kôda, dok se u kôdu za sintezu ova mogućnost ne koristi.

Za dodelu vrednosti signalu koristi se operator ″<=″:

```
ime_signala <= izraz;
```

Treba naglasiti da je dodela vrednosti signalu uvek *odložena* u odnosu na trenutak izračunavanja izraza. Takođe, izraz s desne strane znaka ″<=″, u svom najopštijem obliku, ne definiše samo pojedinačnu vrednost, već kompletan talasni oblik signala, tj. niz vrednosti koje će se u odgovarajućim vremenskim trenucima dodeljivati signalu. Detaljnije razmatranje osobina signala, naredbe dodele i drugih jezičkih konstrukcija koje se tiču signala sledi u glavi 4.

Varijable. Varijable odgovaraju promenljivama iz programskih jezika. Koriste se za čuvanje lokalnih podataka u sekvencijalnim sekcijama VHDL kôda (procesima i potprogramima). Ne postoji neka precizna fizička interpretacija varijabli, a svoju osnovnu primenu nalaze u apstraktnom modeliranju. Varijable se deklarišu u deklarativnoj sekciji procesa (ili potprograma) na sledeći način:

```
VARIABLE ime_varijable,..., ime_varijable : TIP_PODATAKA;
```

Prilikom deklaracije, varijabli se može dodeliti inicijalna vrednost. Međutim, kao i za signale, ova mogućnost je podržana samo u simulaciji.

Za dodelu vrednosti varijabli koristi se operator ″:=″:

```
ime_varijable := izraz;
```

Za razliku od signala, ažuriranje vrednosti varijable je trenutno, odnosno u momentu izvršenja naredbe dodele. Korišćenje varijabli u VHDL-u biće detaljnije razmatrano u glavi 5, koja je posvećena sekvencijalnom kôdu.

Konstante. Konstanta je objekat koji sadrži nepromenljivu vrednost. Konstanta se može deklarisati u entitetu, arhitekturi ili paketu, na sledeći način:

```
CONSTANT ime_konstante : TIP_PODATAKA := izraz;
```

Član *izraz* s kraja deklaracije definiše vrednost konstante. Na primer, u sledećim linijama kôda deklarisane su dve celobrojne (*integer*) konstante, M i N:

```
CONSTANT M : INTEGER := 32;
CONSTANT N : INTEGER := M/4;
```

Pr. 3-1 Upotreba konstanti

```
ARCHITECTURE arch OF parity IS
  SIGNAL x : STD_LOGIC;
BEGIN
  . . .
  p := ´0´;
  FOR i IN 7 DOWNTO 0 LOOP
    P := p AND a(i);
  END LOOP;
  . . .
```

```
ARCHITECTURE arch OF parity IS
  SIGNAL x : STD_LOGIC;
  CONSTANT N : INTEGER := 7;
BEGIN
  . . .
  p := ´0´;
  FOR i IN N DOWNTO 0 LOOP
    P := p AND a(i);
  END LOOP;
  . . .
```

Iznad su data dva funkcionalno identična VHDL opisa. Razlika između ova dva opisa je

samo u tome što je u drugom opisu konkretna vrednost za početni indeks petlje, 7, zamenjena konstantom *N*. S ovom jednostavnom modifikacijom, opis postaje razumljiviji, jer nam brzo otkriva mesto u kôdu gde je definisan jedan bitan parametar sistema. Takođe, eventualna naknadna promena vrednosti ovog parametra zahtevala bi samo modifikaciju vrednosti konstante, ali ne i bilo kakvu intervenciju u kôdu tela arhitekture.

3.3. Tipovi podataka

Osnovna osobina svakog objekta u VHDL-u jeste pripadnost jednom konkretnom tipu podataka. Tip podataka definiše:

- skup vrednosti koje mogu biti dodeljene objektu, i
- skup operacija koje se mogu primenjivati nad objektima datog tipa.

VHDL je *strogo-tipiziran* jezik, što znači da su operacije koje se javljaju u naredbama legalne samo ako su tipovi operanada usklađeni. Drugim rečima, nije dozvoljeno u istom izrazu koristiti operande različitog tipa. Primer nelegalne operacije je sabiranje realnog i celog broja ili dodela binarne vrednosti celobrojnoj promenljivoj. Ukoliko postoji potreba za ovakvom operacijom, onda se najpre moraju uskladiti tipovi operanada, primenom odgovarajuće funkcije za konverziju tipa.

VHDL je bogat tipovima podataka. Osim tipova koji su ugrađeni u sam jezik (tzv. *predefinisani* tipovi podataka) i tipova koji su raspoloživi kroz standardne biblioteke i pakete, VHDL omogućava projektantu da po potrebi uvodi i nove, tzv. *korisničke* tipove podataka. Pojedini tipovi podataka se mogu koristiti samo u simulaciji, dok se drugi mogu koristiti i u sintezi. Dobro poznavanje tipova podataka, načina njihovog korišćenja i ograničenja, od velike je važnosti za dobro razumevanje VHDL-a i pisanje efikasnog i lako razumljivog VHDL kôda. U ovom poglavlju biće reči o svim važnim tipovima podataka u VHDL-u, s posebnim naglaskom na one koji se mogu koristiti u sintezi.

3.3.1. Predefinisani tipovi podataka

Sledi kratak pregled nekoliko često korišćenih predefinisanih tipova podataka.

BIT i BIT_VECTOR: definiše dvonivovsku logiku. Dozvoljene vrednosti tipa *bit* su ´0´ i ´1´. Tip *bit_vector* je vektorska varijanta tipa *bit*, tj. definiše niz (vektor, ili 1D polje) bitova.

```
SIGNAL x : BIT;
-- deklariše x kao jednobitni signal tipa bit

SIGNAL y : BIT_VECTOR(3 DOWNTO 0);
-- y je 4-bitni vektor. (3 DOWNTO 0) definiše opseg indeksa bitova u
-- vektoru i njihov poredak (u ovom slučaju opadajući)

SIGNAL w : BIT_VECTOR(0 TO 7);
-- w je 8-bitni vektor. (0 TO 7) definiše opseg indeksa bitova u
-- vektoru i njihov poredak (u ovom slučaju rastući)

SIGNAL z : BIT := ´1´;
-- z je jednobitni signal inicijalne vrednost ´1´.
```

Prvobitna namena tipa *bit* bila je modeliranje logičkih vrednosti 0 i 1 iz Bulove algebre i digitalne logike. Međutim, u realnim digitalnim sistemima, osim binarnih ´0´ i ´1´ signali

mogu imati i druge (nebinarne) vrednosti, kao što su ″visoka impedansa″ na izlazu trostatičkog bafera, ili ″nedefinisana vrednost″ na kratkom spoju izlaza dvaju ili više digitalnih kola. Iz tog razloga, uveden je tip *std_logic* (i *std_logic_vector*), koji je, budući bogatiji vrednostima, za primene u sintezi zamenio tip *bit*.

***BOOLEAN*:** definiše logičke vrednosti **true** (tačno) i **false** (netačno). Uočimo da tip *boolean* nije isto što i tip *bit*. Razlika je ilustrovana sledećim primerom:

```
IF (a) THEN ...            -- a je tipa boolean
IF (a=´1´) THEN ...        -- a je tipa bit
```

INTEGER: 32-bitni celi (*integer*) brojevi. Opseg dozvoljenih vrednosti je od $-(2^{31}-1)$ do $2^{31}-1$ tj. od -2,147,483,647 do +2,147,483,647. U upotrebi su i dva izvedena tipa podatka (tj. podtipa): *natural* i *positive*. Podtip *natural* obuhvata nenegativne cele brojeve (uključuje 0), a *positive* pozitivne cele brojeve (bez 0).

3.3.2. Operatori

U strogo-tipiziranom jeziku kakav je VHDL, operatori su pridruženi definiciji tipa podataka i mogu se primenjivati samo na objekte tog tipa. Zato je važno znati na koje tipove podataka se koji operator može primeniti. U VHDL-u postoji oko 30 operatora. U tabeli T. 3-1 navedeni su najvažniji operatori koji su bitni za sintezu. Većina operatora su sami po sebi jasni. Neke napomene u vezi operatora poređenja i konkatenacije mogu se naći su odeljku 3.3.4.

Pr. 3-2 Operatori pomeranja

Ako je *x* signal tipa *bit_vector* i njegova vrednost je *x*=″01001″, tada važi:

```
Y <= x sll 2; -- y <= ″00100″, logičko pomeranje ulevo za 2 mesta
y <= x sla 2; -- y <= ″00111″ aritmetičko pomeranje ulevo za 2 mesta
y <= x srl 3; -- y <= ″00001″, logičko pomeranje udesno za 3 mesta
y <= x sra 3; -- y <= ″00001″aritmetičko pomeranje udesno za 3 mesta
y <= x rol 2; -- y <= ″00101″, rotacija ulevo za 2 mesta
y <= x ror 2; -- y <= ″01010″, rotacija udesno za 2 mesta
y <= x rol -2; -- isto što i ror 2
```

Prva četiri operatora obavljaju pomeranje, a poslednja dva rotiranje bit-vektora. Pri pomeranju gubi se onoliko bita za koliko pozicija se vektor pomera. Pri rotiranju bitovi se ne gube, jer se oni koji bi inače ″ispali″ iz vektora vraćaju na suprotan kraj vektora. Postoje dve varijante pomeranja: logičko i aritmetičko. Razlika među njima je u tome kako se tretiraju pozicije koje bi nakon pomeranja ostale prazne. Pri logičkom pomeranju, takve pozicije se popunjavaju prvom vrednošću odgovarajućeg tipa (npr. za tip *bit_vector* to je ´0´). Aritmetičko pomeranje ulevo (*sla*) popunjava ispražnjenje pozicije vrednošću krajnjeg desnog, a aritmetičko pomeranje udesno (*sra*) vrednošću krajnjeg levog bita vektora koji se pomera. Napomenimo da je u VHDL kôdu namenjenom za sintezu operatore pomeranja dozvoljeno koristiti samo nad operandima tipa *bit_vector*, ali ne i nad operandima tipa *std_logic_vector*.

Pr. 3-3 Ograničenja predefinisanih aritmetičkih operatora

U kôdu za sintezu ne postoje striktna ograničenja koja se odnose na korišćenje operacija sabiranja i oduzimanja. U opštem slučaju to važi i za operaciju množenja. Kod deljenja,

dozvoljeni su samo delioci koji imaju vrednost stepena dvojke (jer se u tom slučaju deljenje svodi se na operaciju pomeranja). Stepenovanje je dozvoljeno samo prilikom definisanja vrednosti konstanti (osnova i eksponent mogu biti samo konkretne vrednosti). Alati za sintezu po pravilu ne podržavaju poslednja tri operatora (*mod*, *rem* i *abs*). Rezultat operacije *x rem y* je ostatak celobrojnog deljenja *x*/*y* (= *x* – (*x*/*y*)**y*). Znak rezultata isti je kao znak levog operanda. Npr. 7 *rem* 4 = 3; (-7) *rem* 4 = -3; 7 *rem* (-4) = 3. Rezultat operacije *x mod y* jednak je *x* – *y***N*, gde je *N* najveći ceo broj za koji važi *x* ≤ *y***N*. Znak rezultata isti je kao znak desnog operanda. Npr. 7 *mod* 4 = 3; 7 *mod* (-4) = -1.

T. 3-1 Operatori za predefinisane tipove podataka.

Operacija	**Opis**	**Tip operanda *a***	**Tip operanda *b***	**Tip rezultata**
Logičke operacije				
NOT a	negacija	boolean, bit, bit_vector		boolean, bit, bit_vector
a **AND** b	I	boolean, bit, bit_vector	isti kao *a*	isti kao *a*
a **OR** b	ILI			
a **XOR** b	isključivo ILI			
a **NOR** b	NILI			
a **NAND** b	NI			
a **XNOR** b	isključivo NILI			
Aritmetičke operacije				
a + b	sabiranje	integer	integer	integer
a **-** b	oduzimanje			
a * b	množenje			
a / b	deljenje			
a ** b	stepenovanje			
a **MOD** b	moduo			
a **REM** b	ostatak			
ABS a	apsolutna vrednost	integer		integer
- a	negacija			
Operacije poređenja				
a = b	jednako	bilo koji	isti kao *a*	boolean
a /= b	različito			
a < b	manje	skalar ili 1-D polje	isti kako *a*	boolean
a > b	veće			
a <= b	manje ili jednako			
a >= b	veće ili jednako			
Operacije pomeranja				
a **sll** b	logičko pomeranje ulevo	bit_vector	integer	bit_vector
a **srl** b	logičko pomeranje udesno			
a **sla** b	aritmetičko pomeranje ulevo			
a **sra** b	aritmetičko pomeranje udesno			
a **rol** b	rotacija na levo			
a **ror** b	rotacija na desno			
Konkatenacija				
a **&** b	konkatenacija (nadovezivanje)	1D polje, element	1D polje, element	1D polje

Prioriteti operatora. Ukoliko se koristi više operatora u jednom izrazu, a bez zagrada koje bi odredile redosled izračunavanja, potrebno je voditi računa o prioritetima operatora. U tabeli T. 3-2 data je lista operatora prema prioritetima. Operatori su podeljeni u nekoliko grupa, pri čemu operatori iz iste grupe imaju isti prioritet.

T. 3-2 Prioriteti operatora

Prioritet	Operator
Najviši	** ABS NOT
	* / MOD REM
	- (negacija)
↓	& + -
	sll srl sla sra rol ror
	= /= < <= > >=
Najniži	and or nand nor xor xnor

Izračunavanju operatora nižeg prioriteta prethodi izračunavanje operatora višeg prioriteta. Na primer, razmotrimo sledeći izraz:

```
a + b > c OR a < d
```

Prvo će biti izračunat operator ″+″, zatim operatori ″>″ i ″<″ i konačno, operator *or*. U izrazu koji sadrži više istih operatora izračunavanje se vrši s leva na desno. Tako će u izrazu:

```
a + b + c + d
```

prvo biti izračunato *a* + *b*, zatim će na dobijenu sumu biti dodato *c*, a onda i *d*.

U izrazima se mogu koristiti i zagrade. Zagrade imaju najviši prioritet i koriste se da bi se nametnuo željeni redosled izračunavanja. Na primer, uz pomoć zagrada moguće je promeniti redosled izračunavanje u prethodnom izrazu na s desna na levo:

```
a + (b + (c + d))
```

Za razliku od Bulove algebre, u VHDL-u logički operatori *and* i *or* imaju isti prioritet. Zato su zagrade neophodne da bi se nametnuo željeni redosled računanja, kao u sledećem izrazu:

```
(a AND b) OR (b AND c)
```

Da bi se poboljšala čitljivost i razumljivost kôda, dobra je praksa da se podizrazi nekog složenijeg izraza odvajaju zagradama i onda kada one nisu neophodne. Na primer, izraz:

```
a + b > c OR a < d
```

se može napisati u obliku:

```
((a + b) > c)) OR (a < d)
```

Očito, izraz sa zagradama je jasniji, a šanse za grešku u pisanju ili pogrešno tumačenje izraza su svedene na minimum.

Treba biti obazriv prilikom pisanja logičkih izraza sa više od dva *nand* ili *nor* operatora. Operatori *nand* i *nor* nisu asocijativni, kao što je to poznato iz Bulove algebre, npr:

```
(a NAND b) NAND c ≠ a NAND (b NAND c)
```

Zbog toga je sledeća naredba sintaksno neispravna:

```
y <= a NAND b NAND c;
```

Neophodne su zagrade:

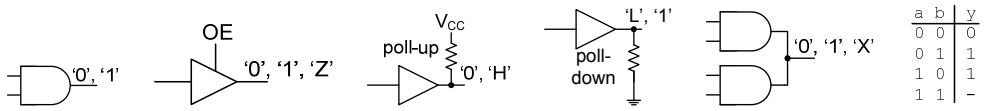
```
y <= (a NAND b) NAND c;
```
(ili, `y <= a NAND (b NAND c);`)

3.3.3. Tipovi podataka iz paketa *IEEE std_logic_1164*

Potreba za vernijim modeliranjem električnih osobina digitalnog hardvera dovela je do razvoja nekoliko novih tipova podataka koji u suštini predstavljaju proširenja tipova *bit* i *bit_vector*. Ovi novi tipovi podataka definisani su u paketu *std_logic_1164*, koji je deo IEEE standarda broj 1164. U ovom odeljku biće predstavljeno nekoliko najvažnijih tipova podataka iz ovog paketa, zajedno sa pripadajućim operacijama i funkcijama za konverziju između novih i predefinisanih tipova podataka.

Std_logic i std_logic_vector - definišu 8-nivovski logički sistem. Dozvoljene vrednosti su:

´X´	Nepoznata vrednost	
´0´	Nizak nivo	(sintetiše se kao logička ´0´)
´1´	Visok nivo	(sintetiše se kao logička ´1´)
´Z´	Visoka impedansa	(sintetiše se kao trostatički bafer)
´W´	″Slaba″ nepoznata vrednost	
´L´	″Slab″ nizak nivo	
´H´	″Slab″ visok nivo	
´-´	Proizvoljna vrednost	



Sl. 3-1 interpretacija novoi signala tipa *std_logic*.

Tip podataka *std_logic* je prevashodno namenjen za modeliranje električnih veza u digitalnim sistemima (Sl. 3-1). U digitalnim sistemima, svaka fizička veza se najčešće pobuđuje izlazom samo jednog digitalnog kola (tzv. drajver veze). Drajver može da pobuđuje vezu logičkom vrednošću ´1´ ili ´0´. Takođe, izlaz drajvera može biti u stanju visoke impedanse ´Z´ (karakteristično za trostatičke bafere). U upotrebi su i digitalna kola sa izlazom tipa ″otvoreni kolektor″, koja zahtevaju ugradnju spoljašnjeg otpornika (tzv. *″pull-up″* otpornik). Drajver tipa otvorenog kolektora postavlja ″jaku″ ´0´, ali ″slabo″ ´1´, tj. slab visok nivo (´H´). Drajver koji zahteva ugradnju *″pull-down″* otpornika postavlja ″jako″ ´1´ i ″slabu″ ´0´, tj. slab nizak nivo (´L´).

T. 3-3 Tabela razrešavanja konflikta za tip *std_logic*.

	X	**0**	**1**	**Z**	**W**	**L**	**H**	**-**
X	X	X	X	X	X	X	X	X
0	X	0	X	0	0	0	0	X
1	X	X	1	1	1	1	1	X
Z	X	0	1	Z	W	L	H	X
W	X	0	1	W	W	W	W	X
L	X	0	1	L	W	L	W	X
H	X	0	1	H	W	W	H	X
-	X	X	X	X	X	X	X	X

U nekim slučajevima, više drajvera mogu da pobuđuju istu vezu (tipičan primer su magistrale). S obzirom na to što u jednom momentu na vezi može postojati samo jedan logički nivo, problem će se javiti onda kada drajveri veze generišu različite logičke vrednosti - ova pojava se naziva *konfliktom* na magistrali. Konflikti se razrešavaju tako što signal dobija vrednost ″najjačeg″ od svih logičkih nivoa kojima je veza pobuđena. Na

primer, ako neku vezu pobuđuju dva drajver, tako što je izlaz jednog u stanju visoke impedanse (´Z´), a drugi generiše ´1´, rezultujuća vrednost signala na liniji biće ´1´ (zato što je signal vrednosti ´1´ ″jači″ od signala vrednosti ´Z´). Ili, ako jedan drajver generiše ´1´, a drugi ´0´, rezultujuća vrednost biće nepoznata (´X´). Pravila za razrešavanje konflikta u logičkom sistemu tipa *std_logic* navedena su u tabeli T. 3-3. Treba napomentuti da se većina vrednosti iz tipa *std_logic* koriste samo u simulaciji. U VHDL kôdu koji je namenjen za sintezu, signalima ovog tipa je od raspoloživih osam dozvoljeno dodeljivati samo sledeće tri vrednosti: ´1´, ´0´ i ´Z´.

```
SIGNAL x : STD_LOGIC;
-- deklariše x kao jednobitni (skalarni) signal tipa std_logic
SIGNAL y : STD_LOGIC_VECTOR(3 DOWNTO 0) := ″0001″;
-- deklariše y kao 4-bitni vektor sa inicijalnom vrednošću ″0001″
-- Inicijalna vrednost je opciona, a dodeljuje se operatorom ″:=″ .
-- Krajnji levi bit je bit najveće težine (MSB).
SIGNAL w : STD_LOGIC_VECTOR(0 TO 7);
-- deklariše w kao 8-bitni vektor
-- Krajnji desni bit je bit najveće težine (MSB).
```

U gornjim primerima, signal *x* je skalar (može da sadrži samo jednu vrednost tipa *std_logic*), dok su signali *y* i *w* vektori (sadrže niz vrednosti tipa *std_logic*). Uočimo da su deklaracije vektorskog tip praćene konstrukcijom (*n downto m*), odnosno (*n to m*), koja definiše dužinu vektora i opseg indeksa. Pri tome, *n* i *m* su vrednosti početnog i krajnjeg indeksa u nizu. Za *downto* važi $n \geq m$, a za *to* $n \leq m$. Vektor sadrži $m-n+1$ elemenata, za *downto*, odnosno $n-m+1$ elemenata, za *to* varijantu. U gornjim primerima, *y* je vektor dužine 3-0+1=4, a *w* vektor dužine 7-0+1=8 bita. Svakom elementu vektora pridružen je indeks, koji ukazuje na njegovu poziciju u vektoru. Sližbene reči *downto* i *to* ukazuju na to da li se kretanjem kroz niz, s leva na desno, indeksi elemenata smanjuju ili povećavaju. Za *downto* redosled važi da krajnji levi element vektora ima najveći, a krajnji desni najmanji indeks. Obrnuto važi za *to* redosled - krajnji levi element ima najmanji, a krajnji desni najveći indeks. Kod tipova podataka koji se koriste za predstavljanje binarnih vrednosti (kao što su *bit_vector* ili *std_logic_vector*) element (bit) na poziciji indeksa najveće vrednosti naziva se bitom najveće težine (ili *MSB*, od eng. *Most Significant Bit*). Slično, bit na poziciji najmanjeg indeksa je bit najmanje težine (ili *LSB* od eng. *Last Significant Bit*).

Sledeće naredbe dodele su legalne:

```
x <= ´1´;
-- jednobitnom signalu x dodeljuje se vrednost ´1´. Za pisanje
-- jednobitnih vrednosti koriste se jednostruki znaci navoda (´ ´).
y <=″0111″;
-- 4-bitnom signalu se dodeljuje vrednost ″0111″ (MSB=´0´).
-- Vektorske vrednosti se pišu pod dvostrukim navodnicima (″ ″).
w <=″01110001″;
-- 8-bitnom signalu w dodeljuje se vrednost ″01110001″ (MSB=´1´).
```

Skalarne vrednosti se pišu pod jednostrukim, a vektorske pod dvostrukim navodnicima. Za obraćanje pojedinačnim elementima vektora koristi se indeksiranje. Na primer, elementi vektora *y*, za koji važi *downto* redosled, su *y*(3), *y*(2), *y*(1) i *y*(0), a njihove vrednosti su redom ´0´, ´1´, ´1´ i ´1´. Elementi vektora *w* (*to* redosled), su *w*(0), *w*(1), ..., *w*(7), a njihove vrednosti su ´0´, ´1´, ..., ´1´.

Redosled *downto* se obično koristi onda kad se vektor tumači kao binarni broj. Navikli smo da u zapisima binarnih brojeva krajni levi bit igra ulogu bit najveće težine. Tako, zapis ″0111″ iz naredbe *y* <=″0111″ predstavlja broj 7. Da je signal *y* bio deklarisan s indeksima u *to* redosledu (*signal y : bit_vector*(0 *to* 3)), tada bi vrednost binarnog broja ″0111″ bila 14.

Pr. 3-4 Vrednosti različitih tipova

```
x0 <= ´1´;           -- tip signala x0 može biti bit ili std_logic
x1 <= ″00011111″;    -- signal x1 može biti tipa bit_vector
                     -- ili std_logic_vector
x2 <= ″0001_1111″;   -- dozvoljena crta za podvlačenje
x3 <= ″101111″;      -- binarna reprezentacija decimalnog broja 47
x3 <= B″101111″;     -- binarna reprezentacija decimalnog broja 47
x3 <= O″57″;         -- oktalna reprezentacija decimalnog broja 47
x3 <= X″2F″;         -- heksadecimalna reprezentacija dec. broja 47
n <= 1200;           -- n je integer
IF ready THEN ...    -- ready je tipa boolean. THEN grana se izvršava
                     -- ako je ready = true
y <= 1.2E-5;         -- y je signal tipa real.
q <= d after 10 ns; -- ns je fizička jedinica za vreme
                     -- (nanosekunde).
```

Pr. 3-5 Signali i tipovi podataka

```
SIGNAL a: BIT;
SIGNAL b: BIT_VECTOR(7 DOWNTO 0);
SIGNAL c: STD_LOGIC;
SIGNAL d: STD_LOGIC_VECTOR(7 DOWNTO 0);
SIGNAL e: INTEGER RANGE 0 TO 255;
...
a <= b(5);  -- ispravno (a i b(5) su skalari tipa bit)
b(0) <= a;  -- ispravno (b(0) i a su skalari tipa bit)
a <= d(5);  -- neispravno (a - bit, d(5)- std_logic)
d(0) <= c;  -- ispravno (d(0) i c su skalari tipa std_logic)
a <= c;     -- neispravno (a - bit, c - std_logic)
b <= d;     -- neispravno (b - bit_vector, d - std_logic_vector)
e <= b;     -- neispravno (e - integer, b - bit_vector)
e <= d;     -- neispravno (e - integer, b - std_logic_vector)
```

Operacije nad tipom std_logic. Kao što je već rečeno, tipa podataka definiše skup vrednosti i skup operacija nad objektima tog tipa. U nekim slučajevima, VHDL dozvoljava da se funkcije i operatori istih imena koriste za operande različitih tipova. Drugim rečima, dozvoljeno je da postoji više funkcija (operatora) sa istim imenom, ali svaka za različit tip podataka. Ova mogućnost se naziva *preklapanje funkcija*, odnosno *preklapanje operatora*.

U paketu *std_logic_1164* svi logički operatori (*not, and, nand, or, nor, xor* i *xnor*), prvobitno definisani za tipove *bit* i *bit_vector*, preklopljeni su za tipove podataka *std_logic* i *std_logic_vector* (tabela T. 3-4). Drugim rečima, nad objektima tipa *std_logic* i

std_logic_vector dozvoljeno je primenjivati logičke operatore. Međutim, treba naglasiti da za tipove podataka *std_logic* i *std_logic_vector* aritmetički operatori nisu preklopljeni.

T. 3-4 Operacije nad tipom *std_logic*.

Preklopljeni operator	**Tip operanda *a***	**Tip operanda *b***	**Tip rezultata**
NOT a	std_logic std_logic_vector		isti kao *a*
a **AND** b	std_logic std_logic_vector	isti kao *a*	isti kao *a*
a **OR** b			
a **XOR** b			
a **NOR** b			
a **XNOR** b			

Pr. 3-6 Logičke operacije nad operandima tipa *std_logic_vector*

Logičke operacije nad vektorima izvršavaju se nezavisno nad svakim parom odgovarajućih bitova dva vektora. Pri tom vektori moraju biti iste dužine.

```
SIGNAL a, b, c: STD_LOGIC_VECTOR(7 DOWNTO 0);
. . .
a <= ″00111010″;
b <= ″10000011″;
c <= NOT a;   -- c dobija vrednost ″11000101″
c <= a OR b;  -- c dobija vrednost ″10111011″
c <= a XOR b; -- c dobija vrednost ″10111001″
```

***Konverzija tipa*.** Paket *std_logic_1164* sadrži i nekoliko funkcija za konverziju podataka iz tipa *bit* (*bit_vector*) u tip *std_logic* (*std_logic_vector*) i obrnuto. Relevantne funkcije su navedene u tabeli T. 3-5.

T. 3-5 Funkcije za konverziju iz paketa *std_logic_1164*.

Funkcija	**Tip operanda a**	**Tip rezultata**
to_bit(a)	std_logic	bit
to_stdlogic(a)	bit	std_logic
to_bitvector(a)	std_logic_vector	bit_vector
to_stdlogicvector(a)	bit_vector	std_logic_vector

Pr. 3-7 Konverzija tipa

Pretpostavimo da su *s*1, *s*2, *s*3, *b*1 i *b*2 signali definisani na sledeći način:

```
SIGNAL s1, s2, s3 : STD_LOGIC_VECTOR(7 DOWNTO 0);
SIGNAL b1, b2 : BIT_VECTOR(7 DOWNTO 0);
```

Sledeće naredbe dodele su neispravne zbog neusklađenosti tipova:

```
s1 <= b1;         -- signalu tipa std_logic_vector se dodeljuje
                  -- vrednost signala tipa bit_vector
b2 <= s1 AND s2; -- vrednost tipa std_logic_vector se dodeljuje
                  -- signalu tipa bit_vector
s3 <= b1 OR s2;   -- operacija or nije definisana između vrednosti
                  -- tipa bit_vector i std_logic_vector
```

Greške iz prethodnih naredbi se mogu ispraviti korišćenjem funkcija za konverziju tipa:

```
s1 <= to_stdlogicvector(b1);
b2 <= to_bitvector(s1 AND s2);
s3 <= to_stdlogicvector(b1) OR s2;
```

Poslednja naredba se može napisati i u sledećem obliku:

```
s3 <= to_stdlogicvector(b1 OR to_bitvector(s2));
```

3.3.4. Operacije nad vektorskim tipovima

U VHDL-u postoji nekoliko operacija koje su definisane direktno nad jednodimezionim (vektorskim) tipovima podataka. Primeri takvih operacija su konkatenacija, relacione operacije i agregacije. Iako se mogu primeniti na bilo koji vektorski tip, ove operacije će biti predstavljene na primeru tipa *std_logic_vector*.

Relacione operacije nad vektorima. U VHDL-u, relacione operacije (>, <, =, <=, >= i /=) se mogu primenjivati nad objektima vektorskog tipa. Vektori nad kojima se primenjuje relacija moraju sadržati elemente istog tipa, ali ne moraju obavezno biti iste dužine. Vektori se porede element po element, počev od elemenata sa krajnjih levih pozicija. Poređenje se nastavlja sve do pozicije kada se konačni rezultat poređenja može sa sigurnošću odrediti. Ako se tokom poređenja dostigne kraj jednog vektora, taj vektor se smatra ″manjim″. Na primer, sve operacije koje slede su tačne:

```
″101″ = ″101″, ″011″ > ″010″, ″011″ > ″00011″, ″0110″ > ″011″
```

Konkatenacija. Konkatenacija se koristi za nadovezivanje (spajanje) manjih vektora, segmenata vektora ili skalara u jedan veći vektor. Operator konkatenacije je znak ″&″. Na primer:

```
y <= x & "10000000";
-- ako je x = ´1´, tada y <= ″11000000″
y <= ″00″ & a(7 DOWNTO 2);
-- ekvivalentno pomeranju vektora a za dve pozicije udesno
y <= a(1 DOWNTO 0) & a(7 DOWNTO 2);
-- ekvivalentno rotaciji vektora a za dve pozicije udesno
```

Pr. 3-8 Grupisanje signala

Neka su deklarisani signali:

```
SIGNAL dbus: STD_LOGIC_VECTOR(0 TO 7);
SIGNAL ctrl: STD_LOGIC_VECTOR(0 TO 1);
SIGNAL en, rw: STD_LOGIC;
SIGNAL count: STD_LOGIC_VECTOR(0 TO 3);
```

Namera je da se signali *ctrl, count, en i rw* objedine u jedinstveni 8-bitni vektor *dbus*. To se može postići pomoću četiri naredbe dodele:

```
dbus(0 TO 1) <= ctrl;
dbus(2) <= en;
dbus(3) <= rw;
dbus(4 TO 7) <= count;
```

Korišćenjem konkatenacije isti rezultat se dobija pomoću samo jedne naredbe dodele:

```
dbus <= ctrl & en & rw & count;
```

Pr. 3-9 Pomeranje pomoću konkatenacije

Razmotrimo sledeći segment VHDL kôda:

```
SIGNAL a, b: STD_LOGIC_VECTOR(7 DOWNTO 7);
. . .
a <= b(5 DOWNTO 0) & "00";
```

Naizgled efekat je isto kao *a* <= *b sll* 2 (*sll* je operator pomeranja ulevo). To bi i bio slučaj da se radi o operandima tipa *bit_vector*. Međutim, za operande tipa *std_logic_vector* rezultat operacije *a* <= *b sll* 2 je sledeći:

```
a <= b(5 DOWNTO 0) & "XX";
```

To je zato što je u definiciji tipa *std_logic* kao prva vrednost navedena vrednost ´X´, a ne vrednost ´0´ (v. 3.3.3 i Pr. 3-3). Vrednost ´X´ (nepoznata vrednosti) se ne može sintetizovati, što je i razlog zašto se operator *sll* (kao ni bilo koji drugi operator pomeranja, v. T. 3-1) ne može koristiti u sintezi tipa *std_logic_vector*. Zbog toga se za pomeranje vektora ovog tipa koristi konkatenacija, na način kako je to pokazano u ovom primeru.

Agregacija. Agregacija je jezička konstrukcija iz VHDL-a koja se koristi za dodelu vrednosti objektu vektorskog tipa. Razmotrimo sledeći segment VHDL kôda:

```
1  SIGNAL w: STD_LOGIC_VECTOR(7 DOWNTO 0);
2  . . .
3  w <= "00001001";
4  w <= ('0','0', '0', '0', '1', '0', '0', '1');
5  w <= (7=>'0',1=>'0',5=>'0',6=>'0',0=>'1',4=>'0',3=>'1',2=>'0');
6  w <= (7|6|5|4|2|1 =>'0', 3|2 =>'1');
7  w <= (0=>'1', 3=>'1', OTHERS => '0')
```

Notacija "00001001" iz naredbe dodele *w* <= "00001001" (linija 3) predstavlja kompaktnu reprezentaciju 8-bitnog binarnog vektora. Sledeće četiri naredbe (linije 4 - 7) predstavljaju različite oblike agregacije i ima isti efekat kao naredba iz linije 3. U naredbi dodele iz linije 4, vrednost vektora *w* je agregacija pojedinačnih vrednosti počev od krajnjeg levog do krajnjeg desnog elementa vektora. U naredbi dodele iz linije 5, pojedinačnim elementima vektora se direktno pridružuju vrednosti notacijom *i* => ´*v*´, gde je *i* indeks elementa u vektoru, a *v* vrednost koja se dodeljuje tom elementu. Na primer, 0=>'1' znači da se elementu vektora sa indeksom 0, dodeljuje vrednost '1' (isto kao *w*(0)<=´1´). Uočimo da redosled parova *i* => ´*v*´ nije od značaja. Indeksi elemenata kojim se dodeljuje ista vrednost mogu biti grupisani, kao u naredbi iz linije 6. Konačno, u ovakvom načinu predstavljanja vektora, službena reč *others* ima posebno značenje, jer može da zameni sve indekse kojim nije eksplicitno dodeljena vrednost, kao u naredbi iz linije 7. Ova mogućnost je posebno bitna u slučajevima kad dužina vektora nije unapred poznata. Na primer, naredba:

```
w <= (OTHERS => '0');
```

inicijalizuje vektor *w* na sve nule. Zapazimo da će prethodna naredba ostati ista čak i ako se naknadno promeni dužina vektora *w*.

3.3.5. Tipovi podataka iz paketa *IEEE numeric_std*

Paket *std_logic_1164* ne pruža podršku za aritmetičke operacije. Na primer, sabiranje i oduzimanje nisu dozvoljene operacije nad vektorima tipa *std_logic_vector*. Predefinisane aritmetičke operacije (v. T. 3-1) se mogu primenjivati samo nad operandima tipa *integer*, npr.:

```
SIGNAL a, b, r : INTEGER;
. . .
r <= a + b;
```

Naredbu sabiranja iz prethodne naredbe teško je sintetizovati u hardver, zato što u kôdu nije eksplicitno naznačen opseg operanada *a* i *b*, pa nije ni poznato koliko bita je potrebno za njihovo predstavljanje. Informacija o ″dužini″ operanada, premda nebitna za simulaciju, od suštinske je važnosti za sintezu. Razlika u hardverskoj složenosti 8-bitnog i 32-bitnog sabirača je ogromna. Bolji i prirodniji pristup je da se za operande u aritmetičkim operacijama umesto tipa *integer* koriste nizovi (vektori) binarnih vrednosti ´0´ i ´1´ koji će se interpretirati kao neoznačeni ili označeni binarni brojevi. Paket *numeric_std* je razvijen upravo s namerom da se nad binarnim vektorima omogući izvođenje aritmetičkih operacija.

Označeni i neoznačeni tipovi podataka. Paket *numeric_std* uvodi dva nova tipa podataka, *signed* i *unsigned*. Oba tipa su nizovi elemenata tipa *std_logic*. Niz tipa *unsigned* se interpretira kao neoznačen binarni broj (tj. ceo broj koji nikada nije manji od nule), a niz tipa *signed* kao označen binarni broj (tj. ceo broj koji može biti i pozitivan i negativan) i to u reprezentaciji potpunog komplementa s bitom znaka na krajnjoj levoj poziciji.

Uočimo da su tipovi *unsigned* i *signed*, baš kao i tip *std_logic_vector*, definisani kao nizovi elemenata tipa *std_logic*. Vizuelno, vrednosti ova tri tipa izgledaju isto. Međutim, razlika postoji u njihovoj interpretaciji. Na primer, niz bitova ″1100″ predstavlja broj 12 ako se interpretira kao neoznačen (*unsigned*) broj, odnosno broj -4 ako se interpretira kao označen (*signed*) broj, ili, prosto, predstavlja 4 nezavisna bita spojena u jedan niz, ako se interpretira kao vrednost tipa *std_logic_vector*. Deklaracija signala tipa *unsigned* i *signed* je slična deklaraciji signala tipa *std_logic_vector*, npr.:

```
SIGNAL a, b : SIGNED(7 DOWNTO 0);
SIGNAL x : UNSIGNED(7 DOWNTO 0);
```

Preklopljeni operatori. U paketu *numeric_std* preklopljeni su svi relevantni aritmetički operatori, +, -, *, /, *mod* i *rem*, i to tako da se mogu primenjivati nad parovima operanada tipa *unsigned* i *unsigned*, *unsigned* i *natural*, *signed* i *signed*, kao i *signed* i *integer*. Lista operatora iz paketa *numeric_std* data je tabeli T. 3-6. Na primer, sve sledeće naredbe dodele su ispravne:

```
SIGNAL au, bu, cu, du, eu : UNSIGNED(7 DOWNTO 0);
SIGNAL as, bs, cs, ds : SIGNED(7 DOWNTO 0);
. . .
au <= bu + cu;             -- unsigned i unsigned
du <= cu + 1;              -- unsigned i natural
eu <= (3 + au + bu) - cu;  -- unsigned i natural
as <= bs + cs;             -- signed i signed
ds <= bs - 1;              -- signed i integer
```

Relacioni operatori, =, /=, <, >, <= i >=, su takođe preklopljeni u paketu *numeric_std* i to na način da se mogu primenjivati ne samo na operande istog tipa (*unsigned-unsigned* i *signed-signed*) već i na parove operanada tipa *unsigned-natural* i *signed-integer*. Takođe, u

proceduri poređenja koju sprovode preklopljeni operatori dva niza se tretiraju kao dva binarna broja. Na primer, razmotrimo izraz: ″011″ > ″1000″. Pod pretpostavkom da su operandi tipa *std_logic_vector*, rezultat poređenja će biti netačan (zato što je prvi levi element niza ″011″ manji od prvog levog elementa niza ″1000″). Ako su operandi tipa *unsigned*, primeniće se preklopljeni operator > iz paketa *numeric_std,* koji tretira nizove ″011″ i ″1000″ kao brojeve 3 i 8, a rezultat poređenja će biti takođe netačan. Međutim, ako su operandi tipa *signed*, preklopljeni operator > protumačiće nizove ″011″ i ″1000″ kao označene brojeve 3 i -4, a rezultat poređenja će biti tačan.

Uočimo da paket *numeric_std* ne sadrži definicije preklopljenih logičkih operatora. To znači da primena logičkih operacija nad objektima tipa *unsigned* i *signed* nije dozvoljena.

T. 3-6 Preklopljeni operatori iz paketa *numeric_std*.

Operacija	Opis	Tip operanda *a*	Tip operanda *b*	Tip rezultata
Aritmetičke operacije				
a + b	sabiranje	unsigned unsigned, natural signed signed, integer	unsigned, natural unsigned signed, integer signed	unsigned unsigned signed signed
a **-** b	oduzimanje			
a * b	množenje			
a / b	deljenje			
a **MOD** b	moduo			
a **REM** b	ostatak			
ABS a	apsolutna vrednost	signed		signed
- a	negacija			
Operacije poređenja				
a = b	jednako	unsigned unsigned, natural signed signed, integer	unsigned, natural unsigned signed, integer signed	boolean boolean boolean boolean
a /= b	različito			
a < b	manje			
a > b	veće			
a <= b	manje ili jednako			
a >= b	veće ili jednako			

Pr. 3-10 Dozvoljene i nedozvoljene operacije

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.NUMERIC_STD.ALL;
. . .
SIGNAL a, b, c: SIGNED(7 DOWNTO 0);
SIGNAL x, y, z: STD_LOGIC_VECTOR(7 DOWNTO 0);
. . .
c <= a + b;
-- ispravno, aritmetičke operacije su dozvoljene nad tipom signed
c <= a AND b;
-- neispravno, logičke operacije nisu dozvoljene nad tipom signed
z <= x + y;
-- neispravno, aritmetičke operacije nisu dozvoljene nad
-- tipom std_logic_vector
z <= x AND y;
-- ispravno, logičke operacije su dozvoljene u tipu std_logic_vector
```

Konverzija tipa. Paket *numeric_std* sadrži tri funkcije za konverziju tipa: *to_unsigned*, *to_signed* i *to_integer*. Funkcija *to_integer* konvertuje (prevodi) objekat tipa *unsigned* ili *signed* u ceo broj, tj. u objekat tipa *integer*. Funkcije *to_unsigned* i *to_signed* konvertuju ceo broj (*integer*) u objekat tipa *unsigned*, odnosno *signed* sa zadatim brojem bita. Funkcije *to_unsigned* i *to_signed* imaju dva ulazna parametra. Prvi je ceo broj koji se konvertuje, dok drugi, takođe ceo broj, specificira broj bita u rezultujućem bit-vektoru *unsigned* ili *signed* tipa.

Za konverziju podataka između tipova *std_logic_vector*, *unsigned* i *signed* koristi se princip eksplicitne konverzije (engl. *type casting*). To znači da se objekat jednog tipa prevodi u drugi tip tako što se omeđi zagradama sa imenom novog tipa podataka ispred zagrade, npr.:

```
SIGNAL u1, u2 : UNSIGNED(7 DOWNTO 0);
SIGNAL v1, v2 : STD_LOGIC_VECTOR(7 DOWNTO 0);
. . .
u1<= UNSIGNED(v1);          -- konvertuje std_logic_vector u unsigned
v2<= STD_LOGIC_VECTOR(u1); -- konvertuje unsigned u std_logic_vector
```

Spisak funkcija za konverziju iz paketa *numeric_std* dat je u tabeli T. 3-7. Treba zapaziti da ne postoji funkcija za konverziju iz tipa *std_logic_vector* u tip *integer* i obrnuto. Razlog za to je jednostavan - objekti tipa *std_logic_vector* se ne mogu interpertirati kao brojevi.

T. 3-7 Podrška za konverziju tipova podataka iz paketa *numeric_std*.

Iz tipa	U tip	Funkcija za konverziju / eksplicitna konverzija
unsigned, signed	std_logic_vector	std_logic_vector(a)
signed, std_logic_vector	unsigned	unsigned(a)
unsigned, std_logic_vector	signed	signed(a)
unsigned, signed	integer	to_integer(a)
natural	unsigned	to_unsigned(a, size)
integer	signed	to_signed(a, size)

Pr. 3-11 Konverzija tipova

Pretpostavimo da su signali deklarisani na sledeći način:

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.NUMERIC_STD.ALL
. . .
SIGNAL s1, s2, s3, s4, s5, s6: STD_LOGIC_VECTOR(7 DOWNTO 0);
SIGNAL u1, u2, u3, u4, u5, u6, u7: UNSIGNED(7 DOWNTO 0);
SIGNEL sg: SIGNED(7 DOWNTO 0);
. . .
```

Sledeće naredbe dodele su ispravno jer je operator + definisan za tipove *unsigned* i *natural* (v. T. 3-6):

```
u3 <= u2 + u1;      -- ispravno, operandi su tipa unsigned
u4 <= u2 + 1;       -- ispravno, operandi su tipa unsigned i natural
```

Međutim, zbog neusklađenosti tipova, sledeće dve naredbe dodele **nisu** ispravne:

```
u5 <= sg;     -- neispravno, neusklađeni tipovi
u6 <= 5;      -- neispravno, neusklađeni tipovi
```

Neophodno je primeniti funkciju za konverziju tipa ili eksplicitnu konverziju:

```
u5 <= UNSIGNED(sg);         -- ispravno, eksplicitna konverzija
u6 <= TO_UNSIGNED(5,8);     -- ispravno, funkcija za konverziju
```

Aritmetički operatori iz paketa *numeric_std* nisu definisani za slučaj kad je jedan operand tipa *signed*, a drugi *unsigned*:

```
u7 <= sg + u1;       -- neispravno,neusklađeni tipovi
```

Neophodno je obaviti konverziju operanda *sg* u tip *unsigned*:

```
u7 <= UNSIGNED(sg) + u1;   -- ispravno, ali rizično
```

Iako je prethodna naredba ispravna, treba voditi računa o tome da je interpretacija podataka tipova *signed* i *unsigned* različita. Na primer, ″11111111″ je -1 za tipa *signed*, a 255 za tipa *unsigned*.

Signalima tipa *std_logic_vector* se ne može direktno dodeliti vrednost signala tipa *signed*, *unsigned* ili *integer*:

```
s3 <= u3;     -- neispravno, neusklađenost tipova
s4 <= 5;      -- neispravno, neusklađenost tipova
```

Neophodna je konverzija tipa:

```
s3 <= STD_LOGIC_VECTOR(u3); -- ispravno, eksplicitna konverzija
s4 <= STD_LOGIC_VECTOR(TO_UNSIGNED(5,8)); -- ispravno,
                                          -- eksplicitna konverzija
```

Uočimo da su u drugoj od dve gornje naredbe upotrebljene dve konverzije. Razlog za to je što za tip *integer* nije podržana eksplicitna konverzija u tip *std_logiv_vector*. Konstanta 5 se prvo konvertuje u 8-bitnu vrednost tipa *unsigned*, a zatim se dobijena vrednost eksplicitno konvertuje u tip *std_logic_vector*.

Aritmetičke operacije se ne mogu primenjivati na tip *std_logic_vector*:

```
s5 <= s2 + s1; -- neispravno, + nije definisano za std_logic_vector
s6 <= s2 + 1;  -- neispravno, + nije definisano
               -- za datu kombinaciju tipova
```

Da bi se problem rešio, potrebno je da se operandi prvo konvertuju u tip *unsigned* (ili *signed*), zatim da se izvrši sabiranje i konačno da se rezultat vrati nazad u tip *std_logic_vector*:

```
s5 <= std_logic_vector(unsigned(s2) + unsigned(s1)); -- ispravno
s6 <= std_logic_vector(unsigned(s2)) + 1);           -- ispravno
```

3.3.6. Nestandardni aritmetički paketi

Za rad sa aritmetičkim operacijama u VHDL-u, osim paketa *numeric_std*, raspoloživo je još nekoliko drugih paketa, a pre svega:

- *std_logic_arith*
- *std_logic_unsigned*, i
- *std_logic_signed*

Ovi paketi nisu deo standarda IEEE i danas se sve ređe koriste u praksi. Međutim, oni su i dalje podržani od strane većine alata za sintezu i mogu se često videti u postojećem VHDL

kôdu, što je i razlog zbog kojeg su u ovoj knjizi uvršteni u pregled tipova podataka. Budući da je paket *numeric_std* deo standarda IEEE i da pruža više mogućnosti u odnosu na pomenuta tri paketa, za rad sa aritmetičkim operacijama i kolima u ovoj knjizi biće korišćen paket *numeric_std*.

Std_logic_arith. Paket *std_logic_arith*, slično paketu *numeric_std*, definiše dva nova tipa podataka, *unsgned* i *signed*, zajedno sa preklopljenim operatorima +, - i *. Ovaj paket takođe sadrži i funkcije za konverziju, premada se njihova imena razlikuju u odnosu na odgovarajuće funkcije iz paketa *numeric_std*. Kao i za paket *numeric_std*, logičke operacije nisu dozvoljene nad podacima tipa *unsigned* i *signed* (v. Pr. 3-10).

Std_logic_unsigned i std_logic_signed. Paketi *std_logic_unsigned* i *std_logic_signed* ne uvode nove tipove podataka, već samo definišu preklopljene aritmetičke operatore za tip *std_logic_vector*. Drugim rečima, ovi paketi omogućavaju da se s objektima tipa *std_logic_vector* manipuliše na isti način kao sa objektima tipa *unsigned* (iz paketa *std_logic_unsigned*) odnosno *signed* (iz paketa *std_logic_signed*). Na taj način, eliminisana je potreba za konverzijom tipa (*std_logic_vector* u *unsigned*/*signed* i obrnuto). Jasno je da ova dva paketa ne mogu da se koriste u isto vreme.

Pr. 3-12 Tip *std_logic_unsigned*

Uključivanjem u projekat paketa *std_logic_unsigned* (ili *std_logic_signed*) proširiju se mogućnosti tipa *std_logic_vector*, koji tako dobija podršku za aritmetičke operacije:

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;
. . .
SIGNAL a, b, c: STD_LOGIC_VECTOR(7 DOWNTO 0);
. . .
c <= a + b;   -- ispravno, aritmetičke operacije su dozvoljene
c <= a AND b; -- ispravno, logičke operacije su dozvoljene
```

Iako, bar na početku, rad sa paketima *std_logic_unsigned* i *std_logic_signed* izgleda lakši nego sa *numeric_std*, ova dva paketa nisu standardizovana i njihovo korišćenje može da dovede do problema kompatabilnosti.

3.3.7. Korisnički tipovi podataka

VHDL omogućava projektantu da definiše svoje sopstvene tipove podataka (tzv. *korisnički tipovi*). Postoje dve kategorije korisničkih tipova podataka: celobrojni (*integer*) i nabrojivi (engl. *enumeration*).

Korisnički celobrojni tipovi. Korisnički celobrojni tipovi podataka su podskupovi tipa *integer* definisani opsegom (*range*) celih brojeva, npr.:

```
TYPE integer IS RANGE -2147483647 TO +2147483647;
-- ekvivalentno predefinisanom tipu integer
TYPE natural IS RANGE 0 TO +2147483647;
-- ekvivalentno predefinisanom tipu natural
TYPE mali_integer IS RANGE -32 TO 32;
-- podskup celih brojeva iz opsega -32 do 32
TYPE ocena IS RANGE 5 TO 10;
-- podskup celih brojeva iz opsega 5 do 10
```

Korisnički celobrojni tipovi se koriste da bi se ograničio opseg vrednosti koje se mogu dodeliti celobrojnoj varijabli ili signalu:

```
VARIABLE markova_ocena, anina_ocena : ocena;
markova_ocena := 8; -- ispravno, dozvoljena ocena
anina_ocena := 11;  -- grešaka, ocena ne može biti veća od 10.
                    -- (kompajler će prijaviti grešku)
```

Korisnički nabrojivi tipovi. Slede četiri primera deklaracije korisničkih nabrojivih tipova podataka:

```
TYPE bit IS (´0´, ´1´);
-- ekvivalentno predefinisanom tipu bit
TYPE nasa_logika IS (´0´, ´1´, ´Z´);
-- podskup vrednosti tipa std_logic
TYPE bit_vector IS ARRAY (NATURAL RANGE <>) OF BIT;
-- ekvivalentno predefinisanom tipu BIT_VECTOR
-- RANGE <> označava opseg bez eksplicitno navedenih granica
-- NATURAL RANGE <> ograničava RANGE <> na opseg NATURAL tipa.
-- ARRAY - službena reč za deklaraciju niza.
TYPE stanje IS (iskljuceno, napred, nazad, stop);
-- definiše nabrojiv tip podataka
-- dozvoljene vrednosti su eksplicitno navedene
-- (tipična upotreba za simbolička imena stanja konačnih automata)
TYPE boja IS (crvena, zelena, plava, bela);
-- još jedan primer nabrojivog tipa
```

Deklaracija nabrojivog tipa definiše novi tip podataka koji čini skup korisnički-definisanih vrednosti. Na primer, u deklaraciji tipa *boja* eksplicitno su navedena imena vrednosti koje su pridružene ovom tipu (*crvena*, *zelena*, *plava* i *bela*). Ove vrednosti su apstraktne, jer ne predstavljaju ni brojnu vrednost ni fizičku veličinu, već imaju značenje koje je poznato samo korisniku (projektantu). Korišćenje korisničkih tipova poboljšava čitljivost (razumljivost) kôda i smanjuje mogućnost greške u pisanju kôda.

Treba napomenuti da je svakoj vrednosti iz deklaracije nabrojivog tipa implicitno pridružen redni broj koji odgovara njenoj poziciji u spisku navedenih vrednosti. Tako, vrednost *crvena* iz poslednje navedene deklaracije ima redni broj 0, *zelena* 1 itd. Poredak vrednosti je od značaja ako se one koriste u relacionim izrazima (<, >, =, ...). Tako je izraz *crvena* < *plava* tačan, a izraz *bela* < *zelena* netačan. Napomenimo da su predefinisani tipovi *bit* i *std_logic* takođe nabrojivi tipovi.

3.3.8. Podtipovi

Podtip (*subtype*) je tip sa uvedenim ograničenjem. Ograničenje definiše podskup vrednosti osnovnog tipa koje su pridružene podtipu. Podtip nasleđuje sve operacije osnovnog tipa.

Sledeći podtipovi su izvedeni iz tipova koji su deklarisani u prethodnom odeljku.

```
SUBTYPE natural IS INTEGER RANGE 0 TO INTEGER´HIGH;
-- prirodni brojevi su podtip (podskup) celih brojeva
-- integer'high znači ″najveća vrednosti tipa integer″

SUBTYPE nasa_logika IS STD_LOGIC RANGE ´0´ TO ´Z´;
-- std_logic = (´X´,´0´,´1´,´Z´,´W´,´L´,´H´,´-´)
-- nasa_logika = (´0´,´1´,´Z´)
```

```
SUBTYPE nasa_boja IS boja RANGE crvena TO plava;
-- boja = (crvena, zelena, plava, bela)
-- nasa_boja = (crvena, zelena, plava)
SUBTYPE mali_integer IS INTEGER RANGE -32 TO 32;
-- podtip tipa integer
```

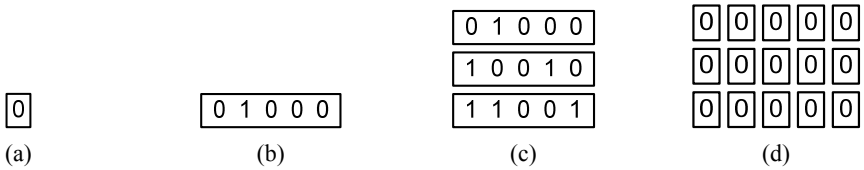
Uporedimo definiciju **podtipa** *mali_integer* sa definicijom istoimenog **tipa** iz prethodnog odeljka. Oba ova tipa su izvedena iz tipa *integer*, imaju identičan skup vrednosti i identičan skup dozvoljenih operacija. Čini se da se radi o ekvivalentnim deklaracijama. Međutim, postoji jedna bitna razlika. Vrednosti **tipa** *mali_integer* se ne mogu kombinovati sa vrednostima tipa *integer* u operacijama kao što su ´+´ i ´-´. S druge strane, dozvoljeno je u istom izrazu kombinovati vrednosti **podtipa** *mali_integer* i vrednosti tipa *integer*.

Pr. 3-13 Tipovi i podtipovi

```
TYPE nova_logika IS STD_LOGIC RANGE ´0´ TO ´1´;
SUBTYPE nasa_logika IS STD_LOGIC RANGE ´0´ TO ´1´;
SIGNAL a: BIT;
SIGNAL b: STD_LOGIC;
SIGNAL c: nasa_logika;
SIGNAL d: nova_logika;
...
b <= a;     -- neispravno (b je std_logic, a je bit)
b <= c;     -- ispravno (isti osnovni tip)
b <= d;     -- neispravno (b je std_logic, d je nova_logika)
```

3.4. Polja

Polja su kolekcije objekata istog tipa. Polja mogu biti jednodimenziona (1D ili vektori), dvodimenziona (2D ili matrice) ili 1D x 1D. Polja viših dimenzija (npr. 3D ili 2D x 2D) se ređe koriste u praksi i obično nisu podržana u alatima za sintezu. Na Sl. 3-2 je ilustrovana organizacija polja različitih dimenzionalnosti. Pojedinačna vrednost (ili skalar) prikazana je na Sl. 3-2(a), vektor (1D polje) na Sl. 3-2(b), polje vektora (1D x 1D polje) na Sl. 3-2(c) i polje skalara (2D polje) na Sl. 3-2(d).



Sl. 3-2 (a) skalar, (b) 1D, (c) 1D x 1D polje, (d) 2D polje.

Svi predefinisani tipovi podataka, kao i tipovi podataka iz paketa *std_logic_1164* i *numeric_std* su ili skalari ili vektori (tj. jednodimenziona polja skalara). Sledeći predefinisani tipovi se mogu sintetizovati (tj. mogu se koristiti u kôdu koji je namenjen za sintezu):

- Skalari: *bit, std_logic* i *boolean*.
- Vektori: *bit_vector*, *std_logic_vector*, *integer*, *signed* i *unsigned*.

Ne postoje predefinisana 2D ili 1D x 1D polja. Ako su neophodna, takva polja mogu biti definisana od strane projektanta. Novo polje se definiše shodno sledećoj sintaksi:

```
TYPE ime_tipa IS ARRAY (opseg_indeksa) OF tip;
```

Pr. 3-14 1D x 1D polje

Naš zadatak je da definišemo polje od četiri vektora dužine osam bita. Pojedinačne vektore nazvaćemo *vrsta*, a celokupnu strukturu *matrica*. Takođe, usvojićemo da je bit najveće težine (MSB) svakog vektora njegov krajnji levi bit i da je prva vrsta matrice vrsta 0. Postavljenim zahtevima odgovaraju sledeće deklaracije:

```
TYPE vrsta IS ARRAY (7 DOWNTO 0) OF STD_LOGIC; -- 1D polje
TYPE matrica IS ARRAY (0 TO 3) OF vrsta;       -- 1D x 1D polje
SIGNAL x: matrica;                             -- 1D x 1D signal
```

Identično 1D x 1D polje može se deklarisati i u jednoj liniji, na sledeći način:

```
TYPE matrica IS ARRAY (0 TO 3) OF STD_LOGIC_VECTOR(7 DOWNTO 0);
```

Pr. 3-15 2D polje

Polje definisano sledećom linijom kôda je ″pravo″ dvodimenziono (2D) polje, u potpunosti zasnovano na skalarima, a ne na vektorima, kao u prethodnom primeru:

```
TYPE matrica2D IS ARRAY (0 TO 3, 7 DOWNTO 0) OF STD_LOGIC;
```

Pr. 3-16 Inicijalizacija polja

Dodela inicijalnih vrednosti signalima i varijablama je opciona. Onda kada je potrebna, može se obaviti kao u sledećim primerima:

```
... := "0001";                                          -- 1D
... := (´0´, ´0´, ´0´, ´1´);                            -- 1D
... := ((‘0´, ´1’, '0', '1'), ('1', '1', '0', '1')); -- 1D x 1D
```

Pr. 3-17 Polja u naredbama dodele

```
TYPE vrsta IS ARRAY (7 DOWNTO 0) OF STD_LOGIC;
TYPE polje1 IS ARRAY (0 TO 3) OF vrsta;
TYPE polje2 IS ARRAY (0 TO 3) OF STD_LOGIC_VECTOR(7 DOWNTO 0);
TYPE polje3 IS ARRAY (0 TO 3, 7 DOWNTO 0) OF STD_LOGIC;
SIGNAL x: vrsta;
SIGNAL y: polje1;
SIGNAL v: polje2;
SIGNAL w: polje3;
-- Ispravne skalarne naredbe dodele: -------------------------------
-- Sledeće skalarne (jednobitne) naredbe dodele su ispravne,zato što
-- je osnovni (skalarni) tip svih signala (x,y,v,w) std_logic
x(0) <= y(1)(2);    -- uočimo dva para zagrada (y je 1Dx1D)
x(1) <= v(2)(3);    -- dva para zagrada (v je 1Dx1D)
x(2) <= w(2,1);     -- jedan par zagrada (w je 2D)
y(1)(1)<= x(6);
y(2)(0)<= v(0)(0);
```

```
y(0)(0)<= w(3,3);
w(1,1) <= x(7);
w(3,0) <= v(0)(3);
-- Vektorske naredbe dodele: ---------------------------------------
x <= y(0);              --ispravno, isti tip podataka: vrsta
x <= v(1);              --neispravno, neusklađeni tipovi: vrsta x
                        --std-logic-vector)
x <= w(2);              --neispravno (pogrešno indeksiranje, w je 2D)
x <= w(2, 2 DOWNTO 0);      --neispravno, neusklađeni tipovi
v(0) <= w(2, 2 DOWNTO 0);   --neispravno, neusklađeni tipovi:
                            --std_logic_vector v i std_logic w
v(0) <= w(2);               --neispravno, w je 2D
y(1) <= v(3);               --neispravno, neusklađeni tipovi
y(1)(7 DOWNTO 3)<= x(4 DOWNTO 0); -- ispravno, isti tip i veličina
v(1)(7 DOWNTO 3)<= v(2)(4 DOWNTO 0); -- ok, isti tip i veličina
w(1,5 DOWNTO 1) <= v(2)(4 DOWNTO 0); -- neispravno, neuskl. tipovi
```

Obratimo pažnju kako se pristupa elementima polja različitih dimenzionalnosti. Za pristup 1D polju (vektoru) koristi se jednodimenzioni indeks (npr. *vrsta*(2)). Za pristup 1Dx1D polju koristi se par jednodimenzionih indeksa (npr. *y*(4)(2) - prvi indeks, (4), se odnosi na vektor, drugi, (2), na element u vektoru). Elementima matrice se pristupa dvodimenzionim indeksom (npr. *w*(2,3)). Takođe, VHDL omogućava pristup delovima polja. Na primer, *v*(2)(4 *downto* 0) označava sve bitove od četvrtog do nultog, drugog vektora polja *v*.

3.5. Atributi

Atributi su dodatne informacije pridružene tipovima podataka, signalima i drugim objektima deklarisanim u VHDL kôdu. Postoje brojni predefinisani atributi, ali i mogućnost da projektant definiše svoje, korisničke atribute. Za referenciranje atributa koristi se apostrof, npr. *x´attr* (odnosi na atribut *attr* tipa ili objekta *x*). Razmotrićemo dve grupe atributa:

- atributi vektora: sadrže informacije koje se odnose na vektor
- atributi signala: služe za nadgledanje signala

***Atributi vektora*.** U VHDL kôdu za sintezu dozvoljeni su sledeći predefinisani atributi vektora:

d´LOW	vraća najmanji indeks vektora d
d´HIGH	vraća naveći indeks vektora d
d´LEFT	vraća krajnji levi indeks vektora d
d´RIGHT	vraća krajnji desni indeks vektora d
d´LENGTH	vraća veličinu (dužinu) vektora d
d´RANGE	vraća opseg vektora d
d´REVERSE_RANGE	vraća inverzni opseg vektora d

Pr. 3-18 Atributi vektora

Neka je signal *d* deklarisan na sledeći način:

```
SIGNAL d : STD_LOGIC_VECTOR(7 DOWNTO 0);
```

Tada važi:

```
d´LOW = 0        d´LENGTH = 8
d´HIGH = 7       d´RANGE = (7 DOWNTO 0)
d´LEFT = 7       d´REVERSE_RANGE = (0 TO 7)
d´RIGHT = 0
```

Pr. 3-19 Primena atributa vektora

Neka je dat signal:

```
SIGNAL d : STD_LOGIC_VECTOR(0 TO 7);
```

Sledeće četiri *for loop* naredbe (realizuju petlju u VHDL kôdu) ekvivalentne su i mogu se sintetizovati (brojač *i* uzima redom sve vrednosti iz pridruženog opsega):

```
FOR i IN RANGE (0 TO 7) LOOP . . .
FOR i IN d´RANGE LOOP . . .
FOR i IN RANGE (d´LOW TO d´HIGH) LOOP . . .
FOR i IN RANGE (0 TO d´LENGTH-1) LOOP . . .
```

Sledeći atributi su raspoloživi za signale nabrojivog tipa:

`d´VAL(pos)`	vraća vrednost sa navedene pozicije
`d´POS(vrednost)`	vraća poziciju navedene vrednosti
`d´LEFTOF(vrednost)`	vraća vrednost sa pozicije levo od navedene vrednosti
`d´VAL(vrsta, kolona)`	vraća vrednost sa navedene pozicije

Pr. 3-20 Atributi nabrojivog tipa

```
TYPE boja IS (crvena, zelena, plava, bela);
```

Za korisnički definisan nabrojivi tip *boja* važi:

```
boja'VAL(1) = zelena
boja'POS(zelena) = 1
boja'LEFTOF(plava) = zelena
```

***Atributi signala*.** Neka je *s* signal. Tada važi:

`s´EVENT`	vraća *true* ako se na signalu *s* desio događaj
`s´STABLE`	vraća *true* ako se na signalu *s* nije desio događaj
`s´ACTIVE`	vraća *true* ako je *s* = ´1´
`s´QUIET<vreme>`	vraća *true* ako se u navedenom vremenu na signalu *s* nije desio događaj
`s´LAST_EVENT`	vraća vreme proteklo od poslednjeg događaja na signalu *s*
`s´LAST_ACTIVE`	vraća proteklo vreme od kad je signal *s* poslednji put imao vrednost ´1´
`s´LAST_VALUE`	vraća vrednost signala *s* neposredno pre poslednjeg događaja

Pojam ″događaj na signalu″ odnosi se na promenu vrednosti signala. Većina navedenih atributa koristi se samo u simulaciji. Za sintezu, dozvoljena su samo prva dva, *´event* i *´stable*. Pri tom se atribut *´event* koristi mnogo češće.

Pr. 3-21 Detekcija ivice signala

Sledeće tri naredbe su ekvivalentne i mogu se koristiti u kôdu za sintezu. Svaka od naredbi vraća *true* ako se na signalu *clk* desio događaj, tj. ako je došlo do promene vrednosti signal *clk* i njegova nova vrednost je *clk*=´1´. Drugim rečima, svaka od sledeće tri naredbe se može koristiti za detekciju rastuće ivica signala *clk*.

```
IF (clk´EVENT AND clk=´1´) ...        -- atribut event u naredbi if
IF (NOT clk´STABLE AND clk=´1´) ...   -- atribut stable u naredbi if
WAIT UNTIL (clk´EVENT AND clk=´1´);   -- atribut event u naredbi wait
```

4. KONKURENTNE NAREDBE DODELE

Većina programskih jezika je zasnovana na sekvencijalnom izvršenju naredbi, što znači da se programske naredbe izvršavaju jedna za drugom, po redosledu kako je to programom definisano. Sekvencijalni programski jezici su pogodni za opis algoritama koji definišu neko izračunavanje, funkciju ili proceduru. Međutim, sekvencijalni model izračunavanja nije pogodan za verno modeliranje digitalnih sistema. Tipičan digitalni sistem se sastoji od više podsistema koji rade nezavisno jedan od drugog, tj. istovremeno ili paralelno. Iako se funkcija svakog podsistema može opisati sekvencijalnim programom, za modeliranje paralelizma na sistemskom nivou neophodna je posebna podrška u jeziku za opis hardvera. U VHDL-u takva podrška postoji u vidu konkurentnih naredbi, koje se slično podsistemima ili kolima izvršavaju (tj. rade) istovremeno.

U VHDL-u postoji više tipova naredbi i jezičkih konstrukcija namenjenih za pisanje konkurentnog kôda. Ova glava je posvećena konkurentnim naredbama dodele. Pomoću ovih naredbi signalima se pridružuju izrazi (u obliku konstanti, logičkih ili aritmetičkih izraza) koji definišu kako dati signal zavisi od drugih signala u kolu. Postoje tri vrste konkurentnih naredbi dodele:

- jednostavna naredba dodele
- naredba *when* i
- naredba *select.*

Kao što ćemo videti u narednim poglavljima, konkurentne naredbe dodele su jednostavne, ali u isto vreme i moćne jezičke konstrukcije. Uz to, VHDL kôd pisan konkurentnim naredbama dodele relativno se lako može vizuelno predstaviti u obliku konceptualnog hardverskog blok dijagrama. Postojanje jasne predstave o hardveru koji proističe iz kôda od suštinske je važnosti za razvoj efikasnog VHDL kôda za sintezu.

Za kreiranje složenijih modela u VHDL, pored konkurentnih naredbi dodele, koriste se još dve konkurentne naredbe: *generate* (služi za kreiranje ″petlji″ i ″grananja″ u konkurentnom kôdu) i naredba za instanciranje komponenti (koristi se za kreiranje hijerarhijski organizovanog kôda). Ove dve konkurentne naredbe biće razmatrane u glavama 7 i 8.

VHDL poseduje podršku kako za konkurentno, tako i za sekvencijalno programiranje. Međutim, u osnovi, VHDL je konkurentan jezik, a da bi se nametnulo sekvencijalno izvršenje naredbi koriste se posebne jezičke konstrukcije, kao što je to *process*. Bez obzira na to što se naredbe obuhvaćene procesom izvršavaju na sekvencijalan način, svaki proces,

kao celina, izvršava se konkurentno u odnosu na druge procese i druge konkurentne naredbe. Naredbi *process* i sekvencijalnim naredbama posvećena je glava 5.

Napomenimo da treba praviti jasnu razliku između konkurentnog/sekvencijalnog kôda s jedne i kombinacionih/sekvencijalnih digitalnih kola s druge strane. Kao što je poznato, digitalna kola se dele na kombinaciona i sekvencijalna. Kombinaciono kolo ne sadrži internu memoriju ili stanja, a njegovi izlazi zavise isključivo od trenutnih vrednosti ulaza. S druge strane, sekvencijalno kolo poseduju internu memoriju, a njegovi izlazi su u funkciji kako ulaza tako i tekućeg stanja (tj. trenutnog sadržaja memorije). Za opisivanje kombinacionih kola u VHDL-u može se koristiti bilo konkurentan bilo sekvencijalan kôd. S druge strane, po pravilu, sekvencijalna kola se opisuju sekvencijalnim kôdom.

4.1. Jednostavna konkurentna naredba dodele

Sintaksa jednostavne konkurentne naredbe dodele je oblika:

```
sig <= izraz AFTER kasnjenje;
```

Član *izraz* definiše novu vrednost signala *sig*, a *kasnjenje* trenutak kad će izračunata vrednost izraza biti dodeljena signalu. Razmotrimo naredbu:

```
y <= a + b + 1 AFTER 10 ns;
```

Značenje ove naredbe je sledeće. Uvek kada se promeni bilo *a* bilo *b*, izraz *a*+*b*+1 se iznova izračunava, a rezultat se dodeljuje signalu *y* nakon (*after*) 10 ns.

Vreme navedeno kao *kasnjenje* definiše interno, tj. propagaciono kašnjenje, odnosno vreme koje bi bilo potrebno da se u realnom kolu, koje realizuje dati izraz, nakon promene ulaza, generiše nova izlazna vrednost. Međutim, pošto propagaciono kašnjenje zavisi od komponenti koje se koriste za realizaciju kola, implementacione tehnologije, prostornog rasporeda komponenti, rutiranja, procesa fabrikacije i brojnih drugih faktora, nije moguće automatski sintetizovati kolo sa tačnim, unapred zadatim iznosom kašnjenja. Iz tog razloga, informacija o kašnjenju (tj. tajmingu) se nikada eksplicitno ne navodi u VHDL kôdu za sintezu. Umesto toga, koristi se oblik konkurentne naredbe dodele bez navedenog propagacionog kašnjenja:

```
sig <= izraz;
```

Gornja naredba dodele zapravo je skraćeni oblik sledeće naredbe:

```
sig <= izraz AFTER 0 ns;
```

Ova naredba modelira idealizovano digitalno kolo, koje je u stanju da nakon promene ulaza trenutno postavi novu vrednost na izlazu. U realnosti, takva digitalna kola ne postoje, jer je propagaciono kašnjenje, makar i u minimalnom iznosu, uvek prisutno. Međutim, ova nedoslednost u modeliranju ne predstavlja problem u VHDL kôdu za sintezu, koji je prevashodno fokusiran na opis funkcionalnosti kola. Ako je to potrebno, informacija o tajmingu se može dobiti nakon obavljene sinteze, vremenskom analizom sintetizovane strukture.

Deo *izraz* naredbe konkurentne dodele sadrži signale i konstante povezane operatorima, kao što su: *and*, *not*, +, *, *sll* i slični. Izraz može biti konstantna vrednost, logička operacija, aritmetička operacija itd. Sledi nekoliko primera jednostavnih konkurentnih naredbi dodele:

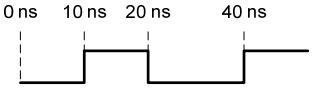
```
enable <= '1';
sel <= (r1 AND r2) OR (r3 AND r4);
sum <= a + b + c - 1;
```

Pr. 4-1 Generisanje talasnog oblika

U svom najopštijem obliku, jednostavna konkurentna naredba dodele može da sadrži proizvoljan broj sekcija oblika *izraz after kašnjenje* razdvojenih zarezima, npr.

```
sig <= ´0´, ´1´ after 10 ns, ´0´ after 10 ns, ´1´ after 20 ns;
```

Prilikom inicijalnog izvršenja gornje naredbe, signal *sig* dobija vrednost ´0´; 10 ns nakon toga, vrednost signala *sig* postaje ´1´; posle narednih 10 ns, *sig* ponovo ima vrednost ´0´; nakon narednih 20 ns, *sig* dobija svoju konačnu vrednost, ´1´. Talasni oblik rezultujućeg signala je prikazan na Sl. 4-1.



Sl. 4-1 Talasni oblik signala generisan konkurentnom naredbom dodele iz Pr. 4-1.

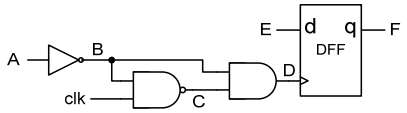
Jednostavna konkurentna naredba dodele u ovakvom obliku se, naravno, ne može sintetizovati, ali se zato često može videti u testbenču, gde se koristi za generisanje pobudnih signala složenog talasnog oblika.

Pr. 4-2 δ-kašnjenje

Propagaciono kašnjenje od 0 ns se na poseban način tretira u simulaciji, kao tzv. *delta*-kašnjenje (odnosno δ-kašnjenje). δ-kašnjenje je veće od nule, ali manje od bilo kog konačnog fizičkog kašnjenja, a uvedeno je radi pravilnog uređenja događaja koji se dešavaju u kolu tokom simulacije. VHDL simulacija je vođena događajima koji se vezuju za diskretne vremenske trenutke. Pod događajem se podrazumeva promena vrednosti bilo kog signala u kolu. Događaj koji se desio u trenutku t_0 inicira nove događaje, koji se vezuju za odgovarajuće buduće vremenske trenutke (određene propagacionim kašnjenjima koja su pridružena signalima). Kad obradi sve događaje vezane za jedan vremenski trenutak, simulator prelazi na prvi sledeći trenutak u koji su ranije raspoređeni događaji i tako redom, sve dok postoje neobrađeni događaji. Ovaj mehanizam dobro funkcioniše sve dok je propagaciono kašnjenje veće od nule. Međutim, ako je propagaciono kašnjenje jednako nuli, novi događaj bi morao da se desi u istom trenutku kad i događaj koji ga je izazvao. U nekim slučajevima to može da dovede do nekonzistentnih rezultata simulacije, odnosno do pojave da se rezultati iz ponovljenih simulacija istog kola razlikuju.

Razmotrimo tok simulacije kola sa Sl. 4-2 pod pretpostavkom da sva logička kola imaju propagaciono kašnjenje od 0 ns i da se **ne** koristi mehanizam δ-kašnjenja. Pretpostavimo da se u trenutku t = 10 ns desila promena signal A sa ´1´ na ´0´. Ova promena se trenutno prenosi kroz invertor, tako da u istom tom trenutku (t = 10 ns) i signal B dobija vrednost ´1´. To je novi događaj, čija pojava zahteva evaluaciju izlaza oba logička kola koje signal B pobuđuje. Moguća su dva scenarija. U prvom, simulator najpre određuje odziv NI kola, a zatim odziv I kola. Pošto je nova vrednost signala C = ´0´, signal D zadržava vrednost ´0´ i pored toga što je B=´1´. Međutim, ako simulator izabere da prvo obradi I kolo, signal D će najpre dobiti vrednost ´1´ (zbog C koji je još uvek ´1´), ali će se neposredno nakon toga, tj. nakon što simulator evaluira izlaz NI kola, vratiti na D=´0´. Iako je u oba scenarija konačna vrednost signala D ista, u drugom slučaju na signalu D se javlja trenutna promena sa ´0´ na ´1´, a onda ponovo na ´0´. Promena sa ´0´ na ´1´ predstavlja rastuću ivicu, koja će inicirati okidanje i upis vrednosti signala E u flip-flop. U prvom scenariju, upis u flip-flop se neće

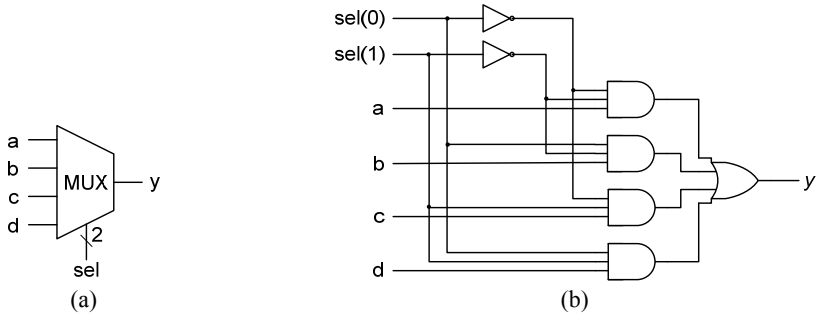
desiti (zato što se signal *D* nije menjao). Redosled obrade događaja tokom simulacije zavisi od toga kako su uređene strukture podataka u kompajliranom VHDL kôdu. Na primer, može se desiti da nakon ponovljene kompilacije istog VHDL kôda, simulator umesto I prvo evaluira NI kolo, ili obrnuto.



Sl. 4-2. Kolo za ilustraciju δ-kašnjenja.

Mehanizam δ-kašnjenja je uveden iz razloga da bi se izbegle opisane nedoslednosti u simulaciji. Konkretno, svaki diskretni vremenski trenutak se deli na fiktivne vremenske intervale beskonačno malog trajanja (tzv. δ-intervale), a svi događaji koji se iniciraju s kašnjenjem od 0 ns raspoređuju se u naredni δ-interval. Primenjeno na primer simulacije kola sa Sl. 4-2, to znači sledeće: Promena signala *A* na ´0´ dešava se u prvom δ-intervalu vremenskog trenutka t=10 ns. Obrada ovog događaja dovodi do promene signala *B* sa ´0´ na ´1´. Međutim, ova promena se ne prenosi odmah na signal *B*, već se fiktivno kasni za jedno δ, tj. raspoređuje u sledeći δ-interval. Pošto u prvom δ-intervalu nema drugih događaja, simulator prelazi na 2. δ-interval. Promenjena vrednost signala *B*, koja je sada vidljiva, dovodi do evaluacije oba logička kola, I i NI, ali se postavljanje signala *C* i *D* na njihove nove vrednosti odlaže za naredni, tj. 3. δ-interval. U 3. δ-intervalu, signal *D* dobija vrednost ´1´, a signal *C* vrednost ´0´. Promenjena vrednost signala *C* inicira ponovnu evaluaciju izlaza I kola, čija se nova i konačna izlazna vrednost, *D*=´0´, za vremenski trenutak t=10 ns postavlja u 4. δ-intervalu. Dakle, u trenutku t=10 ns na signalu *D* se javlja impuls trajanja jednog δ-intervala, a ovakav ishod ne zavisi od redosleda evaluacije logičkih kola.

Pr. 4-3 Multiplekser 4-u-1 - realizacija pomoću jednostavne konkurentne naredbe dodele



Sl. 4-3 Multiplekser 4-u-1: (a) grafički simbol; (b) kombinaciona mreža.

Na Sl. 4-3(a) je prikazan grafički simbol multipleksera 4-u-1. Vrednost dvobitnog selekcionog ulaza *sel* bira jedan od ulaza, *a, b, c* ili *d*, čija se vrednost prenosi na izlaz *y*. Na Sl. 4-3(b) je prikazana realizacija multipleksera pomoću logičkih kola. Funkcija multipleksera 4-u-1 se može opisati u VHDL-u uz pomoć samo jedne jednostavne konkurentne naredbe dodele koja izlaznom signalu *y* dodeljuje vrednost logičkog izraza izvedenog na osnovu kombinacione mreže sa Sl. 4-3(b):

```
1  ----------------------------------------------------------------------
2  LIBRARY IEEE;
3  USE IEEE.STD_LOGIC_1164.ALL;
4  ----------------------------------------------------------------------
5  ENTITY mux IS
6  PORT ( a,b,c,d : IN STD_LOGIC;
7         sel : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
8         y : OUT STD_LOGIC);
9  END mux;
10 ARCHITECTURE arch OF mux IS
11 BEGIN
12    y <= (a AND NOT sel(1) AND NOT sel(0)) OR
13          (b AND NOT sel(1) AND sel(0)) OR
14          (c AND sel(1) AND NOT sel(0)) OR
15          (d AND sel(1) AND sel(0));
16 END arch;
```

Pr. 4-4 Jednostavna konkurentna naredba dodele sa zatvorenom petljom

VHDL ne zabranjuje da se isti signal nađe s obe strane naredbe dodele. Međutim, kad se u istoj naredbi izlazni signal koristi i kako ulaz, formira se zatvorena petlja. U mnogim slučajevima to nije poželjno, jer se tako mogu kreirati interna stanja ili čak izazvati oscilacije u sintetizovanom kolu. Primera radi, razmotrimo sledeću VHDL naredbu:

```
q <= (q AND NOT en) OR (d AND en);
```

Kao što vidimo, *q* je u isto vreme i ulazni i izlazni signal naredbe dodele. Ako važi *en* = ´1´, izlazu *q* se dodeljuje vrednost signala *d*, zato što se izraz svodi na *q* <= *d*. U suprotnom slučaju, ako važi *en* = ´0´, *q* zadržava svoju prethodnu vrednost, jer izraz postaje *q* <= *q*. Uočimo da izlaz *q* ne zavisi samo od drugih signala, tj. *d* i *en*, već i od svoje sopstvene vrednosti. Drugim rečima, da li će pri *en*=´0´ *q* imati vrednost ´0´ ili ´1´ zavisi od toga koju je vrednost ovaj signal imao u momentu promene signala *en* sa ´1´ na ´0´. To znači da kolo više nije kombinaciono, već sekvencijalno. Ako prethodni izraz modifikujemo tako što ćemo s desne strane umesto *q* napisati *not q*:

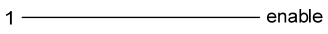
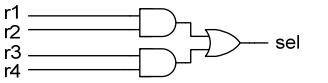
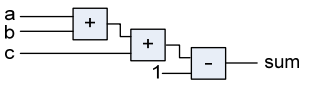
```
q <= (NOT q AND NOT en) OR (d AND en);
```

izlaz *q* će oscilovati između ´0´ i ´1´ za vreme dok je *en*=´0´, zato što se sada izraz svodi na *q* <= *not q*. Frekvencija oscilovanja u kolu sintetizovanom na osnovu ovakvog izraza zavisiće od propagacionog kašnjenja kroz invertor koji će biti iskorišćen za realizaciju operacije *not q*.

Konkurentne naredbe dodele sa zatvorenom petljom su osetljive na interno propagaciono kašnjenje i podložne su oscilacijama. Ovakve konstrukcije mogu da ″zbune″ softver za sintezu i treba ih izbegavati u VHDL kôdu (tačnije, ne treba ih nikada koristiti). Za modeliranje sekvencijalnog ponašanja treba koristiti sekvencijalne naredbe.

Konceptualna implementacija. Kreiranje konceptualnog blok dijagrama na osnovu jednostavne konkurentne naredbe dodele obično je lak zadatak. Celokupna naredba se posmatra kao jedno digitalno kolo. Izlaz kola je signal s leve strane, dok su ulazi kola svi signali iz izraza s desne strane znaka dodele. Svaki operator sadržan u izrazu se preslikava na jedan manji blok i povezuje sa ulazima i izlazima na način kako to diktira izraz. Na Sl. 4-4 su prikazani konceptualni dijagrami tri jednostavne konkurentne naredbe dodele. Treba

naglasiti da ovi dijagrami predstavljaju samo konceptualni prikaz digitalnog sistema i da ne moraju verno da odražavaju strukturu sintetizovanog hardvera. Tokom sinteze, izraz može biti automatski transformisan (npr. radi pojednostavljenja); pojedini operatori, kao ″+″ i ″-″, mogu biti zamenjeni složenijim hardverskim strukturama, dok se neki operatori uopšte i ne mogu automatski sintetizovati (npr. operator deljenja).

Naredba	Konceptualna implementacija
`enable <= ´1´;`	
`sel <= (r1 AND r2) OR (r3 AND r4);`	
`sum <= a + b + c – 1;`	

Sl. 4-4 Konceptualni dijagrmi tri jednostavne konkurentne naredbe dodele.

4.2. WHEN

Naredba *when*, ili konkurentna naredba uslovne dodele zapravo predstavlja uopštenje jednostavne konkurentne naredbe dodele. Za razliku od jednostavne naredba dodele, kod koje postoji samo jedan izraz s desne strane znaka ″<=″, naredba uslovne dodele omogućava da se s desne strane znaka ″<=″ nađe više od jednog izraza. Svakom takvom izrazu pridružen je *uslov*, a jedinstveni signal s leve strane znaka ″<=″ dobija vrednost onog izraza čiji je uslov trenutno ispunjen, tj. tačan. Sintaksa naredbe *when* je sledećeg oblika:

```
sig <= izraz_1 WHEN uslov_1 ELSE
       izraz_2 WHEN uslov_2 ELSE
       ...
       izraz_n-1 WHEN uslov_n-1 ELSE
       izraz_n;
```

Članovi *uslov_1* do *uslov_n-1* su logički iskazi od kojih svaki može biti tačan (*true*) ili netačan (*false*). Naredba *when* se izvršava tako što se uslovi ispituju **redom**, počev od uslova *uslov_1*, pa sve dok se ne naiđe na granu s tačnim uslovom, a onda se izračunava vrednost odgovarajućeg izraza i dodeljuje signalu s leve strane znaka ″<=″. Ako ni jedan uslov nije tačan, za izvršenje se bira izraz iz poslednje grane (*izraz_n*).

Pr. 4-5 Kako radi naredba WHEN?

Razmotrimo sledeću *when* naredbu:

```
outp <= "000" WHEN (inp='0' OR reset='1') ELSE
        "101" WHEN ctl='1' ELSE
        "010";
```

Najviši prioritet u *when* naredbi ima uslov iz prve grane, zatim uslov iz druge i tako redom. U konkretnom primeru to znači da izlazni signal *outp* dobija vrednost ″000″ ako važi *inp*=´0´ ili *reset*=´1´, bez obzira na vrednost signala *ctl*. Tek ako uslov iz prve grane nije zadovoljen (tj. važi *inp*=´1´ i *reset*=´0´), ispituje se da li je *ctl*=´1´ i ako jeste, izlaznom signalu se dodeljuje vrednost ″101″. Konačno, ako ni jedan uslov nije zadovoljen, signal *outp* dobija vrednost iz poslednje grane, tj. ″010″.

Pr. 4-6 Multiplekser 4-u-1 - realizacija pomoću naredbe *when*

Sledi VHDL opis multipleksera 4-u-1 zasnovan na naredbi *when*:

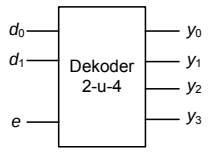
```
1  ----------------------------------------------------------------------
2  LIBRARY IEEE;
3  USE IEEE.STD_LOGIC_1164.ALL;
4  ----------------------------------------------------------------------
5  ENTITY mux4u1 IS
6     PORT ( a,b,c,d : IN STD_LOGIC;
7            sel: IN STD_LOGIC_VECTOR(1 DOWNTO 0);
8            y : OUT STD_LOGIC);
9  END mux4u1;
10 ----------------------------------------------------------------------
11 ARCHITECTURE when_arch OF mux4u1 IS
12 BEGIN
13     y <= a WHEN sel = "00" ELSE
14          b WHEN sel = "01" ELSE
15          c WHEN sel = "10" ELSE
16          d;
17 END when_arch;
18 ----------------------------------------------------------------------
```

Arhitektura sadrži samo jednu *when* naredbu. U ovoj naredbi, prvo se ispituje uslov *sel*=″00″ i ako je tačan, izlaznom signalu *y* se dodeljuje vrednost ulaznog signala *a*. Inače, ako vrednost selekcionog signala *sel* nije jednaka ″00″, ispituje se sledeći uslov, *sel*=″01″. Ako je ovaj uslov tačan, *y* dobija vrednost ulaza *b*. U suprotnom, prelazi se na ispitivanje uslova *sel*=″10″. Ako ni jedan od tri uslova nije tačan, izlazu *y* se dodeljuje vrednost signala *d*.

U prethodnom VHDL opisu, selekcioni signal *sel* je tipa *std_logic_vector*. Podsetimo da tip podataka *std_logic* definiše 8-nivovski logički sistem, koji osim logičke´0´ i ´1´ sadrži još 6 različitih nivoa signala (v. 3.3.3). To znači da postoje 8x8=64 različite vrednosti dvobitnog signala ovog tipa, što osim očekivanih binarnih ″00″, ″01″, ″10″ i ″11″ uključuje i nebinarne vrednosti poput: ″0-″, ″LH″, ″ZW″ itd. Shodno tome, poslednja grana *when* naredbe iz opisa multipleksera ne pokriva samo kombinaciju ″11″, kao što bi se to na prvi pogled moglo zaključiti, već i svih 60 nebinarnih vrednosti. Međutim, nebinarne vrednosti imaju smisla samo u simulaciji. U hardveru nije moguće realizovati poređenja kao što su *sel=″0Z″* ili *sel=″UX″* . Iz tog razloga, softveri za sintezu ignorišu nebinarne vrednosti signala tipa *std_logic* (izuzev donekle vrednosti ´Z´), a rezultat sinteze VHDL kôda, poput opisa multipleksera iz ovog primera, odgovaraće očekivanom.

Pr. 4-7 Binarni dekoder 2-u-4 - realizacija pomoću naredbe *when*

Na Sl. 4-5 su prikazani grafički simbol i tabela istinitosti binarnog dekodera 2-u-4. Kolo ima 2 ulaza za binarno kodirani podatak, d_0 i d_1, i 4 izlaza, y_i, i=0, …, 3, pri čemu svaka kombinacija binarnih vrednosti na ulazu pobuđuje tačno jedan izlaz. Dodatni ulaz za dozvolu rada, *e*, upravlja izlazom dekodera na sledeći način: ako važi *e*=´0´, tada ni jedan izlaz dekodera nije aktivan; ako važi *e*=´1´, aktivan je (tj. ima vrednost ´1´) samo izlaz y_i, gde je *i* ceo broj jednak binarnoj vrednosti (d_1, d_0).



(a)

e	d_1	d_0	y_3	y_2	y_1	y_0
1	0	0	0	0	0	1
1	0	1	0	0	1	0
1	1	0	0	1	0	0
1	1	1	1	0	0	0
0	X	X	0	0	0	0

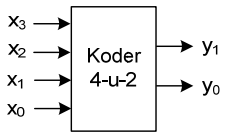
(b)

Sl. 4-5 Binarni dekoder 2-u-4: (a) grafički simbol; (b) tabela istinitosti.

Sledi VHDL opis dekodera 2-u-4. U *when* naredbi, grane su poređane po prioritetima, s granom najvišeg prioriteta na početku. To znači da se pri *e*=´0´ izlaz *y* postavlja na vrednost ″0000″, bez obzira na vrednost ulaza *d* (linija 13). Tek ako važi *e*=´1´, dekoderu je dozvoljen rad, a njegov izlaz će biti postavljen shodno vrednosti ulaza *d* (linije 14-17).

```
1  ----------------------------------------------------------------
2  LIBRARY IEEE;
3  USE IEEE.STD_LOGIC_1164.ALL;
4  ----------------------------------------------------------------
5  ENTITY dek2u4 IS
6    PORT (d : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
7          e : IN STD_LOGIC;
8          y : OUT STD_LOGIC_VECTOR(3 DOWNTO 0));
9  END dek2u4;
10 ----------------------------------------------------------------
11 ARCHITECTURE when_arch OF dek2u4 IS
12 BEGIN
13  y <= "0000" WHEN e = ´0´ ELSE
14       "0001" WHEN d = "00" ELSE
15       "0010" WHEN d = "01" ELSE
16       "0100" WHEN d = "10" ELSE
17       "1000";
18 END when_arch;
```

Pr. 4-8 Binarni koder 4-u-2 - realizacija pomoću naredbe *when*



(a)

x_3	x_2	x_1	x_0	y_1	y_0
0	0	0	1	0	0
0	0	1	0	0	1
0	1	0	0	1	0
1	0	0	0	1	1
sve ostale komb.				-	-

(b)

Sl. 4-6 Binarni koder 4-u-2: (a) grafički simbol; (b) tabela istinitosti.

Na Sl. 4-6(a) je prikazan grafički simbol kodera 4-u-2, a na Sl. 4-6(b) tabela istinitosti ovog kola. S ograničenjem da u svakom trenutku tačno jedan od 4 ulaza mora imati vrednost ´1´, na izlazu binarnog kodera se generiše 2-bitni binarni broj koji ukazuje na indeks ulaza čija je vrednost jednaka ´1´. Za neregularnu pobudu (ulazna kombinacija ″sve nule″ i sve kombinacije sa više od jedne jedinice) odziv kodera nije definisan (što je naznačeno sa ″- -″ u poslednjoj vrsti tabele istinitosti). Nedefinisan odziv znači da pod datim uslovima izlazi kodera mogu biti postavljeni na bilo koju vrednost, ´0´ ili ´1´. Ova proizvoljnost je dopuštena zato što se takva neregularna pobuda ulaza nikad neće desiti u regularnim

uslovima rada. Naravno, proizvoljna vrednost, ″-″, fizički ne postoji, ali se ova informacija može iskoristiti za optimizaciju hardvera u toku logičke sinteze.

U nastavku je prezentovan VHDL opis binarnog kodera 4-u-2 koji koristi naredbu *when*.

```
1  ----------------------------------------------------------------
2  LIBRARY IEEE;
3  USE IEEE.STD_LOGIC_1164.ALL;
4  ----------------------------------------------------------------
5  ENTITY encoder IS
6    PORT (x : IN STD_LOGIC_VECTOR(3 DOWNTO 0);
7          y : OUT STD_LOGIC_VECTOR(1 DOWNTO 0));
8  END encoder;
9  ----------------------------------------------------------------
10 ARCHITECTURE when_arch OF encoder IS
11 BEGIN
12    y <=  "00" WHEN x="0001" ELSE
13          "01" WHEN x="0010" ELSE
14          "10" WHEN x="0100" ELSE
15          "11";
16 END when_arch;
17 ----------------------------------------------------------------
```

Uočimo da u datom VHDL opisu neregularne ulazne kombinacije nisu tretirane izdvojeno, već su sve one implicitno pripojene jednoj regularnoj ulaznoj kombinaciji (tj. kombinaciji ″1000″). To nije pogrešno, ali se gubi mogućnost za logičku minimizaciju, jer se od hardvera zahteva da za svaku neregularnu ulaznu kombinaciju na izlazu uvek postavi istu vrednost, ″11″. Međutim, u VHDL-u je moguće postaviti signale tipa *std_logic* na proizvoljnu vrednost, tj. ″-″, kao u varijanti *when* naredbe koja sledi. Sada je opis u potpunosti usklađen sa specifikacijom kodera (v. Sl. 4-6(b)), a softveru za sintezu se pruža mogućnost da proizvoljnost u opisu iskoristi za optimizaciju (minimizaciju) hardvera. Nažalost, korišćenje proizvoljnih vrednosti, iako podržano u jeziku VHDL, nije univerzalno podržano do strane svih softvera za sintezu.

```
y <=  "00" WHEN x="0001" ELSE
      "01" WHEN x="0010" ELSE
      "10" WHEN x="0100" ELSE
      "11" WHEN x="1000" ELSE
      "--";
```

Treba napomenuti da s obzirom na to što svaka grana *when* naredbe definiše izlaznu vrednost za jednu konkretnu ulaznu binarnu kombinaciju, ovakav način opisivanja kodera je nepraktičan ako je broj ulaza veći od 4. Kompaktniji opis kodera s većim brojem ulaza može se kreirati korišćenjem naredbe *generate* (v. 8.4) ili naredbe *loop* (v. 8.5).

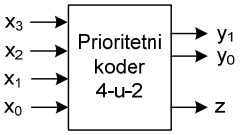
Pr. 4-9 Prioritetni koder 4-u-2 - realizacija pomoću naredbe *when*

Kod binarnog kodera opisanog u prethodnom primeru postoji ograničenje da u bilo kom trenutku najviše jedan ulaz može biti aktivan. Ako se ovo ograničenje ne poštuje, tj. u slučajevima kad je aktivno više od jednog ulaza, izlaz kodera biće pogrešan. Kod prioritetnog kodera, ulazi su uređeni po prioritetima, a izlaz kodera, interpretiran kao binarni broj, ukazuje na indeks aktivnog ulaza najvišeg prioriteta. Drugim rečima, za sve vreme dok je aktivan ulaz visokog prioriteta, svi ulazi nižeg prioriteta se ignorišu. Na Sl. 4-7(a) je prikazana tabela istinitosti, a na Sl. 4-7(b) grafički simbol prioritetnog kodera 4-u-

2. Pretpostavka je da ulaz x_3 ima najviši, a x_0 najniži prioritet. Dvobitni izlaz y predstavlja binarni broj koji ukazuje na ulaz najvišeg prioriteta koji ima vrednost ´1´. Pošto je moguće da ni jedan ulaz nije aktivan, predviđen je još jedan dodatni izlaz, z koji ukazuje na ovu situaciju. Naime, ako je barem jedan ulaz jednak ´1´, tada je z=´1´; inače, ako su svi ulazi jednaki ´0´, tada važi z=´0´.

x_3	x_2	x_1	x_0	y_1	y_0	z
0	0	0	0	0	0	0
0	0	0	1	0	0	1
0	0	1	-	0	1	1
0	1	-	-	1	0	1
1	-	-	-	1	1	1

(a)



(b)

Sl. 4-7 Prioritetni koder 4-u-2: (a) tabela istinitosti; (b) blok dijagram.

Sledi VHDL opis prioritetnog kodera 4-u-2 u kome je iskorišćena naredba *when*.

```
1  ----------------------------------------------------------------
2  LIBRARY IEEE;
3  USE IEEE.STD_LOGIC_1164.ALL;
4  ----------------------------------------------------------------
5  ENTITY pencoder IS
6    PORT (x : IN STD_LOGIC_VECTOR(3 DOWNTO 0);
7          y : OUT STD_LOGIC_VECTOR(1 DOWNTO 0);
8          z : OUT STD_LOGIC);
9  END pencoder;
10 ----------------------------------------------------------------
11 ARCHITECTURE when_arch OF pencoder IS
12 BEGIN
13    y <= "11" WHEN x(3) = '1' ELSE
14         "10" WHEN x(2) = '1' ELSE
15         "01" WHEN x(1) = '1' ELSE
16         "00";
17    z <= '0' WHEN x = "0000" ELSE '1';
18 END when_arch;
19 ----------------------------------------------------------------
```

Kôd sadrži dve *when* naredbe, gde prva (linije 13-16) postavlja izlaz y, a druga (linija 17) izlaz z. Prva naredba *when* nalaže da se signalu y dodeli vrednost ″11″ ako je ulaz x(3)= ´1´ (linija 13). Ako je ovaj uslov tačan, tada preostale *when* grane ne utiču na vrednost izlaznog signala y. Druga *when* grana (linija 14) nalaže da se signalu y dodeli vrednost ″10″ ako važi x(2)=´1´. Ovo može da se desi samo ako je x(3)= ´0´. Svaka sledeća *when* grana može da utiče na y samo ako ni jedan od uslova iz prethodnih *when* grana nije zadovoljen. Drugim rečima, najviši prioritet ima ulaz x(3), zatim x(2), x(1) i konačno x(0) koji je najnižeg prioriteta.

Konceptualna implementacija. Ključ za realizaciju naredbe *when* u hardveru leži u obezbeđivanju željenog redosleda ispitivanja uslova. Za razliku od tradicionalnih programskih jezika, kod kojih bi redosled ispitivanja uslova bio obezbeđen sekvencijalnim načinom izvršenja programa, u sintezi je, za postizanje istog efekta, neophodan dodatni hardver. Na Sl. 4-8(a) je prikazana konceptualna realizacija naredbe *when* sa jednim uslovom:

```
sig <= izraz_1 WHEN uslov_1 ELSE
       izraz_2;
```

Multiplekser 2-u-1, u zavisnosti od ishoda ispitivanja uslova *uslov_1*, postavlja signal *sig* na vrednost izraza *izraz_1* ili izraza *izraz_2*. Multiplekser sa Sl. 4-8(a) je neka vrsta *apstraktnog multipleksera* sa selekcionim signalom tipa *boolean*. Kada je selekcioni signal tačan (***T****rue*) na izlaz se prenosi vrednost sa ulaznog porta T, a kada je netačan (***F****alse*) vrednost sa ulaznog porta F.

Sl. 4-8 Konceptualna realizacija naredbe *when* sa: (a) jednim uslovom; (b) tri uslova.

Na Sl. 4-8(b) je prikazana konceptualna realizacija naredbe *when* sa tri uslova:

```
sig <= izraz_1 WHEN uslov_1 ELSE
       izraz_2 WHEN uslov_2 ELSE
       izraz_3 WHEN uslov_3 ELSE
       izraz_4;
```

Lako je učiti da svaka sledeća grana naredbe *when* zahteva uvođenje još jednog apstraktnog multipleksera čiji se izlaz povezuje na *F* port multipleksera za prethodnu granu. Na ovaj način, formira se prioritetna mreža multipleksera koja najviši prioritet daje uslovu iz prve grane (*uslov_1*). Ako je uslov *uslov_1* tačan, tada signal *sig* dobija vrednost izraza *izraz_1*. U suprotnom, ako je *uslov_1* netačan, *uslov_2* dobija šansu da na izlaz postavi ´svoj´ izraz (*izraz_2*) ili da odlučivanje prepusti sledećem uslovu.

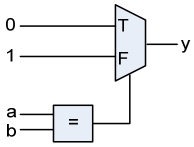
Pr. 4-10 Konceptualna implementacija naredbe *when* - prvi primer

Razmotrimo sledeći segment VHDL kôda:

```
SIGNAL a,b,y : STD_LOGIC;
. . .
y <= ´0´ WHEN a=b ELSE
     ´1´;
```

Naredba *when* sadrži samo jedan uslov, koji je tačan ako su vrednosti jednobitnih signala *a* i *b* jednake, a u suprotnom netačan. Na Sl. 4-9(a) je prikazan odgovarajući konceptualni dijagram. Iako signali tipa *std_logic* mogu imati 8 različitih vrednosti, prilikom sinteze u obzir se uzimaju samo dve, ´0´ i ´1´, pošto za preostalih 6 (izuzimajući ´Z´) ne postoji adekvatna fizička interpretacija. Takođe, vrednosti tipa *boolean* (*true* i *false*), iako nemaju fizički smisao, mogu se interpretirati kao logičko 1 (*true*) i logička 0 (*false*). Imajući to u vidu, funkcija *a=b* se može predstaviti u obliku tabele istinitosti sa Sl. 4-9(b). Ova funkcija se može izraziti u vidu logičke jednačine $\overline{a}\overline{b} + ab = \overline{a \oplus b}$, što odgovara funkciji logičkog kola koincidencije (*xnor* – isključivo NILI). Na Sl. 4-9(c) je prikazana kombinaciona mreža

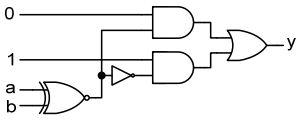
koja predstavlja konačni rezultat sinteze naredbe *when* iz ovog primera. Apstraktni blok jednakosti iz konceptualnog dijagrama zamenjen je logičkim kolom koincidencije, a apstraktni multiplekser dvonivovskom logičkom mrežom koja realizuje funkciju multipleksera 2-u-1.



(a)

a b	a=b
0 0	1
0 1	0
1 0	0
1 1	1

(b)



(c)

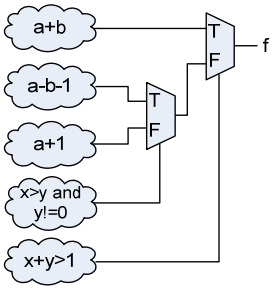
Sl. 4-9 Sinteza VHDL kôda iz Pr. 4-10: (a) konceptualni dijagram; (b) tabela istinitosti funkcije jednakosti; (c) sintetizovana kombinaciona mreža.

Pr. 4-11 Konceptualna implementacija naredbe *when* - drugi primer

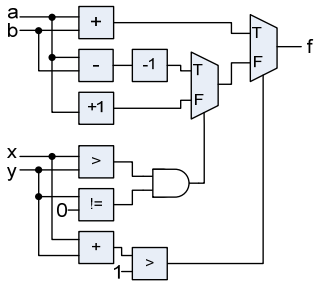
Razmotrimo hardversku realizaciju *when* naredbe iz sledećeg segmenta VHDL kôda:

```
SIGNAL a,b,f : UNSIGNED(7 DOWNTO 0);
SIGNAL x,y : UNSIGNED(3 DOWNTO 0);
. . .
f <= a+b WHEN x+y>1 ELSE
     a-b-1 WHEN x>y AND y!=0 ELSE
     a+1;
```

Na Sl. 4-10(a) je prikazan inicijalni konceptualni dijagram. U sledećem koraku, ″oblaci″ iz polaznog dijagrama se zamenjuju apstraktnim blokovima koji realizuju uslove i izraze iz posmatrane *when* naredbe (Sl. 4-10(b)). Razrada (sinteza) kola se može nastaviti i dalje, zamenom apstraktnih blokova odgovarajućim kombinacionim mrežama: blok označen sa ″+″ zamenjuje se binarnim sabiračem, blok ″>″ komparatorom itd. Tako se konačno stiže do mreže koja sadrži samo logička kola.



(a)



(b)

Sl. 4-10 Sinteza VHDL kôda iz Pr. 4-11: (a) inicijalni konceptualni dijagram; (b) detaljniji konceptualni dijagram.

Primeri Pr. 4-10 i Pr. 4-11 ukazuju na tok sinteze VHDL kôda. Međutim, treba naglasiti da se sinteza ne svodi samo na konstrukciju konceptualnog dijagrama i zamenu apstraktnih blokova odgovarajućim hardverskim realizacijama, već uključuje i različite oblike optimizacija koje se sprovode s krajnjim ciljem da se, u meri u kojoj je to moguće, smanji složenost rezultujućeg hardvera.

4.3. SELECT

Naredba *select*, ili naredba dodele sa izborom vrednosti, ima sledeću sintaksu:

```
WITH selekcioni_izraz SELECT
   sig <= izraz_1 WHEN vrednost_1,
          izraz_2 WHEN vrednost_2,
          ...
          izraz_n WHEN vrednost_n;
```

Kao i naredba *when* i naredba *select* sadrži više alternativnih izraza čije se vrednosti, pod određenim uslovima, dodeljuju signalu s leve strane znaka dodele. Razlika u odnosu na naredbu *when* je u načinu kako se vrši izbor izraza čija će vrednost biti dodeljena izlaznom signalu. Kod naredbe *select* najpre se određuje vrednost selekcionog izraza iz zaglavlja naredbe, zatim se dobijeni rezultat poredi (istovremeno) sa vrednostima iz svih *when* grana (*vrednost_1, ..., vrednost_n*) i, konačno, izlaznom signalu se dodeljuje rezultat izračunavanja izraz iz grane u kojoj se javilo slaganje. Drugim rečima, izračunata vrednost selekcionog izraza se koristi kao ključ za izbor jednog od više alternativnih izraza. Pri tom se zahteva da: a) svaka vrednost bude navedena tačno jedanput i b) svaki mogući rezultat selekcionog izraza bude pokriven tačno jednom vrednošću! Takođe, u poslednjoj grani dozvoljeno je koristiti službenu reč *others* da bi se označio izraz (*izraz_n*) koji će biti izabran ako se rezultat selekcionog izraza ne poklapa ni sa jednom eksplicitno navedenom vrednošću:

```
WITH selekcioni_izraz SELECT
     sig <= izraz_1 WHEN vrednost_1,
            izraz_2 WHEN vrednost_2,
            ...
            izraz_n WHEN OTHERS;
```

″*When* vrednost″ iz naredbe *select* može biti navedena na tri različita načina i to kao: (a) pojedinačna vrednosti, (b) opseg vrednosti i (c) skup vrednosti:

```
WHEN v1               -- pojedinačna vrednost
WHEN v1 TO v2         -- opseg vrednosti od v1 do v2
WHEN v1 | v2 | v3 ... - skup vrednosti {v1, v2, v3,...}
```

Razmotrimo sledeću *select* naredbu:

```
WITH control SELECT
   outp <= "000" WHEN "00",
           "111" WHEN "01" | "10",
           "010" WHEN OTHERS;
```

U ovom primeru, ulogu selekcionog izraza igra 2-bitni signal *control*. Izlazni signal *outp* dobija vrednost ″000″ ako važi *control*=″00″. Ako je vrednost signala *control* jednaka ″01″ ili ″10″, *outp* dobija vrednost ″111″. Signalu *outp* se dodeljuje vrednost ″010″ ako vrednost signala *control* nije jednaka ni jednoj od vrednosti navedenih u prve dve grane. Redosled grana u *select* naredbi nije od značaja. To je zato što se uslovi ne ispituju redom (kao kod naredbe *when*) već svi u isto vreme (paralelno).

Pr. 4-12 Opis tabele istinitosti

Kao što je poznato, bilo koja logička funkcija se može predstaviti u vidu tabele istinitosti. Prosto, za svaku kombinaciju vrednosti ulaznih promenljivih, u tabeli istinitosti je

navedena odgovarajuća vrednost funkcije. Na Sl. 4-11 je prikazana tabela istinitosti logičke funkcije *y* dve promenljive, *a* i *b*.

a	b	y
0	0	0
0	1	1
1	0	0
1	1	1

Sl. 4-11 Jednostavna tabela istinitosti.

Odgovarajući VHDL opis, u kome je iskorišćena *select* naredba, dat je ispod. Signali *a* i *b* su spojeni (operatorom konkatenacije) u novi 2-bitni signal *s* koji se potom koristi kao selekcioni signal u naredbi *select*. Svakoj vrsti tabele istinitosti odgovara jedna grana u naredbi *select*.

```
1  ----------------------------------------------------------------------
2  LIBRARY IEEE;
3  USE IEEE.STD_LOGIC_1164.ALL;
4  ----------------------------------------------------------------------
5  ENTITY tabela_istinitosti IS
6    PORT (a,b : IN STD_LOGIC;
7          y : OUT STD_LOGIC);
8  END tabela_istinitosti;
9  ----------------------------------------------------------------------
10 ARCHITECTURE select_arch OF tabela_istinitosti IS
11  SIGNAL s : STD_LOGIC_VECTOR(1 DOWNTO 0);
12 BEGIN
13   s <= a & b;
14   WITH control SELECT
15      y <= '0' WHEN "00",
16           '1' WHEN "01",
17           '0' WHEN "10",
18           '1' WHEN OTHERS;
19 END select_arch;
20 ----------------------------------------------------------------------
```

S obzirom na to što je signal *s* tipa *std_logic_vector*, grana *others* (linija 18) ne pokriva samo nenavedenu binarnu vrednost, ″11″, već i svih 60 preostalih dvobitnih vrednosti tipa *std_logic* (kao što su ″ZX″, ″U0″, ″-1″ itd.). Kao što je već rečeno u Pr. 4-6, od 8 različitih vrednosti tipa *std_logic*, prilikom sinteze u obzir se uzimaju samo dve, ´1´ i ´0´. To znači da se, za potrebe sinteze, opcija *others* iz prethodne arhitekture praktično svodi na ″11″. Međutim, uprkos tome, naredba *select* koja sledi, a kod koje je opcija *others* zamenjena konkretnom vrednošću, nije sintaksno ispravna:

```
WITH s SELECT
  y <= '0' WHEN "00",
       '1' WHEN "01",
       '0' WHEN "10",
       '1' WHEN "11";
```

Razlog za sintaksnu grešku je zahtev da u *when* granama naredbe *select* moraju biti obuhvaćene sve moguće vrednosti selekcionog izraza, što za slučaj selekcionog signala tipa *std_logic* to očito nije slučaj. Napomenimo da bi gornji oblik naredbe *select* bio korektan u slučaju da je signal *s* tipa *bit* (koji poseduje samo dve binarne vrednosti ´0´ i ´1´).

Ispod su data dva kompaktnija zapisa *select* naredbe koja opisuje tabelu istinitosti sa Sl. 4-11. Kao što vidimo, u oba slučaja naredba ima samo dve *when* grane, gde jedna definiše ulazne kombinacije za koje funkcija *y* ima vrednost ´1´ (´0´), a druga (*others*) sve preostale kombinacije, tj. one za koje funkcija ima vrednost ´0´ (´1´).

```
y <=  '1' WHEN "01" | "11",       y <=  '0' WHEN "00" | "10",
      '0' WHEN OTHERS;                  '1' WHEN OTHERS;
```

Pr. 4-13 Multiplekser, binarni dekoder, binarni koder, prioritetni koder - realizacija pomoću naredbe *select*

U ovom primeru predstavljeni su VHDL opisi četiri standardna digitalna kola: multipleksera, binarnog dekodera, binarnog kodera i prioritetnog kodera. Za razliku od opisa istih kola iz poglavlja 4.2, u ovom primeru se koristi naredba *select* umesto naredbe *when*.

***Multiplekser*.** Sledi VHDL kôd arhitekture multipleksera 4-u-1 koja je funkcionalno identična arhitekturi iz Pr. 4-6.

```
1  ------------------------------------------------------------------
2  ARCHITECTURE select_arch OF mux4u1 IS
3  BEGIN
4     WITH sel SELECT
5       y <= a WHEN "00",   -- "," umesto ";"
6            b WHEN "01",
7            c WHEN "10",
8            d WHEN OTHERS; -- ne moze d WHEN "11"
9  END select_arch;
10 ------------------------------------------------------------------
```

***Binarni dekoder*.** Blok dijagram i tabela istinitosti binarnog dekodera 2-u-4 se mogu videti na Sl. 4-5 iz Pr. 4-7. U tabeli istinitosti sa Sl. 4-5(b) ulazi dekodera su navedeni u redosledu: e, d_1, d_0. U VHDL opisu koji sledi ova tri signala su najpre objedinjena u trobitni signal *ed* koji je potom iskorišćen kao kriterijum izbora u naredbi *select*. Ulazni signali su spojeni pomoću operatora konkatenacije, ″&″ (naredba *ed*<=*e*&*d* iz linije 5), tako da važi: *ed*(2)=*e*, *ed*(1)= d_1 i *ed*(0)=d_0. Za vrednosti iz prve četiri *when* grane važi da je *e*=´1´, tako da izlazi dekodera imaju iste vrednosti kao i prve četiri vrste u tabeli istinitosti sa Sl. 4-5(b). Poslednja *when* grana, koja postavlja izlaz dekodera na ″sve nule″, sadrži službenu reč *others*, što obuhvata sve slučajeve kad važi *e*=´0´.

```
1  ------------------------------------------------------------------
2  ARCHITECTURE select_arch OF dek2u4 IS
3   SIGNAL ed : STD_LOGIC_VECTOR(2 DOWNTO 0);
4  BEGIN
5   ed <= e & d;
6   WITH ed SELECT
7     y <= "0001" WHEN "100",
8          "0010" WHEN "101",
9          "0100" WHEN "110",
10         "1000" WHEN "111",
11         "0000" WHEN OTHERS;
12 END select_arch;
13 ------------------------------------------------------------------
```

***Binarni koder*.** Sledi VHDL kôd arhitekture binarnog kodera 4-u-2, koja je funkcionalno identična arhitekturi iz Pr. 4-8.

```
1  ------------------------------------------------------------------
2  ARCHITECTURE select_arch OF encoder IS
3  BEGIN
4     WITH x SELECT
5       y <= "00" WHEN "0001",
6            "01" WHEN "0010",
7            "10" WHEN "0100",
8            "11" WHEN OTHERS;
9  END select_arch;
10 ------------------------------------------------------------------
```

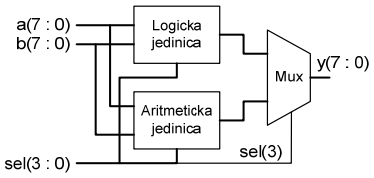
Naredba *select* kojom se izlaz kodera, *y*, postavlja na proizvoljnu vrednost u svim slučajevima kad je na njegovom ulazu prisutna neka neregularna binarna kombinacija, sledećeg je oblika:

```
WITH x SELECT
  y <= "00" WHEN "0001",
       "01" WHEN "0010",
       "10" WHEN "0100",
       "11" WHEN "1000",
       "--" WHEN OTHERS;
```

***Prioritetni koder*.** Naredba *when*, zahvaljujući redoslednom ispitivanju uslova, predstavlja idealan izbor za opis prioritetnog kodera (Pr. 4-9). Međutim, prioritetni koder je moguće opisati i pomoću naredbe *select*, ali će u tom slučaju VHDL kôd biti opširniji i manje razumljiv. U kôdu koji sledi pokazano je kako se naredbom *select* može opisati prioritetni kôder 4-u-2. Prva *when* grana (linija 5) postavlja *y* na ″00″ pod uslovom da je *x*(0) jedini ulaz koji je jednak ´1´. Sledeće *when* grana postavlja *y* na ″01″ ako važi *x*(3)=*x*(2)=´0´ i *x*(1)=´1´. Treća *when* grana postavlja *y* na ″10″ za sve kombinacije ulaznih vrednosti za koje veži: *x*(3)=´0´ i *x*(2)=´1´. Konačno, poslednja *when* grana kaže da u svim preostalim slučajevima, što uključuje i slučaj kad je *x*(3)=´1´, *y* dobija vrednost ″11″. Uočimo da grana *when others* obuhvata i ulaznu kombinaciju “0000”, što znači da će u tom slučaju izlaz *y* biti postavljen na ″11″. Međutim, kako sve nule na ulazu postavljaju *z*=´0´ (linije 9-11), to u ovom slučaju vrednost izlaza *y* nije od značaja. Primetimo da je broj binarnih kombinacija sadržanih u svakoj sledećoj *when* grani dva puta veći u odnosu na prethodnu. Iz tog razloga, opis prioritetnog kodera pomoću naredbe *select* nije primeren onda kad je broj ulaza veći od 4.

```
1  ------------------------------------------------------------------
2  ARCHITECTURE select_arch OF pencoder IS
3  BEGIN
4  WITH x SELECT
5    y <="00" WHEN "0001",
6        "01" WHEN "0010" | "0011",
7        "10" WHEN "0100" | "0101" | "0110" | "0111",
8        "11" WHEN OTHERS;
9   WITH x SELECT
10    z <= '0' WHEN "0000",
11         '1' WHEN OTHERS;
12 END select_arch;
```

Pr. 4-14 ALU



(a)

sel	Operacija	Funkcija	Jedinica
0000	y <= a	Transfer a	Aritmetička
0001	y <= a + 1	Inkrement a	
0010	y <= a - 1	Dekrement a	
0011	y <= b	Transfer b	
0100	y <= b + 1	Inkrement b	
0101	y <= b - 1	Dekrement b	
0110	y <= a + b	Sabiranje	
0111	y <= a-b	Oduzimanje	
1000	y <= NOT a	Komplement a	Logička
1001	y <= NOT b	Komplement b	
1010	y <= a AND b	I	
1011	y <= a OR b	ILI	
1100	y <= a NAND b	NI	
1101	y <= a NOR b	NILI	
1110	y <= a XOR b	Iskljucivo ILI	
1111	y <= a NXOR b	Iskljucivo NILI	

(b)

Sl. 4-12 ALU: (a) Blok dijagram; (b) funkcionalna tabela.

Aritmetičko-logička jedinica (ili ALU, engl. ***A**rithmetic and **L**ogic **U**nit*) je višefunkcionalno kombinaciono kolo koje može da obavlja više različitih aritmetičkih i logičkih operacija nad parom višebitnih operanada. Na Sl. 4-12(a) je prikazan blok dijagram, a na Sl. 4-12(b) funkcionalna tabela jedne jednostavne 8-bitne ALU koja podržava osam aritmetičkih i osam logičkih operacija. Izbor između aritmetičkih i logičkih operacija vrši se bitom najveće težine 4-bitnog selekcionog signala *sel*. Preostala tri bita signala *sel* biraju jednu od osam operacija iz svake grupe.

```
1  ----------------------------------------------------------------------
2  LIBRARY IEEE;
3  USE IEEE.STD_LOGIC_1164.ALL;
4  USE IEEE.NUMERIC_STD.ALL;
5  ----------------------------------------------------------------------
6  ENTITY ALU IS
7  PORT (a,b : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
8        sel : IN STD_LOGIC_VECTOR(3 DOWNTO 0);
9        y : OUT STD_LOGIC_VECTOR(7 DOWNTO 0));
10 END ALU;
11 ----------------------------------------------------------------------
12 ARCHITECTURE select_arch OF ALU IS
13  SIGNAL arith, logic : STD_LOGIC_VECTOR(7 DOWNTO 0);
14  SIGNAL inc_a, dec_a, inc_b,
15         dec_b, sum, dif : STD_LOGIC_VECTOR(7 DOWNTO 0);
16 BEGIN
17 -- Aritmeticka jedinica ------------------------------------------
18  inc_a <= STD_LOGIC_VECTOR(SIGNED(a) + 1);
19  dec_a <= STD_LOGIC_VECTOR(SIGNED(a) - 1);
20  inc_b <= STD_LOGIC_VECTOR(SIGNED(b) + 1);
21  dec_b <= STD_LOGIC_VECTOR(SIGNED(b) - 1);
22  sum   <= STD_LOGIC_VECTOR(SIGNED(a) + SIGNED(b));
23  dif   <= STD_LOGIC_VECTOR(SIGNED(a) - SIGNED(b));
```

```
24  WITH sel(2 DOWNTO 0) SELECT
25    arith <= a     WHEN "000",
26             inc_a WHEN "001",
27             dec_a WHEN "010",
28             b     WHEN "011",
29             inc_b WHEN "100",
30             dec_b WHEN "101",
31             sum   WHEN "110",
32             dif   WHEN OTHERS;
33 -- Logicka jedinica ------------------------------------------------
34  WITH sel(2 DOWNTO 0) SELECT
35    logic <= NOT a    WHEN "000",
36             NOT b    WHEN "001",
37             a AND b  WHEN "010",
38             a OR b   WHEN "011",
39             a NAND b WHEN "100",
40             a NOR b  WHEN "101",
41             a XOR b  WHEN "110",
42             NOT (a XOR b) WHEN OTHERS;
43  -- Mux -------------------------------------------------------------
44   WITH sel(3) SELECT
45     y <= arith WHEN '0',
46          logic WHEN OTHERS;
47 END select_arch;
48 ------------------------------------------------------------------------
```

Organizacija VHDL kôda usklađena je sa strukturom ALU (Sl. 4-12(a)). Svakom bloku sa Sl. 4-12(a) odgovara jedna naredba *select*. Kao što tri bloka u fizičkoj ALU (*Logicka jedinica, Aritmeticka jedinica* i *Mux*) rade paralelno, tako se i tri *select* naredbe u odgovarajućem VHDL kôdu izvršavaju konkurentno. Za spregu konkurentnih sekcija kôda koriste se dva interna signala, *arith* i *logic*, koji odgovaraju vezama između blokova *Logicka jedinica* i *Mux*, odnosno *Aritmeticka jedinica* i *Mux* u strukturi sa Sl. 4-12(a). Takođe, treba zapaziti da se u prezentovanom kôdu aritmetičke operacije obavljaju nad označenim celim brojevima uz podršku paketa *numeric_std* (linije 18-23).

Konceptualna implementacija naredbe select. Kao što znamo, naredba *select* ima sledeću sintaksu:

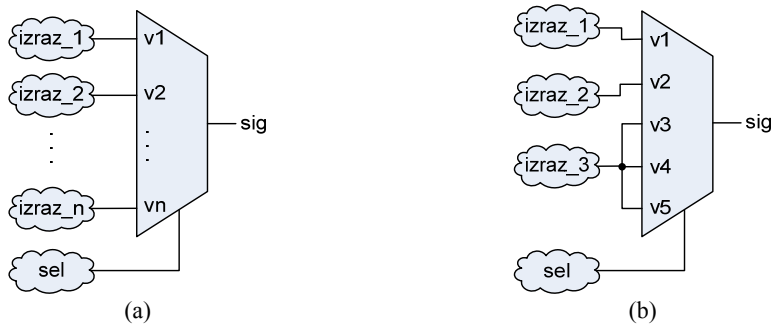
```
WITH sel SELECT
   sig <= izraz_1 WHEN v1,
          izraz_2 WHEN v2,
             ...
          izraz_n WHEN vn;
```

U suštini, naredba *select* se može interpretirati kao apstraktni multiplekser kod koga se selekcioni izraz *sel* koristi za izbor jednog od *n* izraza čiji će rezultat biti dodeljen izlaznom signalu *sig*. Kod jednog ovakvog multipleksera (Sl. 4-13(a)), svakoj mogućoj vrednosti rezultata selekcionog izraza (*v1*, ..., *vn*) odgovara jedan ulazni port na koji se dovodi rezultat odgovarajućeg izraza (*izraz_1*, ..., *izraz_n*). Za razliku od apstraktnog multipleksera za naredbu *when*, selekcioni signal multipleksera za naredbu *select* ne mora biti tipa *boolean*, već može biti proizvoljnog tipa.

Razmotrimo sledeću *select* naredbu sa *when others* granom:

```
WITH sel SELECT
   sig <= izraz_1 WHEN v1,
          izraz_2 WHEN v2,
          Izraz_3 WHEN OTHERS;
```

Pod pretpostavkom da su *v*1, *v*2, *v*3, *v*4 i *v*5 sve moguće vrednosti selekcionog signala *sel*, grana *when others* implicitno pokriva vrednosti *v*3, *v*4 i *v*5. Na Sl. 4-13(b) je prikazana konceptualna realizacija ove naredbe.



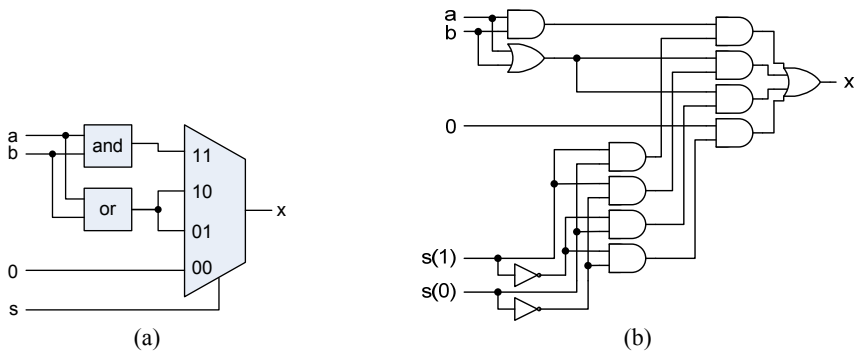
Sl. 4-13 Konceptualna implementacija *select* naredbe: (a) bez *others*; (b) sa *others* granom.

Sve *select* naredbe se prevode u sličan konceptualni dijagram. Razlika je samo u broju mogućih vrednosti selekcionog izraza, što direktno određuje veličinu multipleksera. Napomenimo da uprkos jednostavnoj konceptualnoj realizaciji, sinteza naredbe *select* može biti otežana činjenicom da kod izvesnih implementacionih tehnologija postoje poteškoće prilikom realizacije ekstremno ″širokih″ multipleksera.

Pr. 4-15 Konceptualna implementacija naredbe *select* - prvi primer

Na Sl. 4-14(a) je prikazan konceptualni dijagram koji odgovara sledećem segmentu VHDL kôda:

```
SIGNAL s: STD_LOGIC_VECTOR(1 DOWNTO 0);
. . .
WITH s SELECT
   x <= (a AND b) WHEN "11",
        (a OR b)  WHEN "01" | "10",
        '0'       WHEN OTHERS;
```



Sl. 4-14 Konceptualna implementacija i sinteza *select* naredbe iz Pr. 4-15: (a) konceptualni dijagram; (b) kombinaciona mreža.

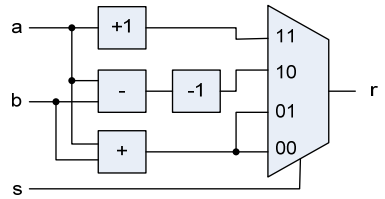
Uočimo da je selekcioni izraz *s* tipa *std_logic_vector*(1 *downto* 0). Od moguće 64 dvobitne vrednosti signala *s*, za sintezu su smislene samo četiri: ″00″, ″01″, ″10″ i ″11″, što znači da je za realizaciju ove *select* naredbe dovoljan multiplekser 4-u-1. Daljom razradom konceptualnog dijagrama sa Sl. 4-14(a), tj. realizacijom apstraktnih blokova *and* i *or* i apstraktnog multipleksera pomoću logičkih kola dolazimo do kombinacione mreže sa Sl. 4-14(b), koja predstavlja i konačni rezultat sinteze razmatrane *select* naredbe.

Pr. 4-16 Konceptualna implementacija naredbe *select* - drugi primer

Razmotrimo *select* naredbu iz sledećeg segmenta VHDL kôda:

```
SIGNAL a,b,r: UNSIGNED(7 DOWNTO 0);
SIGNAL s: STD_LOGIC_VECTOR(1 DOWNTO 0);
. . .
WITH s SELECT
   r <= a+1 WHEN "11",
        a-b-1 WHEN "10",
        a+b WHEN OTHERS;
```

Odgovarajući konceptualni dijagram je prikazan na Sl. 4-15.



Sl. 4-15 Konceptualna implementacija VHDL kôda iz Pr. 4-16.

4.4. Poređenje naredbi WHEN i SELECT

Primeri iz prethodnih poglavlja pokazuju da se isto kombinaciono kolo može opisati korišćenjem bilo *when* bilo *select* naredbe. Međutim, posmatrano sa stanovišta hardverske sinteze, ove dve naredbe se sintetišu na drugačiji način. Sinteza *when* naredbe se zasniva na prioritetnoj strukturi formiranoj od kaskadno povezanih multipleksera. Nasuprot tome, za sintezu *select* naredbe koristi se samo jedan ″široki″ multiplekser sa ulazima jednakog prioriteta. Naredba *select* je pogodna za opisivanje kombinacionih kola poput dekodera, multipleksera i ALU, čija se funkcija može definisati tabelom istinitosti ili nekom vrstom funkcionalne tabele – svaka grana *select* naredbe odgovara jednoj vrsti tabele istinitosti. S druge strane, *when* naredba omogućava kreiranje konciznih opisa kola koja nekim ulazima ili operacijama daju viši prioritet, kao što je to slučaj s prioritetnim koderom. Takođe, naredba *when* omogućava koncizno opisivanje složenih uslova i izraza, npr:

```
r <= a+b WHEN (x+y>1 AND x>3) ELSE
     a-b-1 WHEN (z>v AND f = '1') ELSE
     . . .
```

Međutim, *when* naredba je manje efikasna onda kad treba opisati tabelu istinitosti, jer u opis implicitno unosi ograničenja (prioritete) koja faktički ne postoje. Na primer, sledeće tri *when* naredbe su funkcionalno identične:

```
x<=a WHEN (s="00") ELSE    x<=b WHEN (s="01") ELSE    x<=c WHEN (s="10") ELSE
   b WHEN (s="01") ELSE       a WHEN (s="00") ELSE       b WHEN (s="01") ELSE
   c WHEN (s="10") ELSE       c WHEN (s="10") ELSE       a WHEN (s="00") ELSE
   d;                         d;                         d;
```

Svaka od tri *when* naredbe opisuje multiplekser 4-u-1, ali pri tom dajući različite prioritete ulazima. Međutim, s obzirom na to što se u svim granama kao uslov koristi isti signal, *s*, koji u jednom trenutku može imati samo jednu vrednost, prioriteti se gube, jer će za isto *s* svaka od tri naredbe izabrati isti ulaz bez obzira na redosled ispitivanja. Iako ovakva vrsta opisa nije pogrešna, dodatna ograničenja uvedena *when* naredbom mogu da zahtevaju dodatni hardver i učiniti sintezu i optimizaciju težim.

Idealno, softver za sintezu bi trebalo da je u mogućnosti da automatski odredi optimalnu hardversku strukturu i generiše identičnu gejtovsku realizaciju bez obzira na jezičke konstrukcije koje su korišćene u VHDL opisu. Međutim, to je slučaj samo za veoma jednostavna, trivijalna kola. U opštem slučaju, projektant treba da bude svestan efekta naredbi na sintezu, kako bi u svakom konkretnom slučaju bio u stanju da izabere jezičku konstrukciju koja semantički najbolje odgovara funkciji kola koje opisuje.

4.5. Optimizacija konkurentnog kôda

Karakteristično je za probleme iz oblasti projektovanja digitalnih sistema da gotovo nikada ne postoji samo jedno rešenje. Jedan isti sistem se može realizovati na mnogo različitih načina, a alternativne realizacije se mogu značajno razlikovati po složenosti i performansama. Korišćenje VHDL-a i softvera za sintezu, samo po sebi, ne znači da će realizovano rešenje biti dobrih ili loših karakteristika. VHDL i softver za sintezu pre svega nas rasterećuju implementacionih detalja niskog nivoa i značajno pojednostavljuju i ubrzavaju proces realizacije hardvera. Softver je u stanju da prilikom sinteze automatski obavi izvesna pojednostavljenja i lokalne optimizacije, ali svakako nije u stanju da ″razume″ svrhu kôda kako bi istražio alternativna rešenja ili promenio arhitekturu sistema. Odgovornost za kvalitet rešenja je na projektantu. Pri tome, ne postoji neki mehanički projektantski postupak koji bi garantovao pronalaženje optimalnog rešenje, već se projektovanje u velikoj meri oslanja na iskustvo projektanta i njegovo dobro razumevanje problema koji treba da reši.

U ovom poglavlju biće predstavljene dve optimizacione tehnike: deoba operatora i deoba funkcija, koje se mogu koristiti radi kreiranja efikasnijeg (u pogledu složenosti sintetizovanog hardvera) VHDL kôda.

4.5.1. Deoba operatora

Sinteza VHDL kôda podrazumeva prevođenje naredbi i jezičkih konstrukcija sadržanih u kôdu u odgovarajuću hardversku strukturu. Jedan od glavnih kriterijuma za ocenu kvaliteta hardverskog rešenja jeste veličina (obimnost) sintetizovanog hardvera izražena na primer brojem logičkih kola ili površinom zauzetom na silicijumskom čipu. Veličina sintetizovanog hardvera zavisi od mnogih faktora, a jedan od najznačajnijih je vrsta i broj operatora (aritmetičkih, kao što su ″+″ i ″-″, i u nešto manjoj meri logičkih, kao što su *and*, *or*, *xor*) sadržanih u kôdu. Pojam *deoba operatora* se odnosi na optimizacionu tehniku koja se sprovodi na nivou VHDL kôda u cilju smanjenja broj operatora. Optimizacija se postiže preuređenjem kôda na način da se isti operator može iskoristiti za obavljanje više različitih

operacija. Budući da deoba operatora, po pravilu, zahteva uvođenje dodatnog hardvera (najčešće u vidu multipleksera), njena primena je opravdana samo za relativno složene operatore. Osnovnu ideju deobe operatora ilustrovaćemo na primeru sledeće *when* naredbe:

```
r <= a + b WHEN uslov ELSE
     a + c;
```

Konceptualna implementacija prethodne naredbe sadrži dva sabirača i jedan multiplekser (Sl. 4-16(a)). Međutim, primetimo da su izrazi *a*+*b* i *a*+*c* uzajamno isključivi u smislu da se u zavisnosti od vrednosti uslova za izvršenje uvek bira samo jedan. Drugim rečima, od dva sabirača iz dijagrama sa Sl. 4-16(a) uvek će biti korišćen samo jedan.

(a) (b)

Sl. 4-16 Deobe operatora na primeru *when* naredbe: (a) konceptualna implementacija neoptimizovanog kôda; (b) konceptualna implementacija nakon obavljene deobe operatora.

Efikasnije rešenje, u pogledu hardverske složenosti, bilo bi ono koje sadrži samo jedan sabirač, a koji u zavisnosti od vrednosti uslova sabira *a* i *b* ili *a* i *c*. To se može postići revizijom prethodnog kôda na sledeći način:

```
x <= b WHEN uslov ELSE
     c;
r <= a + x;
```

Odgovarajući blok dijagram prikazan je na Sl. 4-16(b). Dakle, umesto da se multipleksiraju rezultati sabiranja, kao u polaznom kôdu, sada se multipleksiraju sabirci. Na taj način, sabirač postaje deljiv hardverski resurs, takav da se može koristiti za izračunavanje bilo operacije *a*+*b*, bilo operacije *a*+*c*.

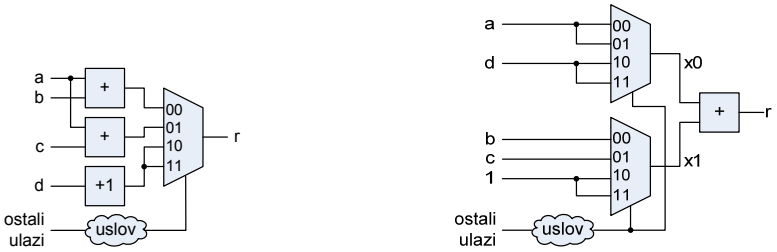
Iako je drugo rešenje hardverski jednostavnije, prvo rešenje poseduje bolje performanse, tj. manje propagaciono kašnjenje. Neka su propagaciona kašnjenja sabirača, multipleksera i logike koja određuje vrednost uslova redom: T_{sab}, T_{mux} i T_{uslov}. Propagaciono kašnjenje prvog kola iznosi: $max\{T_{sab}, T_{uslov}\} + T_{mux}$, a drugog $T_{sab} + T_{uslov} + T_{mux}$. Manje propagaciono kašnjenje prvog rešenja je posledica činjenice da se sabiranje i ispitivanje uslova obavljaju u paraleli, za razliku od drugog rešenja u kojem se ove dve operacije obavljaju jedna za drugom. Međutim, ukoliko je logika za određivanje uslova jednostavna, performanse dva rešenja biće približno iste.

Pr. 4-17 Deoba operatora na primeru naredbe *select*

Razmotrimo sledeću *select* naredbu:

```
WITH uslov SELECT
  r <= a+b WHEN "00",
       a+c WHEN "01",
       d+1 WHEN OTHERS;
```

Odgovarajuća konceptualna implementacija prikazana je na Sl. 4-17(a). Kao što se može videti, kolo sadrži dva sabirača, jedan inkrementer i jedan multiplekser 4-u-1.



Sl. 4-17 Deobe operatora na primeru *select* naredbe: (a) konceptualna implementacija neoptimizovanog kôda; (b) konceptualna implementacija nakon obavljene deobe operatora.

Kôd optimizovan deobom operatora sadrži samo jedan operator sabiranja i oblika je:

```
WITH uslov SELECT
  x0 <= a WHEN ˝00˝ | ˝01˝,
        d WHEN OTHERS;
WITH uslov SELECT
  x1 <= b WHEN ˝00˝,
        c WHEN ˝01˝,
        ˝00000001˝ WHEN OTHERS;
r <= x0 + x1;
```

Blok dijagram konceptualne implementacije optimizovanog kôda prikazan je na Sl. 4-17(b). Kao što se može uočiti, optimizovano rešenje sadrži jedan sabirač i inkrementer manje, ali i jedan multiplekser 4-u-1 više u odnosu na prvobitno rešenje. Ušteda u hardveru je ipak značajna, budući da su sabirač i inkrementer višestruko složenija kola od multipleksera

U zaključku, deoba operatora se ostvaruje zamenom više operatora jednim, uz ugradnju u kolo dodatnih multipleksera za izbor operanada. Nivo ostvarene uštede zavisi od relativnog odnosa složenosti operatora i dodatnih multipleksera. Što je operator složeniji, to je ušteda veća. Međutim, deoba operatora po pravilu unosi dodatno propagaciono kašnjenje, jer forsira kaskadno izvršenje uslova za izbor operanada i deljive operacije. Konačno, treba napomenuti da mnogi alati za hardversku sintezu poseduju podršku za automatsku optimizaciju kôda kroz deobu operatora.

4.5.2. Deoba funkcija

Složeni digitalni sistemi, po pravilu, realizuju mnoštvo funkcija. Moguće je da različite funkcije poseduju neke zajedničke karakteristike, ili da se na neki način mogu dovesti u vezu. To otvara mogućnost da se više funkcija realizuju tako da dele neke zajedničke delove ili da se jedna funkcija iskoristiti za realizaciju neke druge funkcije. Ova vrsta optimizacije se naziva *deobom funkcija*. Za razliku od deobe operatora, deoba funkcija se ne može automatizovati, već se primenjuje na *ad-hoc* način, od slučaja do slučaja, a njena uspešna primena zavisi od toga do koje mere projektant razume osobine funkcija koje treba da realizuje.

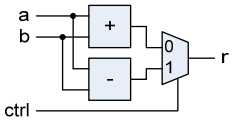
Pr. 4-18 Sabirač/oduzimač

U ovom primeru je predstavljeno višefunkcionalno kolo koje je u stanju da obavlja operacije sabiranja i oduzimanja. Funkcionalna tabela ovog kola (sabirača/oduzimača) prikazana je na Sl. 4-18(a). Operacija se bira posredstvom selekcionog signala *ctrl*. Kolo

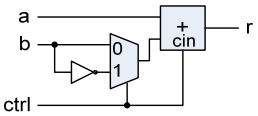
obavlja sabiranje za *ctrl*=′0′, a oduzimanje za *ctrl*=′1′. U nastavku su predstavljena dva VHDL opisa sabirača/oduzimača. Prvi opis (arhitektura *arch_v1*) striktno sledi funkcionalnu tabelu sa Sl. 4-18(a). Uočimo da su ulazni signali interno konvertovani u tip *signed* (linije 15 i 16) kako bi se u *when* naredbi (linije 17-18) omogućila primena aritmetičkih operacija. Na Sl. 4-18(b) je prikazan odgovarajući konceptualni dijagram, koji sadrži sabirač (za operaciju ″+″ iz linije 17), oduzimač (za operaciju ″-″ iz linije 18) i izlazni multiplekser (za *when* naredbu iz linija 17 i 18).

ctrl	operacija
0	a + b
1	a - b

(a)



(b)



(c)

Sl. 4-18 Deoba funkcija na primeru kola za sabiranje/oduzimanje: (a) funkcionalna tabela; (b) polazno rešenje; (c) optimizovano rešenje.

Budući da su sabiranje i oduzimanje dve različite operacije, direktna primena deobe operatora nije moguće. Međutim, u reprezentaciji potpunog komplementa važi da se oduzimanje, *a* - *b*, može izraziti preko sabiranja kao $a+\overline{b}+1$, gde je $\overline{b}$ jedinični komplement umanjioca *b*. Pošto sada u obe funkcije figuriše samo operator ″+″, omogućeno je da se funkcija sabiranja iskoristi za realizacije kako *a*+*b* tako i *a*-*b*, i to na način kao u arhitekturi *arch_v2*.

```
1  ----------------------------------------------------------------------
2  LIBRARY IEEE;
3  USE IEEE.STD_LOGIC_1164.ALL;
4  USE IEEE.NUMERIC_STD.ALL;
5  ----------------------------------------------------------------------
6  ENTITY ADDSUB IS
7  PORT (a,b : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
8        ctrl : IN STD_LOGIC;
9        r : OUT STD_LOGIC_VECTOR(7 DOWNTO 0));
10 END ADDSUB;
11 ----------------------------------------------------------------------
12 ARCHITECTURE arch_v1 OF ADDSUB IS
13   SIGNAL x0, x1, sum : SIGNED(7 DOWNTO 0);
14 BEGIN
15   x0 <= SIGNED(a);
16   x1 <= SIGNED(b);
17   sum <= x0 + x1 WHEN ctrl = '0' ELSE
18          x0 - x1;
19   r <= STD_LOGIC_VECTOR(sum);
20 END arch_v1;
21 ----------------------------------------------------------------------
22 ARCHITECTURE arch_v2 OF ADDSUB IS
23   SIGNAL x0, x1, sum : UNSIGNED(7 DOWNTO 0);
24   SIGNAL cin : UNSIGNED(0 DOWNTO 0);  -- bit ulaznog prenosa
25 BEGIN
26   x0 <= UNSIGNED(a);
27   x1 <= UNSIGNED(b) WHEN ctrl='0' ELSE
28         UNSIGNED(NOT b);
29   cin <= "0" WHEN ctrl='0' ELSE
30          "1";
31   sum <= x0 + x1 + cin;
```

```
32   r <= STD_LOGIC_VECTOR(sum);
33 END arch_v2;
34 ---------------------------------------------------------------------
```

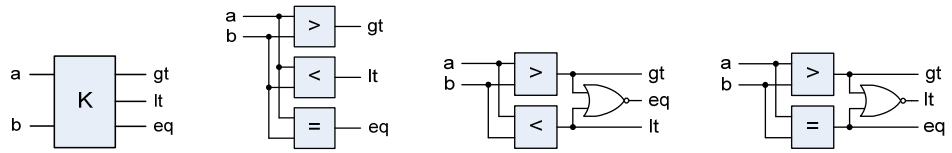
Uočimo da naredba *sum* <= $x0 + x1 + cin$ iz arhitekture *arch_v2* sadrži dve operacije sabiranja, što sugeriše da su za njenu realizaciju potrebna dva sabirača. Međutim, pošto je *cin* jednobitni signal, koji može imati vrednost ´0´ ili ´1´, moguće je koristiti samo jedan sabirač sa portom za ulazni prenos na koji će biti povezan signal *cin*. Većina alata za sintezu mogu da prepoznaju ovakvu situaciju i konvertuju dati kôd u rešenje sa jednim sabiračem. Takođe, uočimo da se naredba *when* koja određuje vrednost signala *cin* (linije 29-30) svodi na naredbu dodele: *cin* <= *ctrl*. Imajući to u vidu, konceptualni dijagram arhitekture *arch_v2* je oblika kao na Sl. 4-18(c). Simbol za invertor koji se vidi na ovoj slici ne predstavlja samo jedno kolo, već niz od 8 invertora od kojih svaki invertuje jedan bit 8-bitnog operanda *b*. Optimizovano rešenje ipak ima manju složenost od polaznog budući da je složenost oduzimača veća od složenosti niza invertora. Međutim, složenost je smanjena po cenu povećanja propagacionog kašnjenja za iznos kašnjenja kroz invertor.

Pr. 4-19 Potpuni komparator

Potpuni komparator je digitalno kolo za poređenje binarnih brojeva (Sl. 4-19(a)). Komparator poseduje dva ulaza, *a* i *b*, za brojeve koji se porede i tri izlaza čije vrednosti ukazuju na ishod poređenja: *gt* - veće (*a* > *b*), *lt* - manje (*a* < *b*) i *eq* - jednako (*a* = *b*). Direktno rešenje podrazumeva korišćenje tri relaciona operatora, kao u arhitekturi *arch_v1* iz VHDL kôda koji sledi. Očigledno, za sintezu arhitekture *arch_v1* neophodna su tri nezavisna kola za poređenje (Sl. 4-19(b)). Međutim, lako se uočava da su tri uslova (<, > i =) međusobno isključiva, u smislu da se na osnovu bilo koja dva može izvesti treći. Na primer, ″*a* je jednako *b*″ ako važi da nije ″*a* veće od *b*″ i nije ″*a* manje od *b*″. Imajući to u vidu, funkcionalnost kola koja ispituju uslove ″veće″ i ″manje″ se može iskoristiti za realizaciju uslova ″jednako″: $eq = \overline{gt} \cdot \overline{lt} = \overline{(gt + lt)}$. Ovo zapažanje je iskorišćeno u arhitekturi *arch_v2* da bi se kreiralo kompaktnije rešenje (Sl. 4-19(c)). Budući da je od tri relaciona operatora, operator jednakosti najjednostavniji za hardversku realizaciju, još efikasnije rešenje se može dobiti ako se, kao u arhitekturi *arch_v3*, operator ″<″ izrazi preko operatora ″>″ i ″=″ (Sl. 4-19(d)).

```
1  ---------------------------------------------------------------------
2  LIBRARY IEEE;
3  USE IEEE.STD_LOGIC_1164.ALL;
4  ---------------------------------------------------------------------
5  ENTITY COMP3 IS
6  PORT (a,b : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
7        gt,lt,eq : OUT STD_LOGIC);
8  END COMP3;
9  ---------------------------------------------------------------------
10 ARCHITECTURE arch_v1 OF COMP3 IS
11 BEGIN
12   gt <= '1' WHEN a > b ELSE
13         '0';
14   lt <= '1' WHEN a < b ELSE
15         '0';
16   eq <= '1' WHEN a = b ELSE
17         '0';
18 END arch_v1;
```

```
19 ------------------------------------------------------------------------
20 ARCHITECTURE arch_v2 OF COMP3 IS
21   SIGNAL xgt,xlt : STD_LOGIC;
22 BEGIN
23   xgt <= '1' WHEN a > b ELSE
24          '0';
25   xlt <= '1' WHEN a < b ELSE
26          '0';
27   gt <= xgt;
28   lt <= xlt;
29   eq <= xgt NOR xlt;
30 END arch_v2;
31 ------------------------------------------------------------------------
32 ARCHITECTURE arch_v3 OF COMP3 IS
33   SIGNAL xgt,xeq : STD_LOGIC;
34 BEGIN
35   xgt <= '1' WHEN a > b ELSE
36          '0';
37   xeq <= '1' WHEN a = b ELSE
38          '0';
39   gt <= xgt;
40   eq <= xeq;
41   lt <= xgt NOR xeq;
42 END arch_v3;
43 ------------------------------------------------------------------------
```



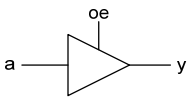
Sl. 4-19 Deoba funkcija na primeru potpunog komparatora: (a) simbol potpunog komparatora; (b) direktna realizacija; (c) prvo optimizovano rešenje; (d) drugo optimizovano rešenje.

4.6. ´Z´

Tip podataka *std_logic* definiše osam vrednosti signala. Vrednosti ´0´ i ´1´ se interpretiraju kao logička 0 i logičko 1 i koriste se u regularnoj logičkoj sintezi. Vrednosti ´L´ i ´H´, tj. slaba 0 i slaba 1-ca, odgovaraju nivoima signala koje generišu kola sa izlaznim stepenom tipa ″otovreni kolektor″, odnosno ″otvoreni emitor″. Budući da se ovakva vrsta digitalnih kola danas retko koristi u praksi, ni vrednosti ´H´ i ´L´ ne bi trebalo koristiti u VHDL-u. Vrednosti ´X´ i ´W´ imaju smisla samo u simulaciji i ne mogu se sintetizovati. Vrednost ´-´ predstavlja proizvoljnu vrednost (*″don't care″*) i može se koristiti u kôdu za sintezu, na način kako je to opisano u Pr. 4-8. Konačno, ´Z´ označava ″visoku impedansu″ ili ″otvoreno kolo″.

Trostatički bafer. ´Z´ nije logička vrednost (u smislu Bulove algebre), već predstavlja električnu osobinu koju poseduje samo jedna posebna vrsta digitalnih kola, poznata pod nazivom *trostički bafer*. Na Sl. 4-20 su prikazani grafički simbol i funkcionalna tabela trostatičkog bafera. Za *oe*=´1´, bafer prenosi logičku vrednost s ulaza na izlaz, odnosno deluje kao ″zatvoreno kolo″. S druge strane, za *oe*=´0´, izlaz bafera je postavljen u stanje

visoke impedanse, odnosno deluje kao ″otvoreno kolo″. Drugim rečima, pri *oe*=´0´ izlaz bafera ne generiše ni logičku 0 ni logičko 1, već je u tzv. *trećem stanju* ili stanju *visoke impedanse*, tj. bez obzira na trenutnu logičku vrednost na ulazu, vrednost na izlazu je ´Z´.



(a)

oe	y
0	Z
1	a

(b)

Sl. 4-20 Trostatički bafer: (a) grafički simbol; (b) funkcionalna tabela.

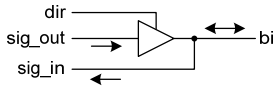
U VHDL-u, trostatički bafer se opisuje na sledeći način:

```
y <= a WHEN oe='1' ELSE
     'Z';
```

'Z' se ne može koristiti kao ulazna vrednost, niti se s ovom vrednošću može manipulisati na način kao sa logičkim vrednostima ´0´ i ´1´. Na primer, sledeće dve naredbe se ne mogu sintetizovati:

```
r <= 'Z' AND a;
g <= d - c WHEN a = 'Z' ELSE
     d - b;
```

***Bidirekcioni (ulazno/izlazni) port*.** Pojedini pinovi integrisanih kola se izvode kao bidirekcioni, odnosno na način da se mogu, po potrebi, koristiti bilo kao ulazi bilo kao izlazi. Konceptualni prikaz bidirekcionog porta prikazan je na Sl. 4-21. Smer porta se kontroliše pomoću signala *dir*. Kada je *dir*=´0´, trostatički bafer deluje kao otvoreno kolo, koje raskida vezu između unutrašnjeg signala *sig_out* i porta *bi*, što pruža mogućnost da se *bi* koristiti kao ulazni port. Za *dir*=´1´, bafer deluje kao zatvoreno kolo koje prenosi vrednost unutrašnjeg signal *sig_out* na sada izlazni port *bi*.



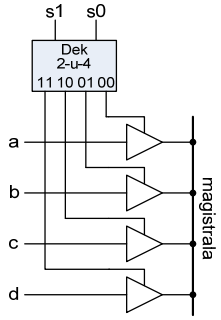
Sl. 4-21 Realizacija bidirekcionog porta pomoću trostatičkog bafera.

Sledi VHDL opis bidirekcionog porta. U entitetu, bidirekcioni port se deklariše sa smerom *inout*. U kôdu arhitekture uočavamo naredbu *when* kojom se kreira trostatički bafer sa ulazom *sig_out*, izlazom *bi* i kontrolnim ulaznom *dir*.

```
ENTITY bi_port IS
  PORT(...
       bi : INOUT STD_LOGIC;
       ...);
BEGIN
  ...
  bi <= sig_out WHEN dir = '1' ELSE
        'Z';
  sig_in <= bi;
  ...
```

***Trostatička magistrala*.** U digitalnim sistemima se često javlja potreba za prenosom podataka iz više različitih izvora do istog odredišta. Ova funkcija se najčešće ostvaruje pomoću multipleksera. Međutim, upotreba multipleksera sa velikim brojem ulaza može predstavljati problem, budući da veliki broj veza mora biti doveden do jednog mesta u sistemu, tamo gde je postavljen multiplekser. U takvim slučajevima, često je racionalnije

umesto multipleksera koristiti magistralu. Konstrukcija magistrale je zasnovana na trostatičkim baferima koji obezbeđuju kontrolisanu spregu izvora podatka sa zajedničkom prenosnom linijom, kao što je prikazano na Sl. 4-22. Pošto u jednom vremenu najviše jedan izvor može da pobuđuje magistralu, jer bi se u suprotnom desio konflikt na magistrali, neophodno je obezbediti da u svakom trenutku najviše jedan trostatički bafer bude aktivan (tj. deluje kao ″zatvoreno kolo″), što se postiže uz pomoć dekodera.



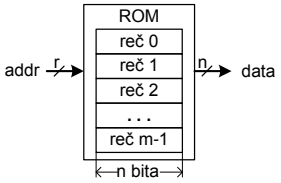
Sl. 4-22 Trostatička magistrala.

U zavisnosti od vrednosti selekcionog signala, *s*, dekoder ″zatvara″ tačno jedan trostatički bafer, koji na zajedničku magistralu prenosi vrednost sa svog ulaza. Izlazi ″otvorenih″ trostatičkih bafera su u stanju visoke impedanse i zato ne utiču na logički nivo signala na zajedničkoj liniji. U nastavku je dat VHDL opis magistrale sa četiri izvora podataka. Kôd sadrži četiri *when* naredbe, od kojih svaka odgovara jednom trostatičkom baferu.

```
1  ----------------------------------------------------------------------
2  LIBRARY IEEE;
3  USE IEEE.STD_LOGIC_1164.ALL;
4  ----------------------------------------------------------------------
5  ENTITY bus IS
6    PORT ( a,b,c,d : IN STD_LOGIC;
7           s: IN STD_LOGIC_VECTOR(1 DOWNTO 0);
8           y : OUT STD_LOGIC);
9  END bus;
10 ----------------------------------------------------------------------
11 ARCHITECTURE arch OF bus IS
12 BEGIN
13     y <= a WHEN s = "00" ELSE 'Z';
14     y <= b WHEN s = "01" ELSE 'Z';
15     y <= c WHEN s = "10" ELSE 'Z';
16     y <= d WHEN s = "11" ELSE 'Z';
17 END arch;
18 ----------------------------------------------------------------------
```

4.7. ROM

ROM (engl. ***R**ead-**O**nly **M**emory*) je memorija sa fiksnim sadržajem, tj. memorijska komponenta koja može samo da se čita. Blok dijagram ROM-a prikazan je na Sl. 4-23. ROM čini *m* memorijskih lokacija, ili reči, od po *n* bita. Na *r*-bitni ulazni port *addr* postavlja se adresa (redni broj) lokacije koja se čita, dok se sa izlaznog porta *data* preuzima sadržaj adresirane reči. Broj reči u ROM-u i broj bita adrese su sledećoj vezi: $r = \lceil \log_2 m \rceil$. Sledi VHDL opis ROM-a 8x8, tj. ROM-a sa 8 memorijskih reči dužine 8 bita.



Sl. 4-23 ROM.

```
1  ----------------------------------------------------------------
2  LIBRARY IEEE;
3  USE IEEE.STD_LOGIC_1164.ALL;
4  USE IEEE.NUMERIC_STD.ALL;
5  ----------------------------------------------------------------
6  ENTITY ROM IS
7    PORT (addr : IN STD_LOGIC_VECTOR(2 DOWNTO 0);
8          data : OUT STD_LOGIC_VECTOR(7 DOWNTO 0));
9  END ROM;
10 ----------------------------------------------------------------
11 ARCHITECTURE rom OF ROM IS
12  TYPE mem_array IS ARRAY (0 TO 7) OF STD_LOGIC_VECTOR(7 DOWNTO 0);
13  CONSTANT memory : mem_array := ("00000000",
14                                  "00000010",
15                                  "00000100",
16                                  "00001000",
17                                  "00010000",
18                                  "00100000",
19                                  "01000000",
20                                  "10000000");
21 BEGIN
22   data <= memory(TO_INTEGER(UNSIGNED(addr)));
23 END rom;
24 ----------------------------------------------------------------
```

ROM se modelira nizom konstantnih vrednosti (linije 13-20). Najpre se definiše novi tip podataka *mem_array* (linija 12) - 2D polje koje odgovara dimenzijama ROM-a. Tip *mem_array* se potom koristi u deklaraciji niza konstantnih vrednosti, *memory* (linija 13). Dati VHDL kôd opisuje ROM 8x8 (8 reči dužine 8 bita) sa sledećim vrednostima zapamćenim na adresama 0 do 7: 0, 2, 4, 8, 16, 32, 64 i 128 (linije 13-20). Telo arhitekture (linija 22) sadrži samo jednu konkurentnu naredbu dodele koja na izlazni port *data* prenosi vrednost koja je prisutnu u adresiranoj lokaciji konstantnog niza *memory*. Budući da je ulazni port *addr* tipa *std_logic_vector*, a da indeks niza mora biti tipa *integer*, neophodna je konverzija tipa: prvo se *addr* konvertuje u tip *unsigned*, a zatim u *integer*.

ROM zapravo nije ″prava″ memorija. Budući da se svaka binarna vrednost sa ulaznog porta preslikava u tačno definisanu binarnu vrednost na izlazu, ROM se može razumeti i kao kombinaciona komponenta sa višebitnim ulazom *addr* i višebitnim izlazom *data*. Zato se ROM ne realizuje pomoću flip-flopova, već se za njegovu implementaciju koriste logička kola, ili LUT blokovi (kod FPGA).

5. SEKVENCIJALNE NAREDBE

VHDL, iako u osnovi konkurentan jezik, pruža podršku i za sekvencijalno programiranje. Za razliku od konkurentnog kôda, gde postoji iluzija istovremenog (paralelnog) izvršavanja naredbi, u sekvencijalnom kôdu, baš kao i kod tradicionalnih programskih jezika, naredbe se izvršavaju po redosledu kako su napisane. Za kreiranje sekvencijalnog kôda u VHDL-u koristi se poseban skup naredbi, tzv. *sekvencijalne naredbe*. Konkurentne i sekvencijalne naredbe se ne mogu ″mešati″ u kôdu arhitekture, već se sekvencijalne sekcije kôda mogu pisati samo u okviru posebnih jezičkih konstrukcija: procesa, funkcija i procedura. Proces je, sam za sebe, kao celina, konkurentna naredba i kao takva može se kombinovati s drugim konkurentnim naredbama. Funkcije i procedure su potprogrami i mogu se pozivati iz drugih delova kôda, bilo konkurentnog, bilo sekvencijalnog.

Glavna namena sekvencijalnih naredbi jeste kreiranje apstraktnih algoritamskih modela digitalnih kola. Međutim, za razliku od konkurentnog kôd, korespondencija između sekvencijalnih naredbi i hardverskih komponenti nije uvek očigledna. Šta više, pojedine sekvencijalne naredbe se ne mogu sintetizovati. Zbog toga, da bi se obezbedila verna transformacija sekvencijalnog kôda u željenu hardversku konfiguraciju, korišćenje procesa, funkcija i procedura zahteva poštovanje određenih pravila pisanja kôda. Takođe, u sekvencijalnom kôdu, osim signala, dozvoljeno je korišćenje i varijabli. Za razliku od signala, varijable ne mogu biti deklarisane kao globalne, već samo kao lokalne promenljive u procesu, funkciji ili proceduri.

U ovoj glavi, naša pažnja biće usmerena na konstrukciju *process*. Preostale dve jezičke konstrukcije za kreiranje sekvencijalnog kôda (*function* i *procedure*) bitne su za projektovanje na sistemskom nivou i biće detaljno razmatrane u glavi 7.

5.1. PROCESS

Jezička konstrukcija *process* definiše oblast arhitekture u kojoj se naredbe izvršavaju na sekvencijalan način. Može se reći da kao što arhitektura predstavlja okvir za konkurentni, tako proces predstavlja okvir za sekvencijalni kôd. Proces je, kao celina, konkurentna naredba i kao takav može se u istoj arhitekturi kombinovati s drugim konkurentnim naredbama i drugim procesima. Sekvencijalni kôd se piše pomoću sekvencijalnih naredbi, koje se po sintaksi i semantici razlikuju od konkurentnih i mogu da egzistiraju samo unutar

procesa. Budući da se naredbe unutar procesa izvršavaju sekvencijalno, njihov redosled u procesu jeste od značaja.

Napomenimo da treba praviti jasnu razliku između sekvencijalnih naredbi i sekvencijalnih kola. Sekvencijalne naredbe su naredbe VHDL-a obuhvaćene procesom, dok su sekvencijalna kola digitalna kola sa internim stanjima (flip-flopovi, registri, konačni automati). Sekvencijalne naredbe se mogu koristiti za modeliranje kako sekvencijalnih tako i kombinacionih kola.

VHDL projekat može sadržati proizvoljan broj procesa, ali svi oni moraju biti smešteni u kôdu arhitekture, tj. ne mogu, kao zasebne projektne jedinice, biti deo paketa/biblioteke. Proces je sličan potprogramima iz programskih jezika po tome što sadrži sekvencijalne naredbe. Međutim, za razliku od potprograma, koji se pozivaju iz drugih delova programa, proces se ″samostalno″ aktivira, onda kada se za to steknu odgovarajući uslovi, odnosno kada se dese određeni događaji. Događaji koji pokreću (aktiviraju) proces su promene vrednosti signala na koje je proces senzitivan (osetljiv). U odnosu na to kako se definišu uslovi za aktiviranje, razlikujemo dve vrste procesa: proces sa listom senzitivnosti i proces sa *wait* naredbom.

5.1.1. Proces sa listom senzitivnosti

Proces sa listom senzitivnosti ima sledeću sintaksu (opcioni delovi sintakse obuhvaćeni su uglastim zagradama):

```
[labela:] PROCESS (lista senzitivnosti)
    [VARIABLE ime tip [opseg] [:= inicijalna_vrednost]]
BEGIN
    sekvencijalne naredbe;
END PROCESS [labela];
```

Proces počinje službenom rečju *process* posle koje sledi tzv. *lista senzitivnosti*. Lista senzitivnosti sadrži spisak signala na koje je proces senzitivan, tj. na čiju promenu proces reaguje. Promena vrednosti bilo kog signala iz liste senzitivnosti aktivira proces. Nakon aktiviranja, naredbe iz dela *sekvencijalne naredbe* izvršavaju se jedna za drugom, a nakon što se izvrši i poslednja naredba, proces se deaktivira (tj. suspenduje) i ostaje neaktivan sve do ponovnog aktiviranja, odnosno do nove promene nekog signala iz liste senzitivnosti. Varijable su opcione. Ako se koriste, moraju biti deklarisane u deklarativnom delu procesa (pre službene reči *begin*). Prilikom deklaracije, varijabli se može dodeliti inicijalna vrednost. Međutim, inicijalne vrednosti varijabli su podržane samo u simulaciji, ali ne i u sintezi VHDL kôda. Takođe opciono, proces može biti označen labelom, kojom započinje i kojom se završava konstrukcija *process*.

Kao što je već rečeno, procesi se koriste za modeliranje kako sekvencijalnih tako i kombinacionih kola. Za kombinaciona kola, lista senzitivnosti procesa mora da sadrži sve ulazne signale kola. Za sinhrona sekvencijalna kola, lista senzitivnosti sadrži signal takta i asinhrone ulaze (kao što je to npr. signal za asinhrono resetovanje), ali ne i sinhrone ulaze kola (kao što je to npr. ulaz *d* D flip-flopa). Proces sa listom senzitivnosti ne sme da sadrži sekvencijalnu naredbu *wait*.

Pr. 5-1 Jednostavan kombinacioni proces

Razmotrimo sledeći proces:

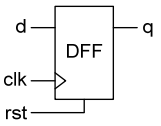
```
SIGNAL a, b, c : STD_LOGIC;
. . .
PROCESS(a, b, c)
BEGIN
  y <= a OR b OR c;
END PROCESS;
```

Proces se aktivira uvek kada se promeni bilo koji signala iz liste senzitivnosti: *a*, *b* ili *c*. Nakon aktiviranja, izvršava se naredba sadržana u telu procesa, a onda se proces deaktivira. Dati proces opisuje troulazno ILI kolo sa ulazima *a*, *b* i *c*, i izlazom *y*, a po funkciji i načinu rada identičan je konkurentnoj naredbi dodele:

```
y <= a OR b OR c;
```

Treba zapaziti da su mehanizmi pokretanja procesa i konkurentne naredbe dodele u osnovi isti. Kao što se konkurentna naredba dodele izvrši jedanput uvek kada se promeni vrednost bilo kog signala iz izraza s desne strane operatora dodele, tako se i proces izvrši jedanput uvek kad se promeni vrednost bilo kog signala iz njegove liste senzitivnosti. Razlika je u tome što je konkurentna naredba dodele osetljiva na promene **svih** svojih signala-operanada, dok proces ne mora biti osetljiv na sve signale koji se koriste u kôdu procesa. Neosetljivost procesa na neki signal se postiže izostavljanjem tog signala iz liste senzitivnosti. Ova osobina procesa se koristi za modeliranje sekvencijalnih digitalnih kola, kao što je to npr. D flip-flop iz sledećeg primera.

Pr. 5-2 D flip-flop sa asinhronim resetom



Sl. 5-1 D flip-flop sa asinhronim resetom.

D flip-flop je memorijsko kolo kapaciteta jednog bita i kao takvo predstavlja osnovni gradivni element složenijih sekvencijalnih kola. Varijanta D flip-flopa sa asinhronim resetom prikazana je na Sl. 5-1. Vrednost koja je upisana (memorisana) u flip-flopu dostupna je na izlazu *q*. Pod dejstvom rastuće ivice taktnog signala, *clk*, u flip-flop se upisuje trenutna vrednost ulaznog signala *d*. Ulaz za asinhrono resetovanje, *rst*, služi za forsirano (nezavisno od *clk*) postavljanje flip-flopa u stanje *q*=´0´.

U datom VHDL kôdu, funkcija D flip-flopa je opisana pomoću procesa koji sadrži samo jednu sekvencijalnu naredbu, *if*. Ako je *rst*=´1´, izlaz flip-flopa se postavlja na *q*=´0´ (linije 14-15) bez obzira na taktni signal *clk* (osobina asinhronog resetovanja). U suprotnom, za *rst*=´0´, a pod uslovom da se signal *clk* promenio sa ´0´ na ´1´, na izlaz flip-flopa se prenosi vrednost ulaza *d*, tj. *q*=*d* (linije 16-17). Signal *q* će zadržati dodeljenu vrednost i nakon deaktiviranja procesa (osobina memorisanja). U liniji 16, za detekciju promene signala *clk* koristi se atribut *event*. Iskaz *clk*'*event* vraća *true* uvek kada se vrednost signala *clk* promeni (bilo sa ´0´ na ´1´ ili sa ´1´ na ´0´). Da bi se izdvojila promena koja odgovara rastućoj ivici, uslov *clk*'*event* je dopunjen uslovom *clk*=´1´. Proces će se izvrši jedanput uvek kad se promeni bilo koji signal iz liste senzitivnosti, *clk* ili *rst*. Uočimo da lista senzitivnosti ne sadrži signal *d*. To znači da eventualne promene signala *d*, same po sebi, ne utiču na rad flip-flopa, već je vrednost ovog signala od značaja samo u trenucima promene takta sa ´0´ na ´1´ (što upravo odgovara načinu rada fizičkog D flip-flopa).

```
1  ----------------------------------------------------------------------
2  LIBRARY IEEE;
3  USE IEEE.STD_LOGIC_1164.ALL;
4  ----------------------------------------------------------------------
5  ENTITY dff IS
6     PORT (d,clk,rst : IN STD_LOGIC;
7           q : OUT STD_LOGIC);
8  END dff;
9  ----------------------------------------------------------------------
10 ARCHITECTURE arch_proc OF dff IS
11 BEGIN
12    PROCESS(clk,rst)
13    BEGIN
14       IF(rst='1') THEN
15          q <= '0';
16       ELSIF(clk'EVENT AND clk='1') THEN
17          q <= d;
18       END IF;
19    END PROCESS;
20 END arch_proc;
```

5.1.2. Proces sa *wait* naredbom

Kao što je već rečeno, proces koji ima listu senzitivnosti pokreće se promenom bilo kog signala iz liste senzitivnosti, a suspenduje nakon izvršenja svoje poslednje sekvencijalne naredbe. Drugim rečima, pri svakom pokretanju, izvrši se celokupan sekvencijalni kôd procesa, od početka do kraja. Naredba *wait* predstavlja alternativan način za suspenziju/aktiviranje procesa. Naredba *wait* (jedna ili više njih) se umeće u sam sekvencijalni kôd procesa. Šta više, ako se u procesu koristi *wait* naredba, lista senzitivnosti mora biti izostavljena. Nakon inicijalizacije sistema, proces sa *wait* naredbom, budući da nema listu senzitivnosti, automatski se startuje i izvršava sve do nailaska na prvu *wait* naredbu koja ga suspenduje. Uslov pod kojim proces nastavlja dalje zavisi od oblika *wait* naredbe. Postoje tri osnovna oblika *wait* naredbe:

```
WAIT UNTIL uslov;
WAIT ON signal1 [, signal2, …];
WAIT FOR vreme;
```

Kod varijante *wait until*, proces ostaje suspendovan sve dok uslov sadržan u naredbi ne postane tačan. Kod varijante *wait on*, uslov za nastavak izvršenja procesa je promena vrednosti bilo kog signala iz liste signala koja sledi posle službene reči *on*. Naredba *wait for* suspenduje proces na zadato vreme. Na primer, *"wait for 5 ns"*, suspenduje proces (tj. privremeno zaustavlja izvršenje procesa) na 5 ns.

Naredba *wait* omogućava kreiranje procesa složenog sekvencijalnog i "vremenskog" ponašanja i uglavnom se koristi za pisanje testbenča (npr. za generisanje pobudnih signala složenog talasnog oblika). U kôdu za sintezu dozvoljeno je koristiti samo nekoliko oblika *wait on* i *wait until* naredbi, s ograničenjem da u procesu može postojati samo jedna *wait* naredba i to na samom početku ili na samom kraju procesa. Naredba *wait for* se ne može koristiti u kôdu za sintezu. U nastavku su data tri primera procesa sa *wait* naredbom koji se mogu sintetizovati.

Pr. 5-3 ILI kolo

Proces koji sledi opisuje troulazno ILI kolo. U odnosu na proces iz Pr. 5-1, lista senzitivnosti zamenjena je *wait on* naredbom.

```
SIGNAL a, b, c : STD_LOGIC;
. . .
PROCESS
BEGIN
  WAIT ON a, b, c;
  y <= a OR b OR c;
END PROCESS;
```

Pr. 5-4 8-bitni registar sa sinhronim resetom

```
1  ------------------------------------------------------------------
2  SIGNAL clk, rst: STD_LOGIC;
3  SIGNAL out : STD_LOGIC_VECTOR(7 DOWNTO 0);
4  . . .
5  PROCESS
6  BEGIN
7    WAIT UNTIL(clk'EVENT AND clk='1');
8    IF(rst = '1') THEN
9      out <= "00000000";
10   ELSIF (clk'EVENT AND clk='1') THEN
11     out <= input;
12   END IF;
13 END PROCESS;
14 ------------------------------------------------------------------
```

Nailaskom na naredbu *wait until*, proces se suspenduje i čeka na ispunjenje uslova koji u ovom slučaju odgovara rastućoj ivici signala *clk* (linija 7).

Pr. 5-5 8-bitni registar sa asinhronim resetom

```
1  ------------------------------------------------------------------
2  SIGNAL clk, rst: STD_LOGIC;
3  SIGNAL out : STD_LOGIC_VECTOR(7 DOWNTO 0);
4  . . .
5  PROCESS
6  BEGIN
7   WAIT ON clk, rst;
8   IF(rst = '1') THEN
9     out <= "00000000";
10  ELSIF (clk'EVENT AND clk='1') THEN
11    out <= input;
12  END IF;
13 END PROCESS;
14 ------------------------------------------------------------------
```

Nakon što je suspendovan naredbom *wait on*, proces će nastaviti rad u momentu nastanka događaja na bilo kom signalu koji je naveden u ovoj naredbi. U datom primeru, svaka promena signala *clk* ili *rst* aktivira proces, koji se nakon izvršenja *if* naredbe vraća na početak i ponovo suspenduje naredbom *wait on*.

Jedina suštinska razlika između dva prethodna primera je u tome što kod prvog promena reset signala ne aktivira, a kod drugog aktivira proces. Kod registra sa sinhronim resetom (Pr. 5-4), proces reaguje jedino na taktni signal, tačnije na rastuću ivicu takta. To znači da eventualno aktiviranje signala *rst* nema uticaja na rad registra sve do pojave rastuće ivice takta. Kod registra sa asinhronim resetom (Pr. 5-5), kao i kod D fili-flopa iz Pr. 5-2, proces se aktivira ne samo taktnim već i reset signalom. Na taj način se postiže da je dejstvo signala *rst* trenutno i nezavisno od takta.

5.1.3. Signali u procesu

Signali se deklarišu u deklarativnom delu arhitekture, a vidljivi su, tj. mogu se koristiti, kako u konkurentnim sekcijama arhitekture, tako i unutar procesa koji su obuhvaćeni arhitekturom. Naredba kojom se signalu dodeljuje vrednost unutar procesa identičnog je oblika kao jednostavna konkurentna naredba dodele:

```
sig <= izraz;
```

Međutim, iako identičnog oblika, ponašanje naredbe dodele zavisi od toga da li se ona nalazi u konkurentnom ili sekvencijalnom kôdu. U konkurentnom kôdu, izvršenje naredbe dodele se inicira promenom bilo kog signala iz izraza sa desne strane znaka ″<=″. U procesu, naredba dodele se izvršava onda kada na nju ″dođe red″, ali se pri tom dodela izračunate vrednosti signalu **odlaže** do trenutka završetka procesa. Kaže se da za signale u procesu važi *odložena dodela*. To znači da u toku jednog izvršenja procesa, signal zadržava vrednost koju je imao u trenutku aktiviranja procesa (a koja potiče iz prethodnog izvršenja procesa), a novu vrednost dobija tek nakon deaktiviranja procesa. Ako se unutar procesa istom signalu više puta dodeli vrednost, samo će poslednja dodela imati vidljiv efekat. Ili, drugim rečima, budući da se signali ne ažuriraju sve do kraja procesa, oni u procesu nikada ne dobijaju međuvrednosti. Na primer, sledeća dva procesa su funkcionalno identični, a njihovom sintezom bio bi kreiran hardver koji sadrži samo jedno *xor* kolo:

```
SIGNAL a, b, c, d : STD_LOGIC;
. . .
PROCESS(a,b,c,d)
BEGIN
  y <= a AND b;
  y <= a OR c;
  y <= c XOR d;
END PROCESS;
```

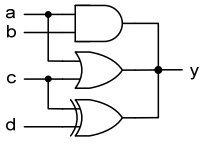
⇔

```
PROCESS(a,b,c,d)
BEGIN
  y <= c XOR d
END PROCESS;
```

Tri naredbe dodele iz prethodnog primera sintetizovale bi se na drugačiji način da su smeštene u konkurentnom kôdu (Sl. 5-2(a)). Iako višestruke dodele vrednosti istom signalu nisu zabranjene u konkurentnom kôdu, sintetizovano kolo ipak ne bi bilo korektno jer bi u njemu postojao kratak spoj (Sl. 5-2(b)). Kratkospajanje izlaza digitalnih kola može da izazove njihovo fizičko oštećenje u situacijama kad različita kola postavljaju različite izlazne vrednosti. U simulaciji, ovakva situacija se manifestuje pojavom nepoznate vrednosti ″X″ na signalu *y*.

```
SIGNAL a,b,c,d : STD_LOGIC;
. . .
-- sledece naredbe nisu u procesu
y <= a AND b;
y <= a OR c;
y <= c XOR d;
```

(a)



(b)

Sl. 5-2: Višestruka dodela u konkurentnom kôdu: (a) primer kôda; (b) sintetizovano kolo.

5.1.4. Varijable

Za razliku od signala, koji mogu biti deklarisani u paketu, entitetu ili arhitekturi i korišćeni kako u konkurentnom tako i u sekvencijalnom kôdu, varijable mogu biti deklarisane i korišćene isključivo u sekvencijalnim sekcijama VHDL kôda. Varijabla je uvek lokalna promenljiva i njena vrednost se ne može direktno preneti iz procesa u kome je deklarisana u neki drugi deo kôda. Ako je to ipak neophodno, vrednost varijable mora biti dodeljena signalu. S druge strane, nasuprot signalima za koje u procesu važi odložena dodela, ažuriranje varijable je trenutno (baš kao kod programskih jezika). Nakon izvršene naredbe dodele, varijabla trenutno dobija novu vrednost, koja se može koristiti već u sledećoj naredbi sekvencijalnog kôda. Sintaksa naredbe dodele za varijable je sledećeg oblika:

```
var := izraz;
```

Razmotrimo sledeći proces s varijablom:

```
1  SIGNAL a, b, y : STD_LOGIC;
2   ...
3  PROCESS(a,b)
4   VARIABLE v : STD_LOGIC;
5  BEGIN
6    v := '0';
7    v := v XOR a;
8    v := v XOR b;
9    y <= v;
10  PROCESS;
```

Uočimo sledeće pojedinosti. Varijable se uvek deklarišu u deklarativnom delu procesa (linija 4). Za varijable se koristite isti tipovi podataka kao i za signale (*std_logic*, *integer*, *bit*, ...). U naredbi dodele se koristi operator ″**:=**″, a ne ″<=″, da bi se naznačilo da se izračunata vrednost izraza dodeljuje varijabli, a ne signalu. Varijabla se ažurira neposredno nakon izvršene naredbe dodele. U datom primeru, vrednost varijable *v* nakon prve naredbe je ´0´ (linija 6), nakon druge *a* (linija 7) i nakon treće *a xor b* (linija 8). U poslednjoj sekvencijalnoj naredbi (linija 9), konačna vrednost varijable se dodeljuje signalu *y* i tako iznosi iz procesa.

Sekvencijalni kôd koji sadrži varijable, iako lakši za razumevanje i praćenje, zahteviniji je za sintezu u odnosu na sekvencijalni kôd koji barata isključivo signalima. Na primer, u prethodnom procesu, uloga varijabile *v* je da tri jednostavna izraza ″poveže″ u jedan složeniji. U domenu hardvera, takvu ulogu igraju električne veze koje npr. povezuju logička kola u složeniju mrežu. Međutim, fizički ne postoji električna veza koja se može koristiti za više od jednog povezivanja. Iz tog razloga, prilikom sinteze procesa koji sadrži varijable, neophodno je transformisati kôd procesa na način da se varijable koje se više puta koriste s leve strane znaka dodele (kao što je to slučaj sa varijablom *v*) zamene sa više novih varijabli:

```
1  PROCESS(a,b)
2   VARIABLE v0, v1, v2 : STD_LOGIC;
3  BEGIN
4    v0 := '0';
5    v1 := v0 XOR a;
6    v2 := v1 XOR b;
7    y <= v2;
8  END PROCESS;
```

Uočimo da se nakon transformacije (koju inače automatski sprovodi softver za sintezu) funkcija kôda nije promenila. Međutim, s obzirom na to što se sada svaka novouvedena varijabla, *v0*, *v1* i *v2*, koristi samo jedanput kao izlazna vrednost, svaka od njih se može interpretirati kao jedna električna veze u sintetizovanom hardveru (Sl. 5-3(a)).

Upravo zbog nepostojanja uvek jasne hardverske interpretacije, varijable treba koristiti u kôdu za sintezu samo u specifičnim slučajevima kad željenu funkciju nije moguće opisati pomoću signala. Poređenja radi, ispod je dat kôd prethodnog procesa u kome je varijabla *v* zamenjena signalom *s*.

```
1 SIGNAL a, b, s : STD_LOGIC;
2  ...
3 PROCESS(a,b,s)
4 BEGIN
5   s <= '0';
6   s <= s XOR a;
7   s <= s XOR b;
8   y <= s;
9 END PROCESS;
```

Uočimo sledeće razlike. Prvo, signal se ne može deklarisati u procesu, već samo u arhitekturi. Drugo, kad se opisuju kombinaciona kola, signali koji se koriste u procesu kao ulazi (tj. kao operandi u naredbama dodele) moraju biti navedeni u listi senzitivnosti procesa (što se u razmatranom primeru odnosi i na signal *s*). Treće i najbitnije, dva procesa (sa varijablom i sa signalom) nemaju istu funkciju. Naime, imajući u vidu osobinu odložene dodele, koja važi za signale u procesu, prethodni kôd se svodi na:

```
1 PROCESS(a,b,s)
2 BEGIN
3   s <= s XOR b;
4   y <= s;
5 END PROCESS;
```

Hardverska interpretacija ovog procesa je *xor* kolo s povratnom vezom (Sl. 5-3(b)).

(a) (b)

Sl. 5-3 Ilustracija razlike između varijabli i signala: (a) konceptualna implementacija procesa sa varijablama; (b) konceptualna implementacija identičnog procesa sa signalima.

5.2. IF

Osim naredbe dodele i naredbe *wait*, koja igra specifičnu ulogu u kontroli izvršenja procesa, skup sekvencijalnih naredbi čine još tri naredbe, *if*, *case* i *loop*, koje se koriste za kontrolu toka izvršenja sekvencijalnog kôda. Sintaksa naredbe *if* je sledećeg oblika:

```
IF uslov THEN
  sekvencijane_naredbe;
ELSIF uslov THEN
  sekvencijane_naredbe;
. . .
ELSE
  sekvencijane_naredbe;
END IF;
```

Naredba *if* počinje službenom rečju *if*, završava se rečima *end if* i sadrži dve opcione klauzule, *elsif* i *else*. Klauzula *elsif* se može ponoviti proizvoljan broj puta, a *else* samo jedanput. Grane *if* i *elsif* sadrže uslove koji predstavljaju logičke izraze (tj. izraze čiji rezultat može biti samo tačno ili netačno). Naredba *if* se izvršava tako što se redom ispituju uslovi sve dok se ne naiđe na granu s tačnim uslovom, a onda se redom izvršavaju sve sekvencijalne naredbe obuhvaćene tom granom. Ako ni jedan uslov nije tačan, izvršiće se *else* grana (ako postoji); ako ni jedan uslov nije tačan i *else* grana ne postoji, *if* naredba neće imati nikakvog efekta. U svakoj grani *if* naredbe može se naći proizvoljan broj sekvencijalnih naredbi, uključujući i druge *if* naredbe.

Pr. 5-6 IF-ELSIF-ELSE

If naredba iz ovog primera poredi signale/varijable *x* i *y* i zavisno od ishoda poređenja varijablama *v* i *w* dodeljuje tri različite vrednosti.

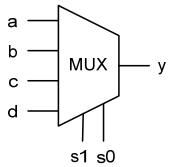
```
IF(x<y) THEN
   v := "11111111";
   w := '1';
ELSIF(x=y) THEN
   v:="11110000";
   w := '0';
ELSE
   v:=(OTHERS => '0');
   w := '1';
END IF;
```

Pr. 5-7 Multiplekser, binarni dekoder, binarni koder i prioritetni koder – realizacija pomoću naredbe *if*.

U primerima iz poglavlja 4.2, za opis multipleksera, binarnog dekodera, binarnog kodera i prioritetnog kodera, korišćena je naredba *when*. U ovom primeru, za opis navedenih digitalnih kola biće korišćen proces i sekvencijalna *if* naredba. Pošto se radi o kombinacionim kolima, svi ulazni signali kola moraju biti sadržani u listi senzitivnosti procesa. Napomenimo da se u svim opisima koji slede koriste signali tipa *std_logic/std_logic_vector*.

Multiplekser. Sledi arhitektura multipleksera 4-u-1. Promena bilo kog ulaznog signala, uključujući i dvobitni selekcioni ulaz, *s*, aktivira proces (linija 4). U procesu, naredba *if* na osnovu vrednosti selekcionog ulaza bira jedan ulazni signal, *a*, *b*, *c* ili *d*, čija će vrednost biti prosleđena na izlaz *y*.

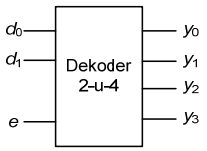
```
1  ----------------------------------------
2  ARCHITECTURE if_arch OF mux IS
3  BEGIN
4    PROCESS(a,b,c,d,s)
5      BEGIN
6        IF(s="00") THEN
7          y <= a;
8        ELSIF(s="01") THEN
9          y <= b;
10       ELSIF(s="10") THEN
11         y <= c;
12       ELSE
13         y <= d;
```



```
14        END IF;
15    END PROCESS;
16  END if_arch;
17  ----------------------------------------
```

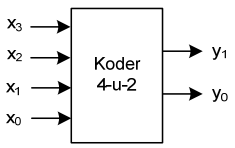
Binarni dekoder. Sledi arhitektura binarnog dekodera 2-u-4 sa ulaznom za dozvolu rada (v. Pr. 4-7). U *if* naredbi najpre se ispituje vrednost ulaza za dozvolu rada, *e*. Ako je dekoderu ″zabranjen″ rad (*e*=´0´), na izlaz *y* se postavljaju ″sve nule″ (linije 6-7). U suprotnom, ako je dekoderu ″omogućen″ rad (*e*=´1´), a u zavisnosti od vrednosti dvobitnog ulaza *d*, na odgovarajući izlaz y_i se postavlja ´1´ (linije 8-16).

```
1   ----------------------------------------
2   ARCHITECTURE if_arch OF dek2u4 IS
3   BEGIN
4   PROCESS(d, e)
5   BEGIN
6     IF(e = '0') THEN
7       y <= "0000";
8     ELSIF(d = "00") THEN
9       y <= "0001";
10    ELSIF(d = "01") THEN
11      y <= "0010";
12    ELSIF(d = "10") THEN
13      y <= "0100";
14    ELSE
15      y <= "1000";
16    END IF;
17  END PROCESS:
18  END if_arch;
19  ----------------------------------------
```



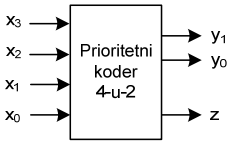
Binarni koder. U *if* naredbi, iz arhitekture binarnog kodera 4-u-2 koja sledi, vrednost ulaza *x* se redom poredi sa dozvoljenim binarnim kombinacijama. Ako je (*if*) *x* jednako ″0001″, na izlaz *y* se postavlja ″00″, inače ako je (*elsif*) *x* jednako ″0010″, *y* dobija vrednost ″01″, za *x* = ″0100″ (još jedno *elsif*), *y* je ″10″, inače (*else*) *y* dobija vrednost ″11″. Uočimo da je *else* grana obavezna zato što je njen zadatak da obuhvati ne samo poslednju dozvoljenu kombinaciju, ″1000″, već i sve preostale binarne i nebinarne četvorobitne vrednosti tipa *std_logic_vector*.

```
1   ----------------------------------------
2   ARCHITECTURE if_arch OF encoder IS
3   BEGIN
4   PROCESS(x)
5   BEGIN
6     IF(x = "0001") THEN
7       y <= "00";
8     ELSIF(x = "0010") THEN
9       y <= "01";
10    ELSIF(x = "0100") THEN
11      y <= "01";
12    ELSE
13      y <= "11";
14    END IF;
15  END PROCESS:
16  END if_arch;
17  ----------------------------------------
```



Prioritetni koder. Proces iz arhitekture prioritetnog kodera 4-u-2 sadrži dve *if* naredbe. Prva (linije 6-14) na izlaz *y* postavlja binarni kôd aktivnog ulaza najvišeg prioriteta, dok druga (linije 15-19) aktivira izlaz *z* ako je aktivan barem jedan ulaz.

```
1  ----------------------------------------
2  ARCHITECTURE if_arch OF pencoder IS
3  BEGIN
4  PROCESS(x)
5  BEGIN
6    IF(x(3) = '1') THEN
7      y <= "11";
8    ELSIF(x(2) = '1') THEN
9      y <= "10";
10   ELSIF(x(1) = '1') THEN
11     y <= "01";
12   ELSE
13     y <= "00";
14   END IF;
15   IF x = "0000" THEN
16     z <= '0';
17   ELSE
18     z <= '1';
19   END IF;
20 END PROCESS:
21 END if_arch;
22 ----------------------------------------
```



Naredba *if* donekle je slična konkurentnoj naredbi *when*. Primera radi, sledeća dva segmenta kôda su funkcionalno ekvivalentna:

```
PROCESS(…)
BEGIN
  IF uslov_1 THEN
    sig <= izraz_1;
  ELSIF uslov_2 THEN
    sig <= izraz_2;
  ELSIF uslov_3 THEN
    sig <= izraz_3;
  . . .
  ELSE
    sig <= izraz_n;
  END IF;
END PROCESS;
```

⇔

```
sig <= izraz_1 WHEN uslov_1 ELSE
       izraz_2 WHEN uslov_2 ELSE
       izraz_3 WHEN uslov_3 ELSE
       . . .
       izraz_n;
```

Međutim, ekvivalencija se može uspostaviti samo onda kada svaka grana *if* naredbe sadrži tačno jednu sekvencijalnu naredbu kojom se vrši dodela uvek istom signalu. Naredba *if* je opštija od naredbe *when*, budući da grane *if* naredbe mogu sadržati ne samo jednu, već proizvoljan broj sekvencijalnih naredbi. Pravilnim i ″disciplinovanim″ korišćenjem naredbe *if* kreira se deskriptivniji (čitljiviji) i često efikasniji (u smislu sinteze) kôd. Na primer, za razliku od naredbe *when*, ugnježdavanje *if* naredbi je dozvoljeno.

Pr. 5-8 ***If vs. when***

Pretpostavimo da želimo da odredimo maksimalnu vrednost tri binarna broja, *a*, *b* i *c*. Jedan od načina kako se to može postići jeste korišćenje ugnježdenih *if* naredbi:

```
1  SIGNAL a,b,c : UNSIGNED;
2  . . .
3  PROCESS(a,b,c)
4  BEGIN
5   IF (a>b) THEN
6    IF (a>c) THEN
7      max <= a; -- a>b i a>c
8    ELSE
9      max <= c; -- a>b i c>=a
10   END IF;
11  ELSE
12    IF (b>c) THEN
13     max <= b; -- b>=a i b>c
14    ELSE
15     max <= c; -- b>=a i c>=b
16    END IF;
17  END IF;
18 END PROCESS;
```

Identična funkcija se može opisati pomoću tri *when* naredbe:

```
SIGNAL ac_max, bc_max : UNSIGNED;
. . .
ac_max <= a WHEN (a > c) ELSE c;
bc_max <= b WHEN (b > c) ELSE c;
max <= ac_max WHEN (a > b) ELSE bc_max;
```

Takođe, istu funkciju možemo opisati i pomoću jedne *when* naredbe, ali pošto ova naredba ne dozvoljava ugnježdavanje, uslovi u *when* granama postaju komplikovaniji, a naredba teža za razumevanje:

```
max <= a WHEN ((a > b) AND (a > c)) ELSE
       c WHEN (a > b) ELSE
       b WHEN (b > c) ELSE
       c;
```

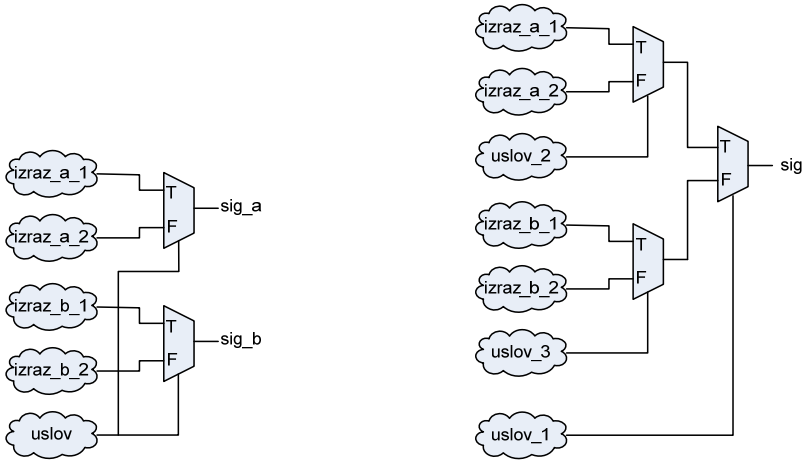
Konceptualna implementacija if naredbe. Kao što je već istaknuto, *if* naredba koja postavlja samo jedan izlazni signal ekvivalentna je po funkciji konkurentnoj *when* naredbi. To znači da konceptualni dijagram *if* naredbe, za ovaj jednostavan slučaj, možemo izvesti na isti način kao za *when* naredbu (v. 4.2). Na primer, konceptualni dijagram sledeće *if* naredbe identičan je konceptualnom dijagramu *when* naredbe sa Sl. 4-8(b).

```
IF uslov_1 THEN
  sig <= izraz_1;
ELSIF uslov_2 THEN
  sig <= izraz_2;
ELSIF uslov_3 THEN
  sig <= izraz_3;
ELSE
  sig <= izraz_4;
END IF;
```

Međutim, naredba *if* je opštija od konkurentne *when* naredbe, zato što dozvoljava da se u svakoj grani nađe ne samo jedna već proizvoljan broj sekvencijalnih naredbi. Na primer, razmotrimo sledeću *if* naredbu:

```
IF uslov_1 THEN
  sig_a <= izraz_a_1;
  sig_b <= izraz_b_1;
ELSE
  sig_a <= izraz_a_2;
  sig_b <= izraz_b_2;
END IF;
```

Pošto ova naredba postavlja dva izlazna signala, *sig_a* i *sig_b*, u odgovarajućem konceptualnom dijagramu postojaće dve zasebne prioritetne multiplekserske mreže iste strukture sa zajedničkim uslovom za selekciju (Sl. 5-4(a)).



(a) (b)

Sl. 5-4 Konceptualna implementacija *if* naredbe: (a) *if* naredba sa dva izlazna signala; (b) ugnježdavanje *if* naredbi.

Takođe, sekvencijalna *if* naredba dozvoljava ugnježdavanje, tj. pruža mogućnost da se u bilo kojoj grani *if* naredbe nađe druga *if* naredba, kao u sledećem kôdu:

```
IF uslov_1 THEN
  IF uslov_2 THEN
    sig <= izraz_a_1;
  ELSE
    sig <= izraz_a_2;
  END IF;
ELSE
  IF uslov_3 THEN
    sig <= izraz_b_1;
  ELSE
    sig <= izraz_b_2;
  END IF;
END IF;
```

Konceptualni dijagram prethodne *if* naredbe prikazan je na Sl. 5-4 (b). Izlazni multiplekser odgovara spoljašnjoj, a dva multipleksera iz sledećeg nivoa unutrašnjim *if* naredbama.

5.3. CASE

CASE je još jedna naredba namenjena isključivo za pisanje sekvencijalnog kôda. Njena sintaksa je sledećeg oblika:

```
CASE selekcioni_izraz IS
   WHEN vrednost => sekvencijalne_naredbe;
   WHEN vrednost => sekvencijalne_naredbe;
   . . .
   WHEN vrednost => sekvencijalne_naredbe;
   WHEN OTHERS => sekvencijalne_naredbe;
END CASE;
```

Naredba *case* je slična konkurentnoj naredbi *select*. U zavisnosti od vrednosti selekcionog izraza, za izvršenje se bira jedna od obuhvaćenih *when* grana. Kao i za konkurentnu naredbu *select*, sve moguće vrednosti selekcionog izraza moraju biti pokrivene *when* granama. Takođe, dozvoljeno je korišćenje grane *when others*, koja pokriva sve vrednosti selekcionog izraza koje nisu obuhvaćene prethodnim granama. Za razliku od naredbe *select*, naredba *case* dozvoljava da svaka grana, u delu *sekvencijalne_naredbe*, sadrži ne samo jednu, već proizvoljan broj naredbi (slično naredbi *if*). U granama gde ne postoji potreba za bilo kakvom akcijom, dozvoljeno je korišćenje službene reči *null*:

```
WHEN vrednost => NULL;
```

Pr. 5-9 Primer *case* naredbe sa *when others* granom

```
CASE ctrl IS
   WHEN "00" => x<=a; y<=b;
   WHEN "01" => x<=b; y<=a;
   WHEN OTHERS => x<="0000"; y<="ZZZZ";
END CASE;
```

Dati kôd se lako može protumačiti. Za *ctrl*=″00″, signal *x* dobija vrednost signala *a*, a signal *y* vrednost signala *b*, dok za *ctrl*=″01″ važi obrnuto, *x* dobija vrednost *b*, a *y* vrednost *a*. Ako važi *ctrl*=″10″ ili ″11″, *x* se postavlja na ″sve nule″, a *y* u stanje visoke impedanse.

Vrednost, ili vrednosti koje pokriva jedna *when* grana mogu se zadati na sledeća tri načina:

`WHEN vrednost`	Testiranje jedinstvene vrednosti
`WHEN vrednost_1 to vrednost_2`	Testiranje opsega vrednosti (važi samo za nabrojive tipove podataka)
`WHEN vrednost_1 | vrednost_2 | ...`	Testiranje skupa vrednosti (vrednost_1 ili vrednost_2 ili ...)

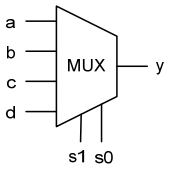
Pr. 5-10 Multiplekser, binarni dekoder, binarni koder i prioritetni koder – realizacija pomoću *case* naredbe

Radi ilustracije primene naredbe *case*, u ovom primeru predstavljeni su VHDL opisi četiri standardna kombinaciona kola: multiplekser, dekoder, koder i prioritetni koder. U svim ovim opisima, naredba *case* sadrži *when others* granu. Iz istih razloga kao za konkurentnu naredbu *select*, *when others* grana je neophodna uvek kada se koriste signali tipa *std_logic* da bi se obuhvatile i sve nebinarne vrednosti selekcionog izraza. Na primer, u opisu multipleksera 4-u-1, grana *when others* ne obuhvata samo poslednju binarnu kombinaciju

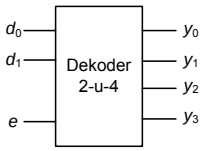
selekcionog signala, ″11″, već i sve moguće nebinarne kombinacije, kao što su to npr. ″LH″, ″-X″ itd.

Uočimo da se u *case* naredbi iz opisa binarnog kodera, za selekciju koristi signal *ed* koji objedinjuje ulazne signale *e* i *d*, na sličan način kao u Pr. 4-13 gde je za opis istog kola korišćena naredba *select*. Proces iz arhitekture binarnog dekodera iz ovog primera sadrži dve konkurentne naredbe, naredbu dodele (linija 5) i proces (linije 6-20). Takođe, treba napomenuti da naredba *case*, slično naredbi *select*, nije pogodna za opisivanje prioritetnih struktura, kao što je to slučaj sa prioritetnim koderom. Da je na primer umesto prioritetnog kodera 4-u-2 dat opis prioritetnog kodera 16-u-4, naredba *case* bi sadržala 16 umesto 4 *when* grane, sa ukupno 2^{14} = 16.384 16-bitnih vrednosti ulaznog signala u poslednjoj *when* grani!

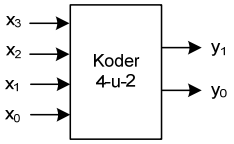
```
1  -- Multiplekser 4-u-1 -------------------
2  ARCHITECTURE case_arch OF mux IS
3  BEGIN
4   PROCESS(a,b,c,d,s)
5   BEGIN
6     CASE s IS
7       WHEN ″00″ =>
8         y <= a;
9       WHEN ″01″ =>
10        y <= b;
11      WHEN ″10″ =>
12        y <= c;
13      WHEN OTHERS
14        y <= d;
15    END CASE;
16  END PROCESS;
17 END case_arch;
18 ----------------------------------------
```



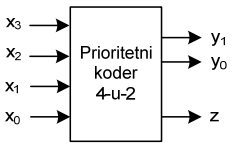

```
1  -- Binarni dekoder 2-u-4 ----------------
2  ARCHITECTURE case_arch OF dek2u4 IS
3   SIGNAL ed : STD_LOGIC;
4  BEGIN
5   ed <= e & d;
6   PROCESS(ed)
7   BEGIN
8     CASE ed IS
9       WHEN "100" =>
10        y <= "0001";
11      WHEN "101" =>
12        y <= "0010";
13      WHEN "110" =>
14        y <= "0100";
15      WHEN "111" =>
16        y <= "1000";
17      WHEN OTHERS
18        y <= "0000";
19    END CASE;
20  END PROCESS;
21 END case_arch;
22 ----------------------------------------
```



```
1  -- Binarni koder 4-u-2 ------------------
2  ARCHITECTURE case_arch OF encoder IS
3  BEGIN
4   PROCESS(x)
5   BEGIN
6     CASE x IS
7       WHEN "0001" =>
8         y <= "00";
9       WHEN "0010" =>
10        y <= "01";
11      WHEN "0100" =>
12        y <= "10";
13      WHEN OTHERS
14        y <= "11";
15    END CASE;
16  END PROCESS;
17 END case_arch;
18 ----------------------------------------
```




```
1  -- Prioritetni koder 4-u-2 -------------
2  ARCHITECTURE case_arch OF pencoder IS
3  BEGIN
4  PROCESS(x)
5  BEGIN
6   CASE x IS
7    WHEN "0001" =>
8      y <= "00";
9    WHEN "0010"|"0011" =>
10     y <= "01";
11   WHEN "0100"|"0101"|"0110"|"0111" =>
12     y <= "10";
13   WHEN OTHERS =>
14     y <= "11";
15   END CASE;
16 END PROCESS:
17 END case_arch;
18 ----------------------------------------
```



Konceptualna implementacija naredbe case. U naredbi *case*, vrednost selekcionog uslova se koristi kao ključ za izbor jednog od više blokova sekvencijalnih naredbi koji će biti izvršen. Naredba *case* kod koje se u svakoj grani postavlja samo jedan i to uvek isti izlazni signal može se vizuelno predstaviti u obliku apstraktnog multipleksera, na identičan način kao i konkurentna naredba *select*. Razmotrimo sledeću *case* naredbu koja je ekvivalentna *select* naredbi iz poglavlja 4.3:

```
CASE case_uslov IS
   WHEN v1 => sig <= izraz_1;
   WHEN v2 => sig <= izraz_2;
   WHEN OTHERS sig <= izraz_3;
END CASE;
```

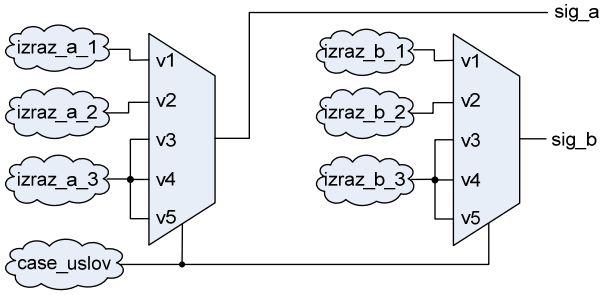
Pretpostavićemo da *case_uslov* može imati sledećih pet vrednosti: *v*1, *v*2, *v*3, *v*4 i *v*5. To znači da *when others* grana pokriva vrednosti *v*3, *v*4 i *v*5. Konceptualni dijagram ove *case*

naredbe je identičan dijagramu sa Sl. 4-13(b), koji je razvijen za ekvivalentnu *select* naredbu.

Prethodna šema se lako može uopštiti na naredbu *case* koja postavlja više od jednog izlaznog signala. Dovoljno je da se u konceptualnom dijagramu predvidi jedan poseban apstraktni multiplekser za svaki takav signal i da se selekcioni izraz *case* naredbe razvede do selekcionih ulaza svih multipleksera. Na primer, u sledećoj *case* naredbi postavljaju se dva izlazna signala:

```
CASE case_uslov IS
   WHEN v1 =>
     sig_a <= izraz_a_1;
     sig_b <= izraz_b_1;
   WHEN v2 =>
     sig_a <= izraz_a_2;
     sig_b <= izraz_b_2;
   WHEN OTHERS
     sig_a <= izraz_a_3;
     sig_b <= izraz_b_3;
END CASE;
```

Odgovarajući konceptualni dijagram prikazan je na Sl. 5-5. (I dalje važi pretpostavka da *case_uslov* može imati pet različitih vrednosti, *v*1, ..., *v*5.) Kao i naredba *if*, i naredba *case* je veoma opšta. U svakoj *when* grani naredbe *case* može se naći bilo koja validna sekcija sekvencijalnog kôda proizvoljne složenosti. U takvim slučajevima, crtanje konceptualnog dijagrama treba početi od apstraktnog multipleksera za *when* grane, a zatim iterativno razrađivati sekvencijalni kôd za svaku granu pojedinačno.



Sl. 5-5 Konceptualni dijagram CASE naredbe sa dva izlazna signala.

5.4. LOOP

Naredba *loop* se koristi za kreiranje petlji u sekvencijalnom kôdu. *Loop* naredba iz VHDL-a je moćnija (izražajnija) od odgovarajućih jezičkih konstrukcija iz mnogih programskih jezika. U VHDL-u postoje tri oblika *loop* naredbe: *for loop*, *while loop* i beskonačna petlja (ili samo *loop*), kao i dva mehanizma za prekid izvršenja petlje: *exit* i *next*. Naredba *loop* se uglavnom koristi za apstraktno modeliranje, dok je samo nekoliko njenih restriktivnih formi moguće sintetizovati.

FOR LOOP - petlja se ponavlja zadati broj puta. Sintaksa ovog oblika *loop* naredbe je:

```
[labela:] FOR indeks IN opseg LOOP
  sekvencijalne_naredbe;
END LOOP [labela];
```

Opseg definiše početnu i krajnju vrednost *indeksa*. Indeks se koristi za odbrojavanje iteracija petlje. Posle svake iteracije, indeks dobija sledeću vrednost iz datog opsega, sve dok ne dostigne krajnju vrednost opsega, što predstavlja i uslov za izlazak iz petlje. Indeks petlje se ne deklariše izvan petlje, već, direktno, u zaglavlju petlje. Pri tom indeks automatski preuzima tip navedenog opsega petlje. U telu petlje indesu nije dozvoljeno dodeljivati vrednosti. U kôdu namenjenom za sintezu, obe granice opsega indeksa *for loop* petlje moraju biti statičke (konstantne). Na primer, deklaracija oblika *"for i in 0 to n"*, gde je *n* varijabla, ne može se sintetizovati.

Pr. 5-11 FOR/LOOP

```
faktorijel := 1;
FOR broj IN 2 TO 10 LOOP
  faktorijel := faktorijel * broj;
END LOOP;
```

U ovom primeru, telo petlje se izvršava 9 puta (za *broj* = 2, 3, ..., 10). Krajnji rezultat je *faktorijel* = 10!.

WHILE LOOP **-** petlja se ponavlja sve dok se ne ispuni zadati uslov. Sintaksa ovog oblika *loop* naredbe je:

```
[labela:] WHILE uslov LOOP
   sekvencijalne naredbe;
END LOOP [labela];
```

Uslov je logički iskaz koji može biti tačan ili netačan. Petlja se ponavlja sve dok je uslov tačan. Kad uslov postane netačan, petlja se završava. Pretpostavka je da se *uslov* na neki način menja u telu petlje.

Pr. 5-12 WHILE LOOP

```
i:= 0; sum:= 10;
WHILE (i < 20) LOOP
   sum:= sum * 2;
   i:= i + 3;
END LOOP;
```

Telo petlje se ponavlja sve dok je vrednost varijable *i* manja od 20. Budući da pre ulaska u petlju varijabla *i* ima vrednost 0, a u svakoj iteraciji se povećava za 3, potrebno je 7 iteracija da bi *i* postalo jednako ili veće od 20.

LOOP - "beskonačna" petlja se realizuje kao naredba *loop* bez definisanog indeksa i uslova:

```
[labela:] LOOP
  sekvencijalne naredbe;
END LOOP [labela];
```

Kad se jednom uđe u beskonačnu petlju, iz nje se više ne može izaći, osim ukoliko u telu petlje ne postoji naredba *exit* koja definiše uslov izlaska, kao npr. u sledećem primeru:

```
sum:=1;
LOOP
  sum:=sum*3;
  EXIT WHEN sum>100;
END LOOP;
```

ili

```
LOOP
  WAIT on a,b;
  EXIT WHEN a=b;
END LOOP;
```

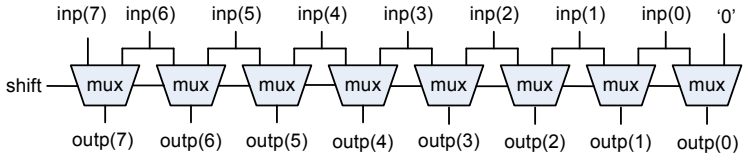
Uočimo da prethodna *loop* naredba ima isti efekat kao naredba:

```
WAIT UNTIL a=b;
```

Više detalja o naredbama *exit* i *next* biće dato u glavi 8. Napomenimo da od svih oblika *loop* naredbe, najveću primenu u sintezi nalazi oblik *for loop*. Varijante *while loop* i *loop* se mogu koristiti za apstraktno modeliranje ili za pisanje testbenča, ali ne i u kôdu za sintezu.

Pr. 5-13 Jednostavan *barel* pomerač

Barel pomerač (engl. *barrel shifter*) je kombinaciono kolo koje realizuje funkciju pomeranja ili rotiranja ulazne informacije za zadati broj bitskih pozicija. Na Sl. 5-6 je prikazana struktura jednog uprošćenog 8-bitnog barel pomerača. Za *shift*='0', ulazni 8-bitni podatak se prenosi, bez ikakve promene, na izlaz pomerača (*outp*(0)=*inp*(0), …, *outp*(7)=*inp*(7)), dok se za *shift*='1', na izlazu javlja ulazni podatak pomeren za jednu bitsku poziciju ulevo (*oupt*(0)=*0*, *oupt*(1)=*inp*(0) …, *oupt*(7)=*inp*(6)). Primetimo da se prilikom pomeranja ulazni bit *inp*(7) gubi, a da se na izlaznu poziciju *outp*(0), koja bi inače ostala upražnjena, postavlja ´0´.



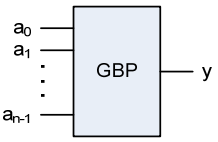
Sl. 5-6 8-bitni barel pomerač.

VHDL opis jednostavnog barel pomerača, koji je dat u nastavku, ilustruje upotrebu *for loop* naredbe. Za slučaj kada važi *shift*='1', najpre se na izlaz *outp*(0) postavi ´0´ (linija 18) , a zatim se u *for loop* petlji na svaki sledeći izlaz prenosi vrednost sa prethodnog ulaza (linije 19-21).

```
1  ----------------------------------------------------------------
2  LIBRARY IEEE;
3  USE IEEE.STD_LOGIC_1164.ALL;
4  ----------------------------------------------------------------
5  ENTITY barrel IS
6   PORT (inp : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
7         shift : IN STD_LOGIC;
8         outp : OUT STD_LOGIC_VECTOR(n-1 DOWNTO 0));
9  END barrel;
10 ----------------------------------------------------------------
11 ARCHITECTURE loop_arch OF barrel IS
12 BEGIN
13   PROCESS(inp,shift)
14   BEGIN
15     IF(shift='0') THEN
16       outp <= inp;
17     ELSE
```

```
18        outp(0)<='0';
19        FOR i IN 1 TO 7 LOOP
20          outp(i) <= inp(i-1);
21        END LOOP;
22      END IF;
23    END PROCESS;
24 END loop_arch;
```

Pr. 5-14 Generator bita parnosti



Sl. 5-7 Generator bita parnosti.

Na Sl. 5-7 je prikazan blok dijagram generatora bita parnosti (GBP). Ovo kolo poseduje višebitni ulaz *a* i jednobitni izlaz *y*. Na izlazu je *y*=′0′ za paran broj 1-ca na ulazu. U suprotnom, za neparan broj 1-ca na ulazu, na izlazu je *y*=′1′. Drugim rečima, izlazni bit svojom vrednošću dopunjava broj 1-ca sa ulaza do parnog broja. VHDL kôd koji sledi predstavlja opis 8-bitnog generatora bita parnosti.

Napomenimo da se parnost višebitnog podatka može odrediti samo jednom konkurentnom naredbom dodele u kojoj se operacija isključivo ILI (*xor*) primenjuje nad svim bitovima ulaznog podatka:

```
y <= a(0) XOR a(1) XOR … XOR a(7);
```

Isti efekat ima i kôd iz procesa (linije 15-19) u kome se koristi varijabla, *p*, za čuvanje međurezultata i naredba *for loop* za sukcesivnu primenu operacije *xor*. Neposredno pre ulaska u petlju, *p* dobija vrednost bita *a*(0) (linija 15). U prvoj iteraciji petlje vrednost varijable *p* postaje *p*=*a*(0) *xor a*(1), u drugoj *p*=*a*(0) *xor a*(1) *xor a*(2), i tako redom sve do poslednje, sedme iteracije kad *p* dobija konačnu vrednost *p*=*a*(0) *xor a*(1) *xor ... xor a*(7). Neposredno po izlasku iz petlje, u liniji 19, izračunata vrednost varijable *p* se prenosi na izlazni port *y*.

```
1  ------------------------------------------------------------------
2  LIBRARY IEEE;
3  USE IEEE.STD_LOGIC_1164.ALL;
4  ------------------------------------------------------------------
5  ENTITY pargen IS
6    PORT (a : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
7          y : OUT STD_LOGIC);
8  END pargen;
9  ------------------------------------------------------------------
10 ARCHITECTURE loop_arch OF pargen IS
11 BEGIN
12  PROCESS(a)
13    VARIABLE p : STD_LOGIC;
14  BEGIN
15    p := a(0);
16    FOR i IN 1 TO 7 LOOP
17      p := p XOR a(i);
```

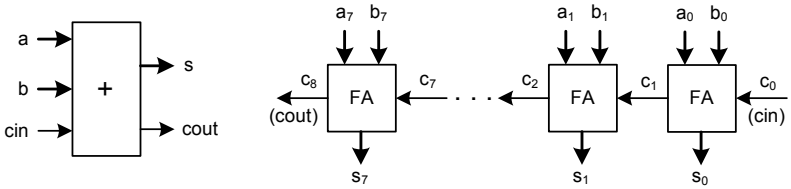
```
18    END LOOP;
19    y <= p;
20  END PROCESS;
21 END loop_arch;
```

Pr. 5-15 Sabirač sa rednim prenosom

Sl. 5-8 prikazuje blok dijagram i unutrašnju strukturu 8-bitnog sabirača sa rednim prenosom. Kolo poseduje dva 8-bitna ulaza, *a* i *b*, za sabirke, jednobitni ulaz za ulazni prenos, *cin*, 8-bitni izlaz za sumu, *s*, i jednobitni izlaz za izlazni prenos, *cout*. Sabirač čini osam redno povezanih potpunih sabirača (FA). Kao što je poznato iz digitalne elektronike, funkcija potpunog sabirača na *i*-toj poziciji može se izraziti jednačinama:

$$s_i = a_i \oplus b_i \oplus c_i$$

$$c_{i+1} = a_i \cdot b_i + a_i \cdot c_i + b_i \cdot c_i$$



Sl. 5-8 Sabirač sa rednim prenosom.

Sabirač sa rednim prenosom predstavlja primer iterativne strukture, koja se formira kaskadnim povezivanjem identičnih hardverskih blokova. Opis iterativnih struktura predstavlja jednu od najčešćih primena naredbe *for loop*. Kao što možemo videti u VHDL opisu sabirača sa rednim prenosom koji sledi, u svakoj iteraciji *for loop* petlje kreiraju se jednačine za jedan potpuni sabirač. Unutar procesa uvedena je vektorska varijabla *c* koja se koristi za ″sprezanje″ potpunih sabirača. Dužina vektora *c* je za jedan veća od dužine sabiraka (zbog izlaznog prenosa iz sabirača, *cout*). Pre ulaska u petlju, bitu najmanje težine vektora *c* dodeljuje se vrednost ulaznog prenosa sabirača, *c*(0):=*cin* (linija 17). U petlji (linije 18-21), za svaku bitsku poziciju se izračunavaju: bit sume, koji se dodeljuje izlaznom signalu *s*(*i*) i bit prenosa *c*(*i*+1), koji se kao ulazni prenos koristi u sledećoj iteraciji. Nakon izlaska iz petlje, vrednost bita najveće težine vektora *c, c*(8), prenosi se na izlazni port *cout*. Projektovanjem iterativnih struktura bavićemo se u glavi 8.

```
1  ----------------------------------------------------------------
2  LIBRARY IEEE;
3  USE IEEE.STD_LOGIC_1164.ALL;
4  ----------------------------------------------------------------
5  ENTITY sabirac IS
6   PORT (a,b : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
7         cin : IN STD_LOGIC;
8         s   : OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
9         cout: OUT STD_LOGIC);
10 END sabirac;
11 ----------------------------------------------------------------
12 ARCHITECTURE loop_arch OF sabirac IS
13 BEGIN
14   PROCESS(a,b,cin)
15     VARIABLE c : STD_LOGIC_VECTOR(8 DOWNTO 0);
```

```
16   BEGIN
17     c(0):=cin;
18     FOR i IN 0 TO 7 LOOP
19       s(i) <= a(i) XOR b(i) XOR c(i);
20       c(i+1):=(a(i) AND b(i))OR(a(i) AND c(i))OR(b(i) AND c(i));
21     END LOOP;
22     cout <= c(8);
23    END PROCESS;
24 END loop_arch;
25 ------------------------------------------------------------------
```

Konceptualna implementacija naredbe loop. Osnovni način za realizaciju *for loop* naredbe u hardveru zasniva se na *razmotavanju* petlje, odnosno na transformaciji polaznog kôda u kôd koji ne sadrži petlje. To znači da se hardver koji je opisan telom petlje ponavlja (replicira) za svaku iteraciju. Da bi razmotavanje bilo moguće, neophodno je da opseg indeksa petlje bude konstantan i poznat u vremenu kad se vrši sinteza (tj. mora biti statički). Na primer, kôd sabirača iz Pr. 5-15 u razmotanom obliku postaje:

```
c(0):=cin;
-- i = 0 ----------------------------------------------------
s(0) <= a(0) XOR b(0) XOR c(0);
c(1):=(a(0) AND b(0)) OR (a(0) AND c(0)) OR (b(0) AND c(0));
-- i = 1 ----------------------------------------------------
s(1) <= a(1) XOR b(1) XOR c(1);
c(2):=(a(1) AND b(1)) OR (a(1) AND c(1)) OR (b(1) AND c(1));
-- i = 2, 3 ... 6 -------------------------------------------

. . .

-- i = 7 ----------------------------------------------------
s(7) <= a(7) XOR b(7) XOR c(7);
c(8):=(a(7) AND b(7)) OR (a(7) AND c(7)) OR (b(7) AND c(7));
------------------------------------------------------------
cout <= c(8);
------------------------------------------------------------
```

5.5. Sekvencijalni kôd za kombinaciona kola

Sekvencijalni kôd se može koristiti kako za opisivanje sekvencijalnih, tako i kombinacionih kola. Razlika između ova dva tipa digitalnih kola je u tome što prva sadrže memorijske elemente, a druga ne. Da bi softver za sintezu prepoznao sekvencijalni kôd kao kôd za kombinaciono kolo, neophodno je da su ispunjena sledeća dva uslova:

- U listi senzitivnosti sadržani su svi signali koji se u procesu koriste kao ulazi.
- U kôdu procesa je definisan odziv za sve kombinacije vrednosti ulaznih signala, tako da softver za sintezu, analizom kôda, može da rekonstruiše potpunu tabelu istinitosti.

Prvi uslov nije obavezan za sintezu. Tipično, ako je lista senzitivnosti nepotpuna, a pri tom je drugi uslov ispunjen, kompajler će izdati odgovarajuće upozorenje, ali će sintetizovano kolo ipak biti korektno. Međutim, problem će se javiti u simulaciji takvog kôda, jer proces neće reagovati na ulazne signale kojih nema u listi senzitivnosti. Na primer, pretpostavimo da su signali *b* i *c* greškom izostavljeni iz liste senzitivnosti sledećeg procesa koji opisuje troulazno ILI kolo:

```
PROCESS(a)
BEGIN
  y <= a OR b OR c;
END PROCESS;
```

Kad se promeni signal *a*, proces se aktivira i reaguje na očekivan način. Međutim, proces ne reaguje na promene signala *b* i *c*, već ostaje suspendovan. To znači da izlaz *y*, bez obzira na novu vrednost signala *b* ili *c*, zadržava svoju staru vrednost. Drugim rečima, u simulaciji, proces se ponaša kao flip-flop sa okidanjem na obe ivice signala *a* i ulazom za podatak koji se formira pomoću troulaznog ILI kola - čak i da je to bila stvarna namera projektanta, takvo kolo se ne može sintetizovati. Međutim, budući da je u kôdu procesa definisan odziv na svaku moguću kombinaciju vrednosti ulaznih signala, *a, b* i *c*, većina alata za sintezu tretiraće dati proces kao opis kombinacionog ILI kola, uz izdavanje upozorenja o tome da je lista senzitivnosti nepotpuna. Bez obzira na to, dobra je praksa da se u listu senzitivnosti procesa uvrste svi ulazni signali kombinacionog kola koje se opisuje.

Drugi uslov čini suštinsku razliku između kombinacionih i sekvencijalnih kola i mora biti ispunjen da bi se sintetizovalo korektno kolo. Nedefinisan odziv procesa na neku ulaznu kombinaciju znači da proces pri takvoj pobudi zadržava (memoriše) zatečen (prethodni) izlaz! Da bi se omogućilo memorisanje, u sintetizovano kolo biće uvršteni memorijski elementi ili će osobina memorisanje biti ostvarena uz pomoć povratnih veza. Greške ovog tipa se tipično javljaju kod nepotpunih *if* i *case* naredbi.

Shodno sintaksi VHDL-a, u *if* naredbi *else* grana je opciona. Takođe, istom izlaznom signalu ne mora biti dodeljena vrednost u svakoj grani *if* naredbe. Na primer, sledeći kôd predstavlja neuspešan pokušaj kreiranja kompaktnog opisa komparatora koji treba da uporedi ulaze *a* i *b* i postavi izlaz *eq* na ´1´ ako su *a* i *b* jednaki:

```
PROCESS(a,b)
BEGIN
  IF(a=b) THEN
    eq <= '1';
  END IF;
END PROCESS;
```

Kôd je sintaksno ispravan. Kad je *a* jednako *b*, *eq* dobija vrednost ´1´. Međutim, ako se *a* i *b* razlikuju, a budući da *else* grana ne postoji, nikakva akcija neće biti preduzeta. Shodno semantici VHDL-a, signal *eq* se neće promeniti, odnosno zadržaće svoju trenutnu vrednost. Drugim rečima, prethodna nepotpuna *if* naredba je ekvivalentna sledećoj potpunoj *if* naredbi:

```
IF(a=b) THEN
  eq <= '1';
ELSE
  eq <= eq;
END IF;
```

Sada je jasno da u kôdu postoji zatvorena petlja, zbog koje se u kolu kreira memorija, što svakako nije željeno ponašanje komparatora. Ispravan kôd je oblika:

```
IF(a=b) THEN
  eq <= '1';
ELSE
  eq <= '0';
END IF;
```

Međutim, kompletnost *if* naredbe, sama po sebi, nije garancija da će biti sintetizovano kombinaciono kolo. Razmotrimo sledeći kôd:

```
PROCESS(a, b)
BEGIN
  IF(a > b) THEN
    gt <= '1';
  ELSIF(a = b) THEN
    eq <= '1';
  ELSE
    lt <= '1';
  END IF;
END PROCESS;
```

Namera je da se procesom opiše komparator sa tri izlaza (v. Pr. 4-19). Kad je *a* veće od *b*, aktivan je izlaz *gt*, za *a* jednako *b*, aktivan je *eq*, a kad je *a* manje od *b*, aktivan je izlaz *lt*. Međutim, kôd procesa definiše samo kada su izlazni signali aktivni, ali ne i kada su neaktivni. Na primer, ako važi $a > b$, izvršiće se prva grana *if* naredbe u kojoj će se samo izlaz *gt* postaviti na ´1´. Pošto signalima *eq* i *lt* proces nije dodelio ni ´1´ ni ´0´, oni će zadržati svoje ranije vrednosti. Slična situacija se javlja i u preostala dva slučaja ($a < b$ i $a = b$). Zbog potrebe pamćenja prethodnih vrednosti izlaznih signala, u sintetizovano kolo će biti ugrađena tri memorijska elementa, po jedan za svaki izlaz. Ispravno napisan kôd treba da ima oblik:

```
PROCESS(a, b)
BEGIN
  IF(a > b) THEN
    gt <= '1';
    eq <= '0';
    lt <= '0';
  ELSIF (a = b) THEN
    gt <= '0';
    eq <= '1';
    lt <= '0';
  ELSE
    gt <= '0';
    eq <= '0';
    lt <= '1';
  END IF;
END PROCESS;
```

Sada nema potrebe za ″pamćenjem″ vrednosti izlaznih signala, jer se u procesu, za bilo koji ishod poređenja, svakom izlaznom signalu dodeljuje odgovarajuća vrednost, ´0´ ili ´1´. Kôd ovog procesa se može učiniti kompaktnijim ako se izlazim signalima dodele podrazumevane vrednosti na samom početku procesa:

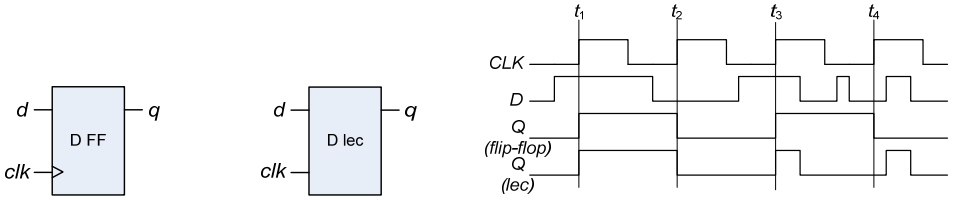
```
PROCESS(a, b)
BEGIN
  gt <= '0';
  eq <= '0';
  lt <= '0';
  IF(a > b) THEN
    gt <= '1';
  ELSIF (a = b) THEN
    eq <= '1';
```

```
  ELSE
    lt <= '1';
  END IF;
END PROCESS;
```

Kao što je već ranije napomenuto, u situacijama kad se unutar procesa istom signalu dodeljuje vrednost više puta, efekat će imati samo poslednja dodela. U konkretnom primeru to znači da ako se nekom signalu ne dodeli vrednost u *if* naredbi, tada će taj signal zadržati vrednost koja mu je dodeljena na početku procesa. U suprotnom, ako se signalu dodeli vrednosti u *if* naredbi, to će ujedno biti i konačna vrednost tog signala. Dodela podrazumevanih vrednosti izlaznim signalima na početku procesa predstavlja dobru projektantsku praksu, jer se time izbegava mogućnost da se pod izvesnim uslovima signalu uopšte i ne dodeli vrednost i na taj način kreira neželjena memorija.

5.6. Leč kola i flip-flopovi

Leč kola i flip-flopovi su dva osnovna tipa elementarnih memorijskih kola. Razmotrićemo D leč i D flip-flop. Grafički simboli D flip-flopa i D leča prikazani su na Sl. 5-9(a) i (b) (uočimo različito označavanje ulaza za takt). Ključna osobina flip-flopova je reakcija na ivicu, a ne na nivo taktnog signala, što je naznačeno trouglom na ulazu za takt. Promena stanja flip-flopa (tj. promena memorisane vrednosti) inicirana je ivicom taktnog signala. Zbog toga, flip-flop nikad nije transparentan. Drugim rečima, ulaz *d* nikad direktno ne utiče na izlaz *q*, već samo definiše stanje u koje će flip-flop biti postavljen pod dejstvom aktivne ivice taktnog impulsa. S druge strane, D leč reaguje na nivo taktnog signala. Za sve vreme dok je taktni signal aktivan (*clk*=′1′), D leč je transparentan jer se svaka promena vrednosti ulaza *d* direktno prenosi na izlaz *q* (tj. važi *q* = *d*). U trenutku deaktiviranja takta (promena signala *clk* sa ′1′ na ′0′), leč se ″zaključava″ i pamti svoje tekuće stanje. Eventualne promene ulaza *d* ne utiču na stanje leča za vreme dok je *clk*=′0′. Opisana razlika u ponašanju D leča i D flip-flopa ilustrovana je vremenskim dijagramom sa Sl. 5-9(c).



Sl. 5-9 Razlike između flip-flopova i leč kola: (a) D flip-flop; (b) D leč; (c) vremenski dijagram.

Leč kola i flip-flopovi nalaze primenu u konstrukciji složenijih sekvencijalnih kola i sistema, kao što su memorijske strukture, registarske komponente i konačni automati. Pri tom flip-flopovi, iako složeniji, koriste se mnogo češće. Razlog za to su karakteristike flip-flopova, koje značajno pojednostavljuju projektovanje složenijih sekvencijalnih sistema. Kao prvo, varijacije i gličevi (kratkotrajne promene) ulaznog signala (*d*) koje se dešavaju između dve rastuće ivice taktnog signala ne utiču na stanje flip-flopa. Drugo, problem trke, koji se može javiti u sistemima s povratnom petljom koji sadrže leč kola, ne postoji ako se koriste flip-flopovi.

U nastavku su dati VHDL opisi D flip-flopa i D leča. Kôd za D flip-flop sličan je kôdu iz Pr. 5-2, s tom razlikom što je sada izostavljen signal za asinhrono resetovanje. Iako kôd ne

zahteva neko posebno objašnjenje, obratimo pažnju da lista senzitivnosti procesa sadrži samo signal takta. U kôdu za D leč, za definisanje vrednosti izlaza *q* koristi se naredba *if* (linije 14-16). Kada je *clk*=´1´, vrednost ulaza *d* se prenosi na izlaz *q*. Međutim, kôd ne navodi koju bi vrednost trebalo da dobije *q* onda kada *clk* nije ´1´. Zbog toga, *q* zadržava svoju tekuću vrednost za sve vreme dok važi *clk*=´0´ - što upravo odgovara načinu rada D leča. Uočimo da lista senzitivnosti procesa koji opisuje leč sadrži oba ulazna signala, *d* i *clk*, pošto oba ova signala neposredno utiču na vrednost izlaza *q*.

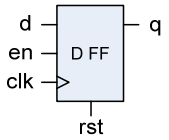
```
1  ---D flip-flop ----------------------------------------------------
2  LIBRARY IEEE;
3  USE IEEE.STD_LOGIC_1164.ALL;
4  ----------------------------------------------------------------------
5  ENTITY dff IS
6    PORT (d,clk : IN STD_LOGIC;
7          q : OUT STD_LOGIC);
8  END dff;
9  ----------------------------------------------------------------------
10 ARCHITECTURE flip_flop OF dff IS
11 BEGIN
12    PROCESS(clk)
13    BEGIN
14       IF(clk'EVENT AND clk='1') THEN
15          q <= d;
16       END IF;
17    END PROCESS;
18 END flip_flop;
```

```
1  ---D lec ----------------------------------------------------------
2  LIBRARY IEEE;
3  USE IEEE.STD_LOGIC_1164.ALL;
4  ----------------------------------------------------------------------
5  ENTITY dlec IS
6    PORT (d,clk : IN STD_LOGIC;
7          q : OUT STD_LOGIC);
8  END dlec;
9  ----------------------------------------------------------------------
10 ARCHITECTURE lec OF dlec IS
11 BEGIN
12    PROCESS(d,clk)
13    BEGIN
14      IF clk = '1' THEN
15        q <= d;
16      END IF;
17    END PROCESS;
18 END lec;
19 ----------------------------------------------------------------------
```

Pr. 5-16 D flip-flop sa dozvolom

U ovom primeru predstavljen je VHDL opis D flip-flopa sa dodatnim signalom dozvole, *en* (engl. *enable*) (Sl. 5-10(a)). Funkcionalna tabela ovog flip-flopa data je na Sl. 5-10(b). Uočimo da signal dozvole ima efekta samo u trenucima delovanja rastuće ivice takta. To znači da je osim signala *d* i ovaj signal sinhronizovan sa taktnim signalom i zato, kao i *d*, nije naveden u listi senzitivnosti procesa. Dakle, u trenutku rastuće ivice taktnog signala,

flip-flop ispituje signal dozvole i ako važi *en*=´1´ (upis u flip-flop je dozvoljen) vrednost izlaza *q* postaje jednaka trenutnoj vrednosti ulaza *d*. U suprotnom slučaju, za *en*=´0´, upis u flip-flop nije dozvoljen i izlaz *q* ostaje nepromenjen. Za razliku od signala dozvole, signal resetovanja, *rst*, je asinhronog tipa i zbog toga je naveden u listi senzitivnosti procesa.



(a)

rst	clk	en	q*
1	x	x	0
0	0	x	q
0	1	x	q
0	↓	0	q
0	↓	1	d

(b)

Sl. 5-10 D FF sa dozvolom: (a) grafički simbol; (b) funkcionalna tabela.

```
---D FF sa dozvolom ------------------------------------------
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
-----------------------------------------------------------------
ENTITY dffe IS
  PORT (d,clk,en,rst : IN STD_LOGIC;
        q : OUT STD_LOGIC);
END dffe;
-----------------------------------------------------------------
ARCHITECTURE dffe_v1 OF dffe IS
BEGIN
   PROCESS(clk,rst)
   BEGIN
     IF(rst = '1') THEN
       q <= '0';
     ELSIF(clk'EVENT AND clk='1') THEN
       IF(en = '1') THEN
         q <= d;
       END IF;
     END IF;
   END PROCESS;
END dffe_v1;
-----------------------------------------------------------------
```

Ispod je dat alternativni VHDL opis arhitekture istog D flip-flopa. Kôd sadrži tri sekcije. Prva sekcija (D FF) je proces koji opisuje regularni D flip-flop sa asinhronim resetom. Ulaz u flip-flop je signal *q_next*, a izlaz signal *q_reg*. Druga sekcija (logika sledećeg stanja) sadrži konkurentnu naredbu *when* koja u zavisnosti od signala dozvole reguliše vrednost internog signala *q_next* (ulaz u flip-flop). Treća sekcija (izlazna logika) prosleđuje vrednost internog signala *q_reg* na izlazni port *q*. U ovoj realizaciji, efekat zabrane upisa u D flip-flop se postiže posredno, upisom u flip-flop vrednosti koja je već sadržana u flip-flopu (jer za *en*=´0´ važi *q_next* = *q_reg*) .

```
--D FF sa dozvolom -------------------------------------------
ARCHITECTURE dffe_v2 OF dffe IS
  SIGNAL q_reg, q_next : STD_LOGIC;
BEGIN
-- D FF ---------------------------------------------------------
 PROCESS(clk,rst)
 BEGIN
   IF(rst = '1') THEN
```

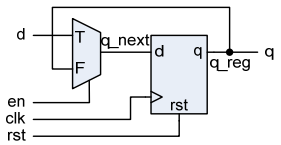
```
9       q_reg <= '0';
10    ELSIF(clk'EVENT AND clk='1') THEN
11      q_reg <= q_next;
12    END IF;
13  END PROCESS;
14 -- Logika sledeceg stanja ----------------------------------------
15  q_next <= d WHEN en = '1' ELSE
16            q_reg;
17 -- Izlazna logika -----------------------------------------------
18  q <= q_reg;
19 END dffe_v2;
20 ------------------------------------------------------------------
```

Konceptualni dijagram prethodnog VHDL opisa prikazan je na Sl. 5-11. Prilikom crtanja konceptualnih dijagrama za sekvencijalna kola, prvo treba uočiti signale (ili varijable) koji se sintetišu kao memorijski elementi. U datom primeru, to je slučaj sa signalom *q_reg*, koji se sintetiše u D flip-flop sa asinhronim resetom. Apstraktni multiplekser odgovara *when* naredbi (linije 15-16).

Treba istaći da bi se strukturno identični konceptualni dijagram dobio i analizom prvog od dva VHDL opisa iz ovog primera - u kome je celokupna funkcionalnost D flip-flopa sa dozvolom opisana jednim procesom. Signal iz ovog opisa koji se sintetiše u D flip-flop sa asinhronim resetom je signal *q* (spoljašnja *if* naredba, linije 14-20). Ugnježdena *if* naredba (linije 17-19) odgovara apstraktnom multiplekseru. Treba obratiti pažnju da je ova *if* naredba nepotpuna jer ne sadrži *else* granu. Međutim, *else* grana se podrazumeva, a naredba je ekvivalentna sledećoj *if* naredbi sa *else* granom:

```
IF(en = '1') THEN
  q <= d;
ELSE
  q <= q;
END IF;
```

što objašnjava vezu apstraktnog multipleksera i flip-flopa u konceptualnom dijagramu sa Sl. 5-11.

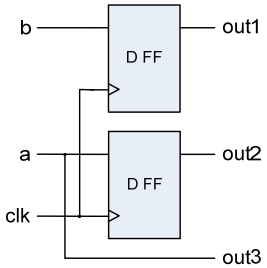


Sl. 5-11 Konceptualni dijagram D FF sa dozvolom i asinhronim resetom.

Sinteza signala u flip-flop. U zavisnosti od konteksta, signali kojima se dodeljuje vrednost u procesu mogu se sintetizovati bilo u električnu vezu, bilo u flip-flop (ili registar). Signal se sintetiše u flip-flop ako je dodela vrednosti tom signalu uslovljena tranzicijom (promenom) nekog drugog signala. Takve naredbe dodele se nazivaju *sinhronim naredbama dodele* i mogu se javiti samo u sekvencijalnom kôdu, obično posle konstrukcija tipa: *"if sig'event …"* ili *"wait until …"*. Na primer, VHDL kôd sa Sl. 5-12(a) sadrži dve sinhrone naredbe dodele (u linijama 4 i 5), jer su dodele vrednosti signalima *out1* i *out2* sinhronizovane s rastućom ivicom signala *clk*. Zbog toga u sintetizovanom kolu (Sl. 5-12(b)) postoje dva flip-flopa: jedan za signal *out1*, a drugi za signal *out2*. S druge strane, dodela vrednosti signalu *out3* nije sinhronizovana sa signalom *clk* zbog čega se ovaj signal sintetiše u električnu vezu koja povezuje ulaz *a* i izlaz *out3*.

```
1 PROCESS(clk,a)
2 BEGIN
3 IF(clk'EVENT AND clk='1')THEN
4   out1 <= b;
5   out2 <= a;
6 END IF;
7  out3 <= a;
8 END PROCESS;
```

(a)



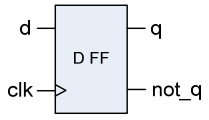
(b)

Sl. 5-12 Primer sinteze flip-flopova: (a) VHDL kôd; (b) sintetizovano kolo.

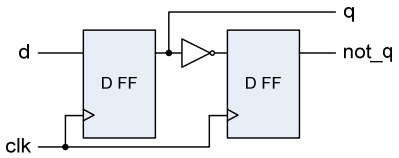
Pr. 5-17 D flip-flop sa komplementarnim izlazima

Razmotrimo D flip-flop sa Sl. 5-13(a) koji, za razliku od ranije razmatranih flip-flopova, poseduje dodatni izlaz *not_q* na kojem se javlja komplement tekućeg stanja flip-flopa. U nastavku su data tri VHDL opisa ovog flip-flopa, od kojih je ispravan samo treći.

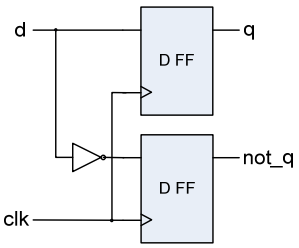
Tri rešenja se razlikuju po mestu gde se u kôdu nalazi naredba dodele kojom se postavlja signal *not_q*. U prvom rešenju, ova naredba je **u** procesu neposredno posle naredbe *q_reg* <= *d* (linije 16 i 17). Međutim, s obzirom na odloženu dodelu koja važi za signale u procesu, vrednost signala *q_reg* koja se u liniji 17 komplementira i dodeljuje signalu *not_q*, nije ona koja je dodeljena signalu *q_reg* u liniji 16, već stara, tj. ona koju je ovaj signal zadržao iz prethodnog ciklusa rada procesa. Na taj način, izlaz *not_q* dobija pogrešnu vrednost, koja u odnosu na ispravnu kasni za jedan taktni ciklus, kao što se to može videti na vremenskom dijagramu sa Sl. 5-14. Primetimo da talasni oblici izlaznih signala *q* i *not_q* počinju isprekidanim linijama. (Isprekidana linija označava nepoznatu vrednost - ″X″). Razlog za to je što ova dva signala nisu inicijalizovani prilikom deklaracije, tako da definisanu vrednost (´0´ ili ´1´) dobijaju tek s prvim upisom, koji je sinhronizovan s rastućom ivicom signala *clk*. Struktura kola sintetizovanog na osnovu rešenja 1 prikazana je na Sl. 5-13(b). Dva flip-flopa proističu iz dve sinhrone naredbe dodele (linije 16 i 17).



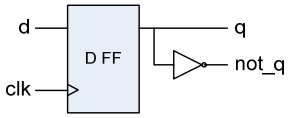
(a)



(b)



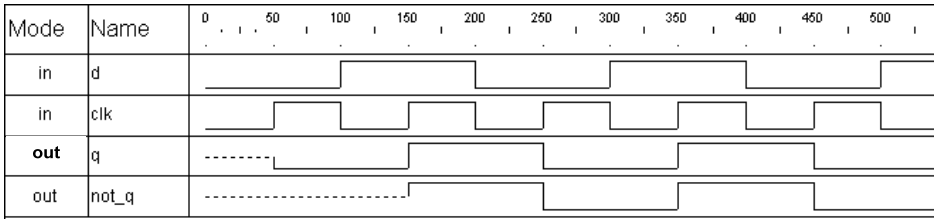
(c)



(d)

Sl. 5-13 D flip-flop sa komplementarnim izlazima: (a) grafički simbol; kolo sintetizovano na osnovu: (b) rešenja 1; (c) rešenja 2, (d) rešenja 3.

```
1  ----------------------------------------------------------------
2  LIBRARY IEEE;
3  USE IEEE.STD_LOGIC_1164.ALL;
4  ----------------------------------------------------------------
5  ENTITY dff IS
6    PORT (d, clk : IN STD_LOGIC;
7          q, not_q : OUT STD_LOGIC);
8  END dff;
9  --- Resenje 1: pogrešno ------------------------------------------
10 ARCHITECTURE neispravna OF dff IS
11  SIGNAL q_reg : STD_LOGIC;
12 BEGIN
13   PROCESS(clk)
14   BEGIN
15     IF(clk'EVENT AND clk='1') THEN
16        q_reg <= d;              -- generise flip-flop
17        not_q <= NOT q_reg;      -- generise flip-flop
18     END IF;
19   END PROCESS;
20   q <= q_reg;
21 END neispravna;
22 --- Resenje 2: ispravno, ali dva flip-flopa ----------------------
23 ARCHITECTURE dva_flip_flopa OF dff IS
24 BEGIN
25   PROCESS(clk)
26   BEGIN
27     IF(clk'EVENT AND clk='1') THEN
28        q <= d;                  -- generise flip-flop
29        not_q <= NOT d;          -- generise flip-flop
30     END IF;
31   END PROCESS;
32 END dva_flip_flopa;
33 --- Resenje 3: ispravno, jedan flip-flop -------------------------
34 ARCHITECTURE ispravna OF dff IS
35  SIGNAL q_reg : STD_LOGIC;
36 BEGIN
37    PROCESS(clk)
38    BEGIN
39      IF(clk'EVENT AND clk='1') THEN
40         q_reg <= d;                    -- generise flip-flop
41      END IF;
42    END PROCESS;
43    q <= q_reg;
44    not_q <= NOT q_reg;                 -- generise invertor
45 END ispravna;
```



Sl. 5-14 Rezultati simulacije rešenja_1 iz Pr. 5-17.

U rešenju 2, signal *not_q* u procesu dobija invertovanu vrednost ulaza *d* (linija 29), a ne invertovanu vrednost signala *q_reg*. Zbog toga je ovo rešenje funkcionalno ispravno. Međutim, pošto proces i dalje sadrži dve sinhrone naredbe dodele (linije 28 i 29) sintetizovano kolo će sadržati dva flip-flopa (Sl. 5-13(c)).

Konačno, rešenje 3 je i funkcionalno ispravno i sadrži samo jedan flip-flop. U ovom opisu, naredba dodele *not_q* <= *q_reg* je izmeštena van procesa (linija 44). Pošto više nije sinhronizovana sa taktnim signalom *clk*, ova naredba se ne sintetiše u flip-flop već u invertor (Sl. 5-13(d)). Napomenimo da razlika između rešenja 2 i 3 ilustruje bitnu situaciju kad zbog propusta u organizaciji VHDL kôda može doći do sinteze dodatnog hardvera.

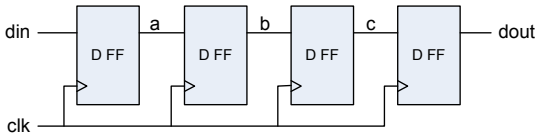
Sinteza varijable u flip-flop. Uslovi pod kojim se varijable sintetišu u flip-flopove nešto su složeniji u odnosu na one koji važe za signale. Prvo, kao i za signale, potrebno je da dodela vrednosti varijabli bude uslovljena promenom nekog drugog signala. Drugo, potrebno je da se vrednost varijable, putem dodele nekom signalu, prenese izvan procesa. Na primer, varijabla *v* iz sledećeg VHDL kôda se sintetiše u flip-flop (*x* je signal):

```
PROCESS(clk)
   VARIABLE v : BIT;
BEGIN
  IF(clk'EVENT AND clk='1')THEN
     v := a;
  END IF;
  x <= v;
END PROCESS;
```

Oba neophodna uslova za sintezu varijable u flip-flop su ispunjena: dodela vrednosti varijabli *v* je sinhronizovana s rastućom ivicom signala *clk* i njena vrednost se iznosi van procesa (posredstvom signala *x*).

Drugi slučaj kad se varijabla sintetiše u flip-flop je onda kada se varijabla koristi u kôdu procesa pre nego što joj je dodeljena vrednost (v. Pr. 5-18).

Pr. 5-18 Pomerački registar – realizacija pomoću varijabli.



Sl. 5-15 Pomerački registar iz Pr. 5-18.

U ovom primeru razmotrićemo tri VHDL arhitekture napisane s namerom da modeliraju pomerački registar sa Sl. 5-15. Arhitekture *arch_v2* i *arch_v3* se ispravno sintetišu u strukturu sa Sl. 5-15, dok se arhitektura *arch_v1* sintetiše u kolo koje umesto četiri sadrži samo jedan flip-flop.

```
1  ----------------------------------------------------------------------
2  ENTITY pomreg IS
3    PORT (din,clk : IN BIT;
4          dout: OUT BIT);
5  END pomreg;
6  ----------------------------------------------------------------------
7  ARCHITECTURE arch_v1 OF pomreg IS
8  BEGIN
```

```
9    PROCESS (clk)
10      VARIABLE a,b,c : BIT;
11   BEGIN
12      IF(clk'EVENT AND clk='1') THEN
13         a := din;
14         b := a;
15         c := b;
16         dout <= c;
17      END IF;
18   END PROCESS;
19 END arch_v1;
20 ------------------------------------------------------------------
21 ARCHITECTURE arch_v2 OF pomreg IS
22 BEGIN
23   PROCESS (clk)
24      VARIABLE a,b,c : BIT;
25   BEGIN
26      IF(clk'EVENT AND clk='1') THEN
27         dout <= c;
28         c := b;
29         b := a;
30         a := din;
31      END IF;
32   END PROCESS;
33 END arch_v2;
34 ------------------------------------------------------------------
35 ARCHITECTURE arch_v3 OF pomreg IS
36   SIGNAL a,b,c : BIT;
37 BEGIN
38   PROCESS (clk)
39   BEGIN
40      IF(clk'EVENT AND clk='1') THEN
41         a <= din;
42         b <= a;
43         c <= b;
44         dout <= c;
45      END IF;
46   END PROCESS;
47 END arch_v3;
48 ------------------------------------------------------------------
```

U procesu iz arhitekture *arch_v1* deklarisane su tri varijable (*a*, *b* i *c*, linija 10) koje bi trebalo da memorišu vrednosti tri krajnje leve bitske pozicije pomeračkog registra sa Sl. 5-15. (Krajnja desna pozicija registra pokrivena je signalom *dout*). U telu procesa, sinhronizovano s ivicom taktnog signala, dolazi do prenosa vrednosti kroz varijable, i to s leva na desno, počev od upisa nove vrednosti sa ulaza *din* u varijablu *a*, pa sve do prenosa vrednosti varijable *c* na izlazni port *dout*. Međutim, budući da za varijable važi trenutna, a ne odložena dodela, naredbe dodele iz linija 13-16 se sažimaju u jednu ekvivalentnu naredbu: *dout*<=*din*, a kôd tela procesa se svodi na:

```
IF(clk'EVENT AND clk='1') THEN
  dout <= din;
END IF;
```

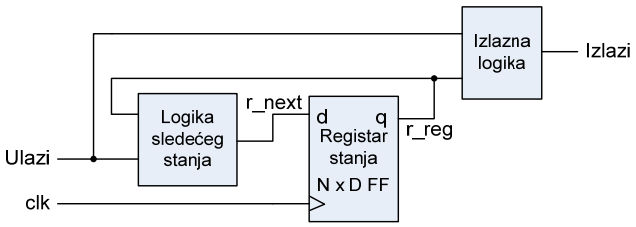
Uslovi za sintezu signala *dout* u flip-flop su ispunjeni, ali to je samo jedan flip-flop, a ne četiri koliko je potrebno da bi se dobilo ispravno rešenje.

U procesu iz arhitekture *arch_v2*, redosled naredbi dodele je promenjen, tako što se sada prenos vrednosti kroz varijable obavlja s desna na levo. U liniji 27, na izlaz *dout* se prosleđuje vrednost varijable *c*. Međutim, to nije vrednost koja će varijabli *c* biti dodeljena u sledećoj liniji, već je to vrednost koja je varijabli *c* bila dodeljena prilikom prethodnog ciklusa izvršenja procesa. Slično važi i za varijable *b* i *a*, koje se kao i *c* koriste pre nego što im se dodeli nova vrednost. Zbog toga se za svaku varijablu ugrađuje u sintetizovano kolo po jedan flip-flop, koji treba da sačuva vrednost varijable do sledećeg ciklusa izvršenja procesa, tj. do naredne rastuće ivice taktnog signala. Takođe, u flip-flop se sintetiše i signal *dout* (zbog naredbe iz linije 27), što kao konačan ishod ima ispravno rešenje.

U arhitekturi *arch_v3*, varijable su zamenjene signalima (linija 36). Ova arhitektura takođe pravilno opisuje rad pomeračkog registra, bez obzira na to što se dodela vrednosti signalima obavlja u normalnom redosledu, kao u arhitekturi *arch_v1* - od *din* do *dout*. Budući da se u opisu koriste samo signali, i pri tom je dodela vrednosti signalima sinhronizovana s rastućom ivicom signala *clk*, svaki od njih, računajući tu i signal *dout*, se sintetiše u jedan flip-flop. Napomenimo da razlike u sintezi između varijabli i signala potiču od činjenice da za signale u procesu važi odložena dodela. Zbog toga se u procesu koriste stare vrednosti signala (one koje su im bile dodeljene prilikom prethodnog izvršenja procesa). Još jedna posledica odložene dodele jeste i to da redosled naredbi dodele u linijama 41-44 nije od značaja i da bi i bilo koji drugi redosled, uključujući i onaj iz arhitekture *arch_v2*, bio korektan.

5.7. Registarske komponente

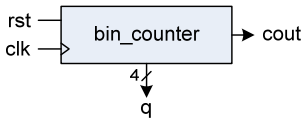
Registarske komponente su sinhrona sekvencijalna digitalna kola standardne (prepoznatljive) funkcije, poput prihvatnih (stacionarnih), pomeračkih i brojačkih registara. Blok dijagram registarske komponente prikazan je Sl. 5-16. U osnovi svake registarske komponente se nalazi kolekcija D flip-flopova sa zajedničkim signalom takta – tzv. *registar stanja*. Registar stanja memoriše tekuće stanje kola. Kombinacioni blok ″logika sledećeg stanja″ određuje naredno stanje kola na osnovu vrednosti ulaznih signala i tekućeg stanja. ″Logika izlaza″ je još jedan kombinacioni blok, koji na osnovu tekućeg stanja i vrednosti ulaznih signala generiše izlazne signale kola. U osnovi, blok dijagram registarske komponente se ne razlikuje od uopštenog blok dijagrama sinhronih sekvencijalnih kola i konačnih automata (v. 6). Specifičnost registarskih komponenti je u tome što se funkcije blokova sledećeg stanja i izlaza mogu izraziti jednostavnim i regularnim logičkim ili aritmetičkim funkcijama. Na primer, funkcija logike sledećeg stanja brojača je oblika: *″r_next <= r_reg + 1″*, a pomeračkog registra: *″r_next <= r_reg(N-2 DOWNTO 0) & '0'″*.



Sl. 5-16 Uopšteni konceptualni dijagram registarske komponente.

U nastavku ovog poglavlja sledi nekoliko primera VHDL opisa tipičnih registarskih komponenti.

Pr. 5-19 Binarni brojač



Sl. 5-17 4-bitni binarni brojač.

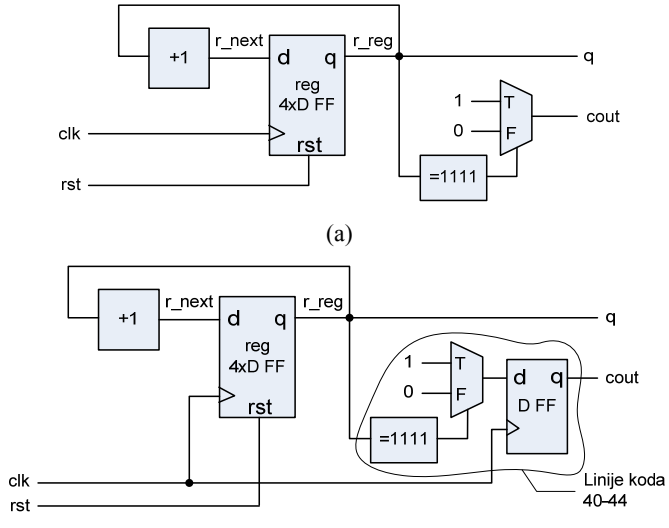
Na Sl. 5-17 prikazan je grafički simbol 4-bitnog binarnog brojača. Stanja brojača, koja se javljaju na izlazu *q*, su redom: ″0000″, ″0001″, ..., ″1111″. Stanja brojača se interpretiraju kao neoznačeni celi brojevi: 0, 1, ..., 15. Sa svakom rastućom ivicom taktnog signala brojač prelazi u sledeće stanje koje je za 1 veće od prethodnog. Nakon stanja 15 brojač se vraća u stanje 0. Signal *rst* služi za asinhrono resetovanje brojača, odnosno za forsirano postavljanje brojača u početno stanje, ″0000″. Izlazni signal *cout* (izlazni prenos brojača) ima vrednost ′1′ za vreme dok je brojač u završnom stanju, ″1111″, dok u svim ostalim stanjima važi *cout*=′0′. VHDL kôd koji sledi sadrži tri arhitekture: *arch_v1* i *arch_v3* predstavljaju ispravne opise, dok je *arch_v2* neispravan opis 4-bitnog binarnog brojača.

```
1  ----------------------------------------------------------------
2  LIBRARY IEEE;
3  USE IEEE.STD_LOGIC_1164.ALL;
4  USE IEEE.NUMERIC_STD.ALL;
5  ----------------------------------------------------------------
6  ENTITY bin_counter IS
7   PORT (clk,rst : IN STD_LOGIC;
8         cout : OUT STD_LOGIC;
9         q : OUT STD_LOGIC_VECTOR(3 DOWNTO 0));
10 END bin_counter;
11 - Ispravan trosegmentni opis -------------------------------------
12 ARCHITECTURE arch_v1 OF bin_counter IS
13   SIGNAL r_reg, r_next : UNSIGNED(3 DOWNTO 0);
14 BEGIN
15    --- Registar stanja ----------------------------------------
16    PROCESS(clk,rst)
17    BEGIN
18       IF(rst = '1') THEN
19          r_reg <= (OTHERS => '0');
20       ELSIF(clk'EVENT AND clk = '1') THEN
21          r_reg <= r_next;
22       END IF;
23    END PROCESS;
24    -- Logika sledeceg stanja ---------------------------------
25    r_next <= r_reg + 1;
26    -- Izlazna logika -----------------------------------------
27    q <= STD_LOGIC_VECTOR(r_reg);
28    cout <= '1' WHEN r_reg = "1111" ELSE '0';
29 END arch_v1;
30 - Neispravan dvosegmentni opis -------------------------------
31 ARCHITECTURE arch_v2 OF bin_counter IS
32   SIGNAL r_reg : UNSIGNED(3 DOWNTO 0);
33 BEGIN
34    PROCESS(clk,rst)
35    BEGIN
36      IF(rst = '1') THEN
```

```
37          r_reg <= (OTHERS => '0');
38        ELSIF(clk'EVENT AND clk = '1') THEN
39          r_reg <= r_reg + 1;
40          IF(r_reg = "1111") THEN
41             cout <= '1';
42          ELSE
43             cout <= '0';
44          END IF;
45        END IF;
46    END PROCESS;
47     -- Izlazna logika ------------------------------------------
48    q <= STD_LOGIC_VECTOR(r_reg);
49  END arch_v2;
50  - Ispravan jednosegmentni opis ---------------------------------
51  ARCHITECTURE arch_v3 OF bin_counter IS
52  BEGIN
53     PROCESS(clk,rst)
54        VARIABLE q_tmp : UNSIGNED(3 DOWNTO 0);
55     BEGIN
56        IF(rst = '1') THEN
57           q_tmp := (OTHERS => '0');
58        ELSIF(clk'EVENT AND clk = '1') THEN
59           q_tmp := q_tmp + 1;
60        END IF;
61        IF(q_tmp = "1111") THEN
62           cout <= '1';
63        ELSE
64           cout <= '0';
65        END IF;
66        q <= STD_LOGIC_VECTOR(q_tmp);
67     END PROCESS;
68  END arch_v3;
69  ------------------------------------------------------------------
```

Kôd arhitekture *arch_v1* kreiran je u skladu sa blok dijagramom sa Sl. 5-16. Registar stanja je opisan u vidu procesa, dok se za opis logike sledećeg stanja i izlazne logike koriste dve zasebne sekcije konkurentnog kôda. Logiku sledećeg stanja čini samo jedna konkurentna naredba dodele kojom se tekuća vrednost registra stanja (*r_reg*) povećava za 1 (linija 25). Operacija sabiranja koja se koristi u ovoj naredbi je razlog zašto su signali *r_reg* i *r_next* deklarisani s tipom *unsigned* (linija 13). Budući da za ovaj tip važi: ″1111″ + 1 = ″0000″, naredba inkrementiranja iz linije 25 je dovoljna da u potpunosti opiše funkciju logike sledećeg stanja binarnog brojača. Sekcija kôda ″Izlazna logika″ sadrži dve konkurentne naredbe dodele. U prvoj (linija 27), tekuće stanje brojača (*r_reg*) se prenosi na izlazni port *q*, uz konverziju tipa *unsigned* u tip *std_logic_vector*. Druga naredba izlazne logike (linija 28) reguliše vrednost izlaznog signala *cout*. Konceptualni dijagram arhitekture *arch_v1* prikazan je na Sl. 5-18(a).

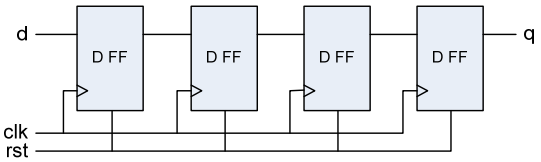
Arhitektura *arch_v2* predstavlja neuspešan pokušaj da se funkcije brojanja i izlaznog prenosa opišu zajedno s registrom stanja u okviru istog procesa. Problem koji postoji u ovom rešenju odnosi se na signal izlaznog prenosa *cout*. Naredba *if* koja reguliše vrednost ovog signala (linije 40-44) ugnježdena je u *elsif* granu čiji uslov odgovara rastućoj ivici taktnog signala *clk*. Zbog toga se signal *cout* sintetiše u flip-flop, a izlaz *cout* kasni za jedan taktni ciklus u odnosu na očekivano ponašanje, što se jasno može videti na osnovu konceptualnog dijagrama ove arhitekture sa Sl. 5-18(b).



Sl. 5-18 Konceptualni dijagrami VHDL opisa 4-bitnog binarnog brojača iz Pr. 5-19: (a) konceptualni dijagram za *arch_v1* i *arch_v3*; (b) konceptualni dijagram arhitekture *arch_v2*.

U arhitekturi *arch_v3*, celokupan opis binarnog brojača smešten je unutar jednog procesa, a problem koji postoji u arhitekturi *arch_v2* prevaziđen je uvođenjem varijable *q_tmp*. Pošto za varijable ne važi odložena dodela, vrednost koju *q_tmp* ima u uslovu *q_tmp = ″1111″* iz druge *if* naredbe u procesu (linije 61-65) upravo je ona koja je ovoj promenljivoj dodeljena u prvoj *if* naredbi (linije 56-60). Takođe, varijabla *q_tmp* zadovoljava uslove za sintezu u registar: u liniji 59 varijabli *q_tmp* se dodeljuje vrednost sinhronizovano s rastućom ivicom signala *clk*, a u liniji 66 vrednost varijable *q_tmp* se prenosi signalu *q* i na taj način iznosi iz procesa. Konceptualni dijagram arhitekture *arch_v3* identičan je konceptualnom dijagramu arhitekture *arch_v1* sa Sl. 5-18(a).

Pr. 5-20 Pomerački registar



Sl. 5-19 Pomerački registar sa asinhronim resetom.

U ovom primeru predstavljena su dva standardna načina za opis pomeračkog registra. Razmotrićemo registar sa Sl. 5-19 koji u odnosu na onaj iz Pr. 5-18 dodatno poseduje mogućnost asinhronog resetovanja (ulaz *rst*). U prvom rešenju (arhitektura *arch_v1*) koriste se signali, a drugom (arhitektura *arch_v2*) varijable. Sintetizovana kola su ista u oba slučaja i odgovaraju strukturi sa Sl. 5-19. Signal *reg* iz arhitekture *arch_v*1 se sintetiše u 4-bitni registar zato što se dodela vrednosti ovom signalu obavlja sinhronizovano s promenom signala *clk* (linije 17-18). U arhitekturi *arch_v2*, promena signala *clk* inicira upis u varijablu *reg* (linije 31-33) i budući da vrednost ove varijable napušta proces (tj. prenosi se na port *q*)

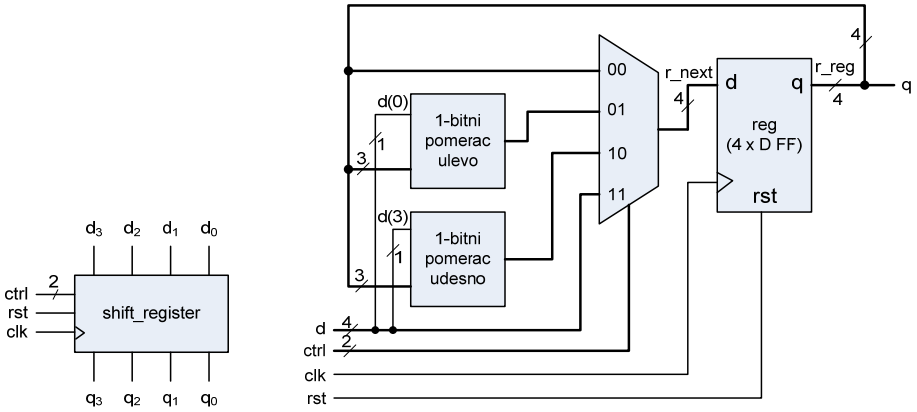
(linija 34) i ona se sintetiše takođe u registar. Funkcija pomeranja, u oba rešenja, realizovana je pomoću operator konkatenacije.

```
1  ----------------------------------------------------------------------
2  LIBRARY IEEE;
3  USE IEEE.STD_LOGIC_1164.ALL;
4  ----------------------------------------------------------------------
5  ENTITY pomreg IS
6  PORT ( d,clk,rst: IN STD_LOGIC;
7         q: OUT STD_LOGIC);
8  END pomreg;
9  -- Resenje 1 ---------------------------------------------------------
10 ARCHITECTURE arch_v1 OF pomreg IS
11    SIGNAL reg : STD_LOGIC_VECTOR(3 DOWNTO 0);
12 BEGIN
13   PROCESS (clk, rst)
14   BEGIN
15     IF(rst='1') THEN
16       reg <= (OTHERS => '0');
17     ELSIF(clk'EVENT AND clk='1') THEN
18       reg <= d & reg(3 DOWNTO 1);
19     END IF;
20   END PROCESS;
21   q <= reg(0);
22 END arch_v1;
23 -- Resenje 2 ---------------------------------------------------------
24 ARCHITECTURE arch_v2 OF pomreg IS
25 BEGIN
26   PROCESS (clk, rst)
27      VARIABLE reg : STD_LOGIC_VECTOR(3 DOWNTO 0);
28   BEGIN
29     IF(rst='1') THEN
30        reg := (OTHERS => '0');
31     ELSIF(clk'EVENT AND clk='1') THEN
32        reg := d & reg(3 DOWNTO 1);
33     END IF;
34     q <= reg(0);
35   END PROCESS;
36 END arch_v2;
```

Pr. 5-21 Univerzalni pomerački registar

Univerzalni pomerački registar je registarska komponenta koja poseduje mogućnost paralelnog upisa podatka i pomeranja upisanog podatka u oba smera. Na raspolaganju su četiri operacije: paralelni upis, pomeranje ulevo, pomeranje udesno i pauza (tj. nepromenjeno stanje). Na Sl. 5-20 su prikazani grafički simbol i blok dijagram 4-bitnog univerzalnog pomeračkog registra. Ulaz $d(0)$ i izlaz $q(3)$ se koriste kao serijski ulaz i serijski izlaz pri pomeranju ulevo, a ulaz $d(3)$ i izlaz $q(0)$ kao serijski ulaz i serijski izlaz pri pomeranju udesno. Željena operacija se bira posredstvom dvobitnog upravljačkog signal *ctrl*. Izbor operacije zapravo se svodi na izbor podatka koji će rastućom ivicom takta biti upisan u registar stanja. Operacije pomeranja realizuju dva kombinaciona pomerača, jedan za pomeranje ulevo, a drugi za pomeranje udesno. Treba napomenuti da pomerači sa Sl. 5-20(b) ne sadrže logička kola, već samo veze kojim se na odgovarajući način povezuju

ulazi i izlazi ovih blokova. Tako se na izlazu pomerača ulevo javlja: *r_reg(2)*, *r_reg(1)*, *r_reg(0)*, *d(0)*, a na izlazu pomerača udesno: *d(3), r_reg(3)*, *r_reg(2)*, *r_reg(1)*. Paralelni upis podrazumeva upis podatka uzetog s ulaza *d* u registar stanja, a ″pauza″ upis u registar stanja njegove sopstvene (tekuće) vrednosti.



Sl. 5-20 Univerzalni pomerački registar: (a) grafički simbol; (b) blok dijagram.

U nastavku su data dva VHDL opisa univerzalnog pomeračkog registra. Prvi opis (arhitektura *arch_v1*) je usklađen sa blok dijagramom sa Sl. 5-20(b). Registar stanja je opisan pomoću procesa, a kombinacioni deo kola (logika sledećeg stanja) pomoću konkurentne naredbe *select*. U drugom opisu (arhitektura *arch_v2*), celokupno kolo je opisano jednim procesom u kome je funkcija logike sledećeg stanja realizovana pomoću sekvencijalne naredbe *case*. Konceptualni dijagrami obe arhitekture su identični i podudaraju se sa blok dijagramom sa Sl. 5-20(b).

```
1  ----------------------------------------------------------------------
2  LIBRARY IEEE;
3  USE IEEE.STD_LOGIC_1164.ALL;
4  ----------------------------------------------------------------------
5  ENTITY shift_register IS
6    PORT (clk, rst : IN STD_LOGIC;
7          ctrl : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
8          d : IN STD_LOGIC_VECTOR(3 DOWNTO 0);
9          q : OUT STD_LOGIC_VECTOR(3 DOWNTO 0));
10 END shift_register;
11 - Trosegmentni opis ------------------------------------------------
12 ARCHITECTURE arch_v1 OF shift_register IS
13   SIGNAL r_reg : STD_LOGIC_VECTOR(3 DOWNTO 0);
14   SIGNAL r_next : STD_LOGIC_VECTOR(3 DOWNTO 0);
15 BEGIN
16   -- Registar stanja ----------------------------------------------
17   PROCESS (clk,rst)
18   BEGIN
19     IF(rst='1')THEN
20        r_reg <= (OTHERS => '0');
21     ELSIF(clk'EVENT AND clk='1') THEN
22        r_reg <= r_next;
23     END IF;
24   END PROCESS;
```

```
25   -- Logika sledeceg stanja -------------------------------------
26   WITH ctrl SELECT
27     r_next <= r_reg                     WHEN "00",   -- pauza
28               r_reg(2 DOWNTO 0) & d(0) WHEN "01",   -- ulevo
29               d(3) & r_reg(3 DOWNTO 1) WHEN "10",   -- udesno
30               d                        WHEN OTHERS; -- "load"
31   -- Izlazna logika -----------------------------------------------
32   q <= r_reg;
33 END arch_v1;
34 -- Dvosegmentni opis ----------------------------------------------
35 ARCHITECTURE arch_v2 OF shift_register IS
36   SIGNAL r_reg : STD_LOGIC_VECTOR(3 DOWNTO 0);
37 BEGIN
38   -- Registar stanja + logika sledećeg stanja --------------------
39   PROCESS (clk,rst)
40   BEGIN
41      IF(rst='1')THEN
42         r_reg <= (OTHERS => '0');
43      ELSIF(clk'EVENT AND clk='1') THEN
44        CASE ctrl IS
45          WHEN "00" => NULL;
46          WHEN "01" => r_reg <= r_reg(2 DOWNTO 0) & d(0);
47          WHEN "10" => r_reg <= d(3) & r_reg(3 DOWNTO 1);
48          WHEN OTHERS => r_reg <= d;
49        END CASE;
50      END IF;
51   END PROCESS;
52   -- Izlazna logika -----------------------------------------------
53   q <= r_reg;
54 END arch_v2;
```

Pr. 5-22 Programabilni brojač po modulu *m* - dvosegmentni kôd

U ovom primeru predstavljen je VHDL opis 4-bitnog programabilnog brojača po modulu *m*, tj. ″mod-*m* brojača″. Parametar *m* definiše osnovu brojanja i zadaje se posredstvom posebnog 4-bitnog ulaza. Dozvoljeni opseg vrednosti ovog parametra je od ″0010″ do ″1111″, što znači da se brojač može ″programirati″ kao mod-2, mod-3, ... ili mod-15 brojač. Sekvenca brojanja mod-*m* brojača je: 0, 1, ..., *m*-1. Sve dok je tekuće stanje brojača različito od *m*-1, njegovo sledeće stanje je za 1 veće od prethodnog; ako je brojač u završnom stanju, *m*-1, njegovo sledeće stanje je 0. Prvo od dva izložena rešenja, *arch_v1*, zasnovano je upravo na ovom zapažanju.

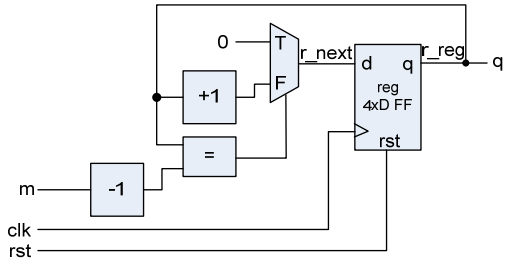
```
1  ----------------------------------------------------------------------
2  LIBRARY IEEE;
3  USE IEEE.STD_LOGIC_1164.ALL;
4  USE IEEE.NUMERIC_STD.ALL;
5  ----------------------------------------------------------------------
6  ENTITY prog_counter IS
7   PORT (clk,rst : IN STD_LOGIC;
8         m : IN STD_LOGIC_VECTOR(3 DOWNTO 0);
9         q : OUT STD_LOGIC_VECTOR(3 DOWNTO 0));
10 END prog_counter;
11 ----------------------------------------------------------------------
12 ARCHITECTURE arch_v1 OF prog_counter IS
```

```
13   SIGNAL r_reg : UNSIGNED(3 DOWNTO 0);
14   SIGNAL r_next : UNSIGNED(3 DOWNTO 0);
15 BEGIN
16   --- Registar stanja ------------------------------------------
17   PROCESS(clk,rst)
18   BEGIN
19      IF(rst = '1') THEN
20         r_reg <= (OTHERS => '0');
21      ELSIF(clk'EVENT AND clk = '1') THEN
22         r_reg <= r_next;
23      END IF;
24   END PROCESS;
25   -- Logika sledeceg stanja -----------------------------------
26   r_next <= (OTHERS => '0') WHEN r_reg = (UNSIGNED(m) - 1) ELSE
27             r_reg + 1;
28   -- Izlazna logika -------------------------------------------
29   q <= STD_LOGIC_VECTOR(r_reg);
30 END arch_v1;
31 - Optimizovano rešenje -----------------------------------------
32 ARCHITECTURE arch_v2 OF prog_counter IS
33   SIGNAL r_reg : UNSIGNED(3 DOWNTO 0);
34   SIGNAL r_next, r_inc : UNSIGNED(3 DOWNTO 0);
35 BEGIN
36   --- Registar -------------------------------------------------
37   PROCESS(clk,rst)
38   BEGIN
39     IF(rst = '1') THEN
40        r_reg <= (OTHERS => '0');
41     ELSIF(clk'EVENT AND clk = '1') THEN
42        r_reg <= r_next;
43     END IF;
44   END PROCESS;
45   -- Logika sledeceg stanja -----------------------------------
46   r_inc <= r_reg + 1;
47   r_next <= (OTHERS => '0') WHEN r_inc = UNSIGNED(m) ELSE
48             r_inc;
49   -- Izlazna logika -------------------------------------------
50   q <= STD_LOGIC_VECTOR(r_reg);
51 END arch_v2;
52 ---------------------------------------------------------------------
```

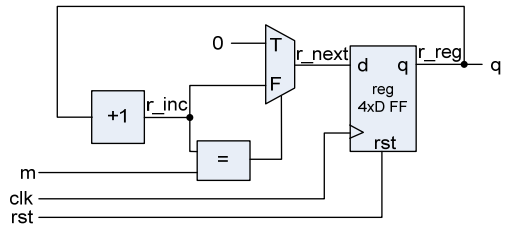
Interni signali *r_reg* i *r_next* su tipa *unsigned* zbog toga što se sa stanjem brojača manipuliše na način kao sa neoznačenim celim brojevima. Međutim, budući da su portovi *m* i *q* tipa *std_logic_vector*, neophodna je konverzija tipa podataka: *unsigned*(*m*) iz linije 26 konvertuje *m* u neoznačen ceo broj, dok *std_logic_vector*(*r_reg*) iz linije 29 konvertuje *r_reg* u tip *std_logic_vector*. Konceptualni dijagram arhitekture *arch_v1* prikazan je na Sl. 5-21(a). Logiku sledećeg stanja čine: inkrementer (realizuje operaciju sabiranja iz *else* grane naredbe *when* - linija 27), dekrementer (realizuje operaciju oduzimanja iz uslova *when* naredbe iz linije 26) i komparator (realizuje operator poređenja jednakosti iz uslova *when* naredbe – linija 26).

Uočimo da je logički uslov *r_reg* = *unsigned(m)* – 1 identičan uslovu *r_reg* + 1 = *unsigned(m)*. Imajući u vidu da je *r_reg* + 1 već neophodno za operaciju inkrementiranja (linija 27), u mogućnosti smo da isto ovu operaciju iskoristimo i za poređenje iz uslova *when* naredbe (linija 26) i tako eliminišemo dekrementer. Arhitektura *arch_v2* sadrži

revidiran VHDL kôd. Sada je operacija *r_reg* + 1 izdvojena u posebu naredbu *r_inc* <= *r_reg* + *1* (linija 46), a signal *r_inc* se koristi i za inkrementiranje brojača (*else* grane naredbe *when* – linija 48) i za poređenje u uslovu *when* naredbe (linija 47). Konceptualni dijagram revidiranog kôda prikazan je na Sl. 5-21(b). U suštini, optimizacija koja je sprovedena nad kôdom arhitekture *arch_v1* predstavlja još jedan primer primene tehnike deobe operatora (v. 4.5.1).



(a)



(b)

Sl. 5-21 Konceptualni dijagrami programabilnog mod-*m* brojača: (a) polazno rešenje; (b) rešenje nakon izvršene optimizacije.

Pr. 5-23 Programabilni brojač po modulu *m* - jednosegmentni kôd

U ovom primeru predstavljen je jednosegmentni kôd mod-*m* brojača iz Pr. 5-22, tj. VHDL opis u kome je celokupna logika mod-*m* brojača realizovana jednim procesom. Prvi pokušaj pisanja jednosegmentnog kôda je neuspešan (arhitektura *arch_v3*, linije 1-17). Kao što znamo, vrednost signal u procesu se ažurira na kraju procesa. To znači da se *r_reg* postavlja na novu vrednost *r_reg* + 1 (linija 10) tek na kraju procesa, a da se u naredbi u kojoj se vrši poređenje *r_reg* = *unsigned(m*) (linija 11) još uvek koristi njegova stara (prethodna) vrednost. Pošto korektna vrednost signala *r_reg* kasni za jedan taktni ciklus, brojač će odbrojati jedno stanje više, što znači da arhitektura *arch_v3* zapravo predstavlja opis mod-(*m*+1) brojača.

Prvi način kako se problem koji postoji u arhitekturi *arch_v*3 može rešiti jeste da se operacija inkrementiranja izmesti iz procesa (arhitektura *arch_v4*). Pošto se sada proces (linije 23-34) i naredba dodele *r_inc* <= *r_reg* + 1 (linija 35) izvršavaju konkurentno, problem neažurnosti signala *r_reg* više ne postoji. Međutim, ovo nije jednosegmentni kôd. Drugi način za rešenje istog problema podrazumeva korišćenje varijable za privremeno čuvanje rezultata operacije inkrementiranja (arhitektura *arch_v5*). Za razliku od signala, dodela vrednosti varijabli je trenutna, a novoupisana vrednost je dostupna za korišćenje već u sledećoj sekvencijalno naredbi.

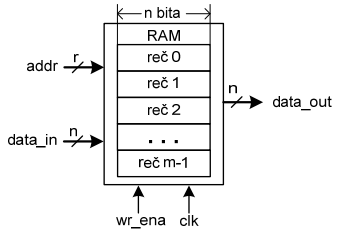
```
---- Pogresno resenje -------------------------------------------
ARCHITECTURE arch_v3 OF prog_counter IS
 SIGNAL r_reg : UNSIGNED(3 DOWNTO 0);
BEGIN
 PROCESS(clk,rst)
 BEGIN
   IF(rst = '1') THEN
       r_reg <= (OTHERS => '0');
     ELSIF(clk'EVENT AND clk = '1') THEN
       r_reg <= r_reg + 1;
       IF(r_reg = UNSIGNED(m)) THEN
          r_reg <= (OTHERS => '0');
       END IF;
     END IF;
  END PROCESS;
  q <= STD_LOGIC_VECTOR(r_reg);
END arch_v3;
----- Ispravno resenje ----------------------------------------
ARCHITECTURE arch_v4 OF prog_counter IS
  SIGNAL r_reg : UNSIGNED(3 DOWNTO 0);
  SIGNAL r_inc : UNSIGNED(3 DOWNTO 0);
BEGIN
   PROCESS(clk,rst)
   BEGIN
     IF(rst = '1') THEN
       r_reg <= (OTHERS => '0');
     ELSIF(clk'EVENT AND clk = '1') THEN
       IF(r_inc = UNSIGNED(m)) THEN
          r_reg <= (OTHERS => '0');
       ELSE
          r_reg <= r_inc;
       END IF;
     END IF;
  END PROCESS;
  r_inc <= r_reg + 1;
  q <= STD_LOGIC_VECTOR(r_reg);
END arch_v4;
----- Ispravno resenje koje koristi varijablu -------------------
ARCHITECTURE arch_v5 OF prog_counter IS
BEGIN
  PROCESS(clk,rst)
    VARIABLE q_tmp : UNSIGNED(3 DOWNTO 0);
  BEGIN
     IF(rst = '1') THEN
       r_reg <= (OTHERS => '0');
     ELSIF(clk'EVENT AND clk = '1') THEN
       q_tmp := r_reg + 1;
       IF(q_tmp = UNSIGNED(m)) THEN
         r_reg <= (OTHERS => '0');
       ELSE
         r_reg <= q_tmp;
       END IF;
     END IF;
  END PROCESS;
  q <= STD_LOGIC_VECTOR(r_reg);
```

```
56 END arch_v5;
```

5.8. RAM

RAM (engl. ***R**andom **A**ccess **M**emory*) je memorijska komponenta u koju se može upisivati i iz koje se može čitati upisana informacija. S logičke tačke gledišta, RAM je organizovan kao kolekciju registara s ograničenjem da se uvek može pristupati samo jednom registru radi upisa ili čitanja. Registar kome se pristupa bira se (tj. adresira) binarnom kombinacijom koja se postavlja na adresne ulaze RAM-a. U nastavku će biti predstavljeni VHDL opisi dva tipa RAM-a koji se razlikuju po načinu realizacije porta za podatke. U prvom rešenju, za razmenu podataka s RAM-om koriste se dva zasebna porta za podatke, jedan za upis, a drugi za čitanje, dok se u drugom za obe ove operacije koristi jedan bidirekcioni port.

RAM sa razdvojenim portovima za upis i čitanje. Na Sl. 5-22 je ilustrovana organizacija RAM-a sa zasebnim portovima za upis i čitanje podataka. Vrednost prisutna na adresnom portu, *addr*, bira (adresira) memorijsku reč kojoj se pristupa. Pri *wr_ena*=´1´, vrednost prisutan na ulaznom portu za podatke, *data_in*, upisuje se u adresiranu reč pod dejstvom rastuće ivice takta, *clk*. Sadržaj adresirane reči je dostupan na izlaznom portu za podatke, *data_out*. Napomenimo da između broja reči u RAM-u, *m*, i broja bita adrese, *r*, postoji sledeća vezi: $r = \lceil \log_2 m \rceil$.



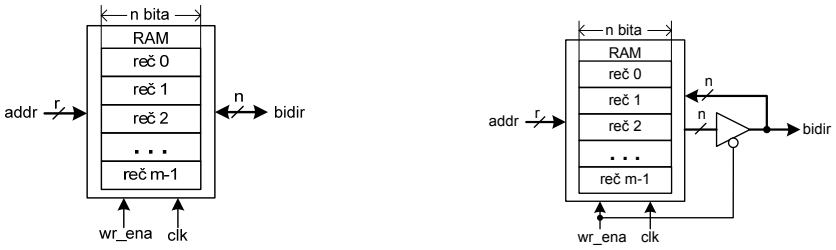
Sl. 5-22 RAM sa razdvojenim ulazim i izlazim portovima za podatke.

Ispod je dat VHDL opis RAM-a sa Sl. 5-22. Modeliran je RAM kapaciteta 16x8, tj. RAM koji sadrži 16 reči dužine 8 bita. Kao i za ROM (v. 4.7), opis RAM-a sadrži definiciju tipa podataka *mem_array* koji predstavlja dvodimenzionalno polje dimenzija 16x8 (linija 14), ali za razliku od ROM-a, a zbog potrebe upisa, ″memorijski medijum″ *memory* nije konstanta već signal (linija 15). Arhitektura sadrži jedan proces (linije 17-24), koji reguliše upis, i jednu konkurentnu naredbu dodele (linija 25), koja je zadužena za čitanje iz memorije. Proces sadrži dve ugnježdene *if* naredbe: prva sinhroniše upis s rastućom ivicom taktnog signala *clk* (linija 19-23), a druga omogućava upis ako važi *wr_ena*=´1´ (linija 20-22). S obzirom na to što linija 21 sadrži sinhronu narednu dodele, signal *mamory* se sintetiše u polje od 16x8=128 flip-flopova. Konkurentno s procesom, naredba dodele iz linije 26 prenosi sadržaj adresirane reči na izlazni port *data_out.*

```
1  ------------------------------------------------------------------
2  LIBRARY IEEE;
3  USE IEEE.STD_LOGIC_1164.ALL;
4  USE IEEE.NUMERIC_STD.ALL;
5  ------------------------------------------------------------------
6  ENTITY RAM IS
7    PORT (wr_ena, clk : IN STD_LOGIC;
```

```
        addr : IN STD_LOGIC_VECTOR(3 DOWNTO 0);
        data_in : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
        data_out : OUT STD_LOGIC_VECTOR(7 DOWNTO 0));
END RAM;
-------------------------------------------------------------------
ARCHITECTURE ram OF RAM IS
TYPE mem_array IS ARRAY (0 TO 15) OF STD_LOGIC_VECTOR(7 DOWNTO 0);
SIGNAL memory : mem_array;
BEGIN
   PROCESS(clk)
   BEGIN
      IF(clk'EVENT AND clk='1') THEN
          IF(wr_ena = '1') THEN
             memory(TO_INTEGER(UNSIGNED(addr))) <= data_in;
          END IF;
      END IF;
   END PROCESS;
   data <= memory(TO_INTEGER(UNSIGNED(addr)));
END ram;
-------------------------------------------------------------------
```

RAM sa ulazno-izlaznim (bidirekcionim) portom za podatke. Organizacija RAM-a sa bidirekcionim portom za podatke prikazana je na Sl. 5-23(a). Struktura je slična onoj sa Sl. 5-22, s tom razlikom što se upis i čitanje podataka sada vrše preko istog porta, *bidir*. Kad je *wr_ena*=´1´, a pod dejstvom rastuće ivice taktnog signala, podatak koji je spolja postavljen na port *bidir*, upisuje se u reč adresiranu vrednošću koja je prisutnom na adresnom portu, *addr*. Kad je *wr_ena*=´0´, podatak iz adresirane reči se prenosi na port *bidir*. Na taj način, signal *wr_ena* reguliše smer bidirekcionog porta *bidir*: za *wr_ena*=´1´, *bidir* se ponaša kao ulazni, a za *wr_ena*=´0´ kao izlazni port. Princip realizacije bidirekcionog porta *bidir* prikazan je na Sl. 5-23(b).



Sl. 5-23 RAM sa bidirekcionim portom za podatke: (a) interfejs; (b) princip realizacije dvosmernog porta *bidir*.

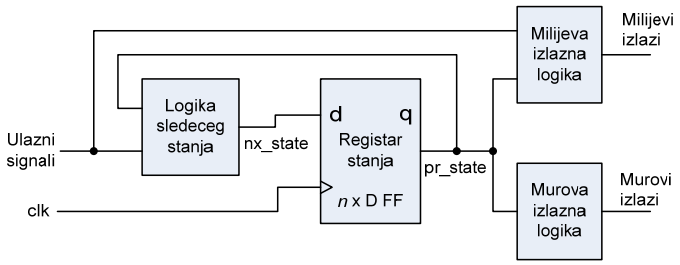
```
-------------------------------------------------------------------
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.NUMERIC_STD.ALL;
-------------------------------------------------------------------
ENTITY RAM IS
  PORT (wr_ena, clk : IN STD_LOGIC;
        addr : IN STD_LOGIC_VECTOR(3 DOWNTO 0);
        bidir : INOUT STD_LOGIC_VECTOR(7 DOWNTO 0));
END RAM;
-------------------------------------------------------------------
```

```
12 ARCHITECTURE ram OF RAM IS
13 TYPE mem_array IS ARRAY (0 TO 15) OF STD_LOGIC_VECTOR(7 DOWNTO 0);
14 SIGNAL memory : mem_array;
15 BEGIN
16    PROCESS(clk)
17    BEGIN
18      IF(clk'EVENT AND clk='1') THEN
19         IF(wr_ena = '1') THEN
20           memory(TO_INTEGER(UNSIGNED(addr))) <= bidir;
21         END IF;
22      END IF;
23    END PROCESS;
24    bidir <= memory(TO_INTEGER(UNSIGNED(addr))) WHEN wr_ena = '0'
25             ELSE (OTHERS => 'Z');
26 END ram;
27 ---------------------------------------------------------------
```

U gornjem VHDL kôdu, koji opisuje RAM sa Sl. 5-23, uočimo liniju 9 u kojoj je deklarisan port *bidir*. Smer porta je postavljen na *inout*, što ukazuje da se radi o bidirekcionom portu. Operacija upisa (*wr_ena*=´1´) realizuje se pomoću procesa, na identičan način kao kod RAM-a sa dva porta (linije 16-23). Bidirekcioni port *bidir* opisan je konkurentnom naredbom *when* (linije 24-25). Pri upisu (*wr_ena*=´1´) port *bidir* je postavljen u stanje visoke impedanse. Pod ovim uslovom, podatak koji se uzima s porta *bidir* i pamti u nizu *memory* (linija 20) određen je vrednošću koja se spolja postavlja na ovaj port. Fizički, bidirekcioni portovi se realizuju pomoću trostatičkih bafera, na šta ukazuje linija 25 u kojoj se na port postavlja vrednost ´Z´ (v. 4.6).

6. KONAČNI AUTOMATI

Konačni automati (engl. ***F**inite **S**tate **M**achine – FSM*) predstavljaju najznačajniju tehniku za modeliranje ponašanja sekvencijalnih kola, koja je naročito pogodna za projektovanje pojedinih tipova digitalnih sistema (kao što su to upravljačke jedinice) čija se funkcija može opisati precizno definisanom sekvencom aktivnosti. Koncept konačnog automata zasnovan je na sledećih pet entiteta: (a) stanja; (b) ulazni signali; (c) izlazni signali; (d) funkcija sledećeg stanja i (e) funkcija izlaza. U svakom trenutku, automat je u jednom od konačnog broja svojih stanja. Novo (sledeće) stanje automata određeno je funkcijom sledećeg stanja i zavisi kako od tekućeg stanja, tako i od trenutnih vrednosti ulaznih signala. Kod sinhronih konačnih automata, prelaz u sledeće stanje se kontroliše signalom takta i dešava se pod dejstvom aktivne ivice takta. Izlazna funkcija definiše vrednosti izlaznih signala. Signal čija vrednost zavisi isključivo od tekućeg stanja automata naziva se *Murovim (Moore) izlazom*. S druge strane, *Milijev (Mealy) izlaz* je izlazni signal čija vrednost ne zavisi samo od tekućeg stanja automata, već i od trenutnih vrednosti ulaznih signala. Konačni automat čiji su svi izlazi Murovog (Milijevog) tipa, naziva se Murovim (Milijevim) automatom. Složeniju konačni automati obično poseduju izlaze oba tipa.



Sl. 6-1 Blok dijagram konačnog automata.

Na Sl. 6-1 je prikazan uopšteni blok dijagram konačnog automata. Ovaj dijagram je sličan konceptualnim dijagramima regularnih registarskih komponenti, kao što su to brojači i pomerački registri (v. 5.7). Međutim, za razliku od registarskih komponenti, kod kojih se funkcija sledećeg stanja može izraziti ″regularnim″ operacijama (npr. +, >>, ...), kod konačnih automat funkcija sledećeg stanja (a to važi i za funkciju izlaza) u opštem slučaju može biti proizvoljna. Registar stanja je memorijski element, sinhronizovan sa taktnim signalom, koji čuva informaciju o tekućem stanje automata. Kad se konačni automat realizuje u hardveru, stanja automata se kodiraju, tako što se svakom stanju pridružuje

jedinstveni *n*-bitni kôd. Informacija sadržana u registru stanja je upravo kôd tekućeg stanja automata. Logika sledećeg stanja je kombinaciona mreža koja realizuje funkciju sledećeg stanja. Njeni ulazi su tekuće stanje (tj. izlazi registra stanja) i ulazni signali. Dijagram sa Sl. 6-1 odgovara uopštenom modelu konačnog automata, koji sadrži i Murove i Milijeve izlaze. Murove izlaze generiše kombinaciona mreža ″Murova izlazna logika″, a Milijeve kombinaciona mreža ″Milijeva izlazna logika″.

Konačni automati nalaze glavnu primenu u modeliranju složenih operacija koje se mogu razložiti na niz uzastopnih koraka. Kod složenih digitalnih sistema, konačni automati često igraju ulogu upravljačke jedinice, koja koordiniše rad ostalih sistemskih jedinica. Naš fokus će upravo biti ovaj aspekt konačnih automata (tj. konačni automat kao upravljačka jedinica). Konačni automati mogu da obavljaju i mnoge jednostavnije zadatke, kao što je detekcija ulazne binarne sekvence zadatog oblika ili generisanje neke specifične izlazne binarne sekvence.

6.1. Predstavljanje konačnih automata

Projektovanje konačnog automata počinje kreiranjem apstraktnog grafičkog modela u vidu *dijagrama stanja* ili u vidu *ASM dijagrama*. Iako vizuelno uočljivo različiti, oba ova dijagrama sadrže sve potrebne informacije o konačnom automatu (stanja, ulaze, izlaze, funkciju sledećeg stanja i funkciju izlaza). Šta više, dijagram stanja se lako može prevesti u ekvivalentan ASM dijagram i obrnuto.

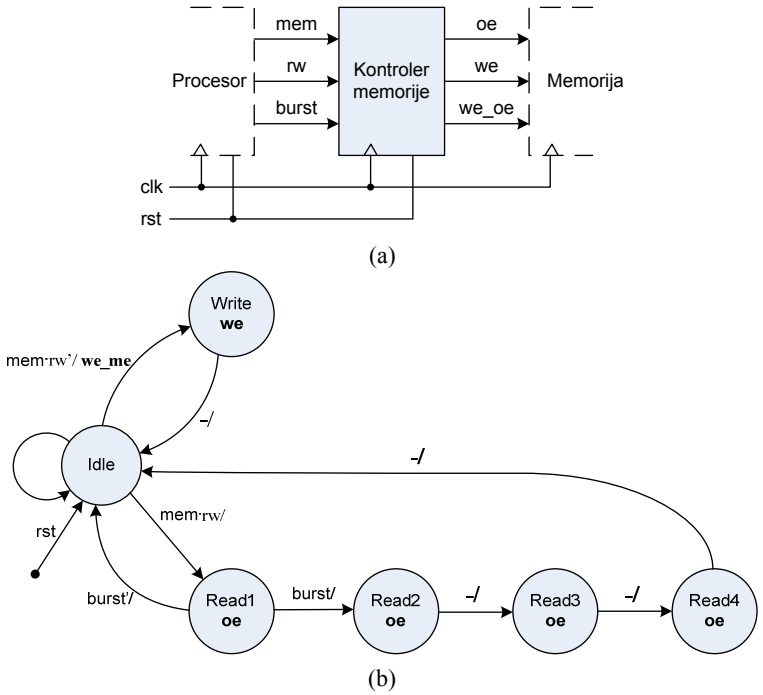
6.1.1. Dijagram stanja

Dijagram stanja se sastoji od *čvorova*, koji se crtaju u vidu krugova i usmerenih potega (ili grana) (Sl. 6-2). Svaki čvor predstavlja jedno stanje konačnog automata. Čvoru je pridruženo jedinstveno simboličko ime koje se zapisuje unutar kruga. Poteg predstavlja prelaz iz jednog u drugo stanje i označen je uslovom pod kojim dolazi do prelaza. Uslov je logički izraz formiran od ulaznih signala. Prelaz je omogućen onda kada odgovarajući logički izraz dobije vrednost logičke 1-ce, odnosno postane tačan. Dijagram stanja definiše takođe i vrednosti izlaznih signala automata. Murovi izlazi su isključivo u funkciji stanja i uobičajeno se zapisuju unutar čvora. S druge strane, Milijevi izlazi zavise i od stanja i od ulaza i zbog toga se pridružuju potezima i vezuju za odgovarajuće uslove.

Sl. 6-2 Notacija u dijagramu stanja: (a) eksplicitna dodela vrednosti izlaznim signalima; (b) implicitna dodela vrednosti izlaznim signalima – navodi se spisak aktivnih signala. Napomena: ime stanje je ST0; *mo* je Murov, a *me* Milijev izlaz. U slučaju pod (a), v*rednost* koja se dodeljuje signalima *mo* i *me* može biti ´0´ ili ´1´; u slučaju pod (b), signalima *mo* i *me* se dodeljuje vrednost (´1´).

Da bi se pojednostavilo zapisivanje, uobičajeno je da se u čvorovima i na granama navode imena samo onih izlaznih signala koji su *postavljeni* (tj. *aktivni*) u datom stanju, odnosno pri datom prelazu. Pri tom se usvaja da izlazni koji nisu navedeni unutar čvora ili na potegu imaju *podrazumevanu* vrednost. Takođe, uobičajeno je da se pod aktivnom vrednošću signala smatra logičko 1, a pod podrazumevanom logička 0, osim ukoliko nije drugačije naglašeno.

Radi potpunijeg razjašnjenja koncepta dijagrama stanja, razmotrićemo rad konačnog automata jednog hipotetičkog kontrolera memorije (Sl. 6-3(a)). Kontroler igra ulogu posrednika između procesora i memorije, tako što interpretira komande koje izdaje procesor i transformiše ih u odgovarajuće sekvence upravljačkih signala kojima deluje na memoriju. Procesor izdaje komande posredstvom signala *mem, rw* i *burst*. Signal *mem* je postavljen na ´1´ uvek kada procesor zahteva pristup memoriji, dok signali *rw* i *burst* ukazuju na zahtevani način pristupa. Signal *rw* svojom vrednošću, ´0´ ili ´1´, ukazuje da li procesor zahteva čitanje iz memorije (*rw*=´1´) ili upis u memoriju (*rw*=´0´). Signal *burst* inicira jednu posebnu vrstu operacije čitanja. Aktiviranjem ovog signala (uz istovremeno *mem*=´1´) procesor zahteva od kontrolera memorije da automatski izvršiti četiri uzastopne operacija čitanja. S druge strane, memorijsko kolo poseduje dva ulazna upravljačka signala, *oe* (dozvola izlaza) i *we* (dozvola upisa). Pri *oe*=´1´ od memorije se zahteva da isporuči sadržaj adresirane memorijske reči (operacija čitanja), a pri *we*=´1´ da u adresiranu reč smesti dostavljeni podatak (operacija upisa). Radi jednostavnijeg prikaza, na Sl. 6-3(a) su izostavljeni signali koji povezuju procesor i memoriju, a služe za prenos adrese i podatka. Kontroler poseduje još jedan izlaz, *we_me*, koji je uvršten u ovaj primer samo iz razloga ilustracije Milijevog izlaza.



Sl. 6-3 Kontroler memorije: (a) blok dijagram; (b) dijagram stanja.

Dijagram stanja kontrolera memorije prikazan je na Sl. 6-3(b). Inicijalno, kontroler je u stanju *Idle*[1], u kojem čeka da procesor zatraži pristup memoriji, tj. da aktivira signal *mem*. Sve dok je izraz *mem'* tačan (tj. *mem*=′0′)[2] kontroler ostaje u stanju *Idle*. Kad signal *mem* postane aktivan, kontroler prelazi ili u stanje *Read1* ili u stanje *Write*, zavisno od vrednosti signala *rw*. Ako je izraz *mem·rw* tačan (tj. *mem*=*rw*=′1′), kontroler prelazi u stanje *Read1*. U ovom stanju aktivan je signal *oe*. S druge strane, ako je tačan izraz *mem·rw'* (tj. *mem*=′1′ i *rw*=′0′), kontroler iz stanja *Idle* prelazi u stanje *Write*. U stanju *Write* aktivan je izlazni signal *we*. Na prvu sledeću rastuću ivicu taktnog signala nakon ulaska u stanje *Read1*, kontroler ispituje signal *burst*. Ako važi *burst*=′0′, operacija čitanja je završena i kontroler se vraća u stanje *Idle*. U suprotnom slučaju, ako važi *burst*=′1′, kontroler će u tri naredna taktna ciklusa proći redom kroz stanja *Read2*, *Read3* i *Read4* pre nego što se vratiti u početno stanje *Idle*. Oznaka ″-″, koja se može videti na pojedinim potezima, npr. na potegu između stanja *Read4* i *Idle*, ili na potegu između stanja *Write* i *Idle*, ukazuje na uvek tačan uslov. Poteg sa uvek tačnim uslovom je uvek omogućen, a odgovarajući prelaz je bezuslovan. Stoga, kontroler boravi u stanjima *Read2, Read3* i *Read4*, kao i u stanju *Write*, samo jedan taktni ciklus i već sa prvom narednom rastućom ivicom takta nastavlja dalje. Takođe, uočimo da se u dijagramu stanja sa Sl. 6-3(b) koristi implicitna dodela vrednosti signalima. Naime, navedena su imena samo onih signala koji u datom stanju ili pri datom prelazu imaju vrednost ′1′.

Pr. 6-1 Detektor ivice

Detektor ivice je sinhrono (taktovano) kolo sa jednim ulazom, *strobe*, i jednim izlazom, *p*. Ulazni signal *strobe* je sporo promenljiv u odnosu na takt kola. Na izlazu *p* generiše se kratkotrajan impuls (trajanja jednog taktnog perioda) uvek kada se *strobe* promeni s ′0′ na ′1′. Naš cilj je da rad ovog kolo opišemo u vidu dijagrama stanja. Osnovna ideja se sastoji u tome da se konstruiše konačni automat koji će imati barem dva stanja, stanje *zero* i stanje *one*, koja će ukazivati na činjenicu da je ulazni signal *strobe* već neko duže vreme na nivou logičke ′0′, odnosno logičke ′1′, a da se izlaz *p* aktivira onda kad automat prelazi iz stanja *zero* u stanje *one*. Razmotrićemo dva moguća rešenja. Prvo je dato u obliku konačnog automata Murovog, a drugo u obliku konačnog automata Milijevog tipa.

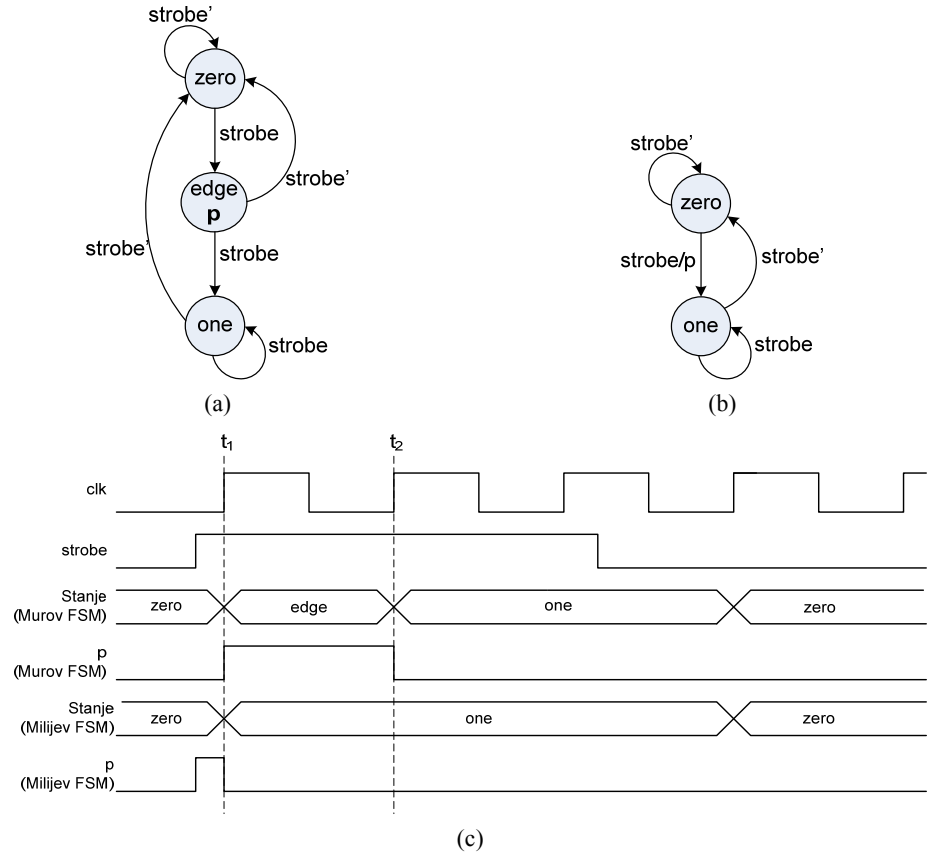
Murov konačni automat detektora ivice prikazan je na Sl. 6-4(a). Kao što se može videti, ovaj automat, osim stanja *zero* i *one*, poseduje još jedno međustanje, *edge*. U stanju *zero*, automat čeka da ulazni signal *strobe* dobije vrednost ′1′, a za sve to vreme na izlazu je *p*=′0′. Onda kad se *strobe* promeni na ′1′, automat prelazi u stanje *edge* u kojem je vrednost izlaznog signal *p* jednaka ′1′. Budući da se u normalnom režimu rada nivo signala *strobe* ne može promeniti tako brzo, automat ostaje u stanju *edge* samo za vreme jednog taktnog perioda i već sa sledećom rastućom ivicom takta prelazi u stanje *one*. U ovom stanju, u kome ponovo važi *p*=′0′, automat ostaje sve dok se vrednost signala *strobe* ne vrati na ′0′, a onda prelazi u stanje *zero*. Ako se ipak desi da *strobe*=′1′ traje kraće od jedne periode takta, automat neće iz stanje *edge* preći u stanje *one*, već će se vratiti u stanje *zero*.

Vremenski dijagrami koji opisuju rad ovog automata prikazani su u gornjem delu Sl. 6-4(c). Vidimo da promena ulaza *strobe*, sama po sebi, ne dovodi do prelaza iz stanja *zero* u stanje *edge*, već se ovaj prelaz inicira rastućom ivicom takta pod uslovom da u tom trenutku

[1] Reč *Idle* znači pasivno, neaktivno.

[2] *mem'* je isto što i $\overline{mem}$ (komplement od *mem*)

(trenutak t_1) važi *strobe*=´1´. Pošto je *p* Murov izlaz, pridružen stanju *edge*, *p*=´1´ traje tačno jedan taktni period. Izlaz *p*=´1´ bi trajao jedan taktni period i da se *strobe* promeni na ´0´ i pre isteka stanja *edge* (tj. pre t_2), jer *p* ne zavisi od ulaza, već samo od tekućeg stanja.



Sl. 6-4 Dijagram stanja detektora ivice: (a) Murov konačni automat; (b) Milijev konačni automat; (c) vremenski dijagrami.

Druga varijanta detektora ivice zasnovana je na konačnom automatu Milijevog tipa (Sl. 6-4(b)). Dijagram stanja ovog automata sadrži samo dva stanja *zero* i *one*. Kad se u stanju *zero* ulaz *strob* promeni s ´0´ na ´1´, automat prelazi u stanje *one*. Kad se u stanju *one* ulaz *strobe* vrati na ´0´ i automat se vraća u stanje *zero*. Gledajući ovaj dijagram stanja, čini se kao da se izlaz *p*, koji je sada Milijevog tipa, postavlja na ´1´ samo u trenutku promene stanja. Međutim, to nije tako, već se *p* postavlja na ´1´ u istom trenutka kada, za vreme stanja *zero*, ulaz *strobe* postane ´1´ i ostaje aktivan sve do naredne rastuće ivice takta kad automat prelazi u stanje *one*. Vremenski dijagrami za ovaj konačni automat prikazani su u donjem delu Sl. 6-4(c). Kao što se može videti, u stanju *zero* izlazni signal *p* prati promene ulaza *strobe*, dok je u stanju *one* njegova vrednost ´0´.

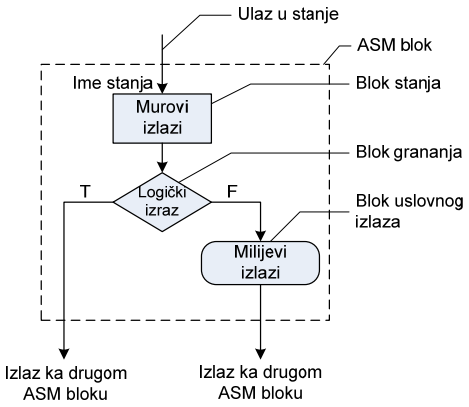
Između Murovih i Milijevih automata postoje tri glavne razlike. Prvo, kod Milijevih automata, po pravilu, potreban je manji broj stanja za opis iste funkcionalnosti. Dugo, odziv Milijevog automata je brži. Pošto su Milijevi izlazi u funkciji ulaza, oni se menjaju istog trenutka kad vrednosti ulaza zadovolje zadati uslov. S druge strane, odziv Murovog

automata se odlaže do sledećeg stanja. Treća razlika se odnosi na kontrolu ″širine″ i tajminga izlaznih signala. Širina (trajanje) izlaznog signala Milijevog tipa određena je širinom ulaznog signala. Milijev izlaz se aktivira u momentu kad ulaz zadovolji postavljeni uslov, a deaktivira onda kad automat pređe u novo stanje. S druge strane, širina izlaznog signala Murovog tipa je približna periodi taktnog signala.

6.1.2. ASM dijagram

Dijagram stanja je pogodno sredstvo za predstavljanje konačnih automata sa malim brojem ulaza i izlaza. Međutim, za predstavljanje složenijih automata, a pre svega onih koji se koriste za modeliranje algoritama koji se realizuju u hardveru, projektanti obično koriste tzv. *algoritamske mašine stanja*, ili ASM dijagrame (engl. ***A**lgorithmic **S**tate **M**achine*).

ASM dijagram se sastoji od više povezanih ASM blokova. Svaki *ASM blok* sadrži jedan blok stanja i, opciono, jedan ili više povezanih blokova grananja i blokova uslovnog izlaza. Tipičan ASM blok je prikazan na Sl. 6-5. Blok stanja se crta u vidu pravougaonika i služi za predstavljanje jednog stanja konačnog automata. Ekvivalentan je čvoru iz dijagrama stanja. Simboličko ime stanja piše se izvan bloka stanja, obično uz njegov gornji levi ugao. Unutar bloka stanja navode se vrednosti koje izlazni signali Murovog tipa imaju za vreme dok je automat u datom stanju. Uobičajeno je da se u bloku stanja ne navode svi signali i njihove vrednosti, već samo spisak imena signala koji su aktivni u konkretnom stanju. Za signale koji su izostavljeni iz spiska, podrazumeva se da su neaktivni. Kao i za dijagram stanja, pod aktivnom vrednošću signala smatra se logičko 1, a pod podrazumevanom logička 0, osim ukoliko nije drugačije rečeno. Na primer, da bi se naglasilo da u datom stanju izlaz z ima vrednost ´1´, dovoljno je u odgovarajućem bloku stanja napisati samo z umesto z<=´1´. Ako automat poseduje još jedan izlaz, y, koji nije naveden u posmatranom bloku stanja, tada u tom stanju važi: y=´0´.



Sl. 6-5 ASM blok.

Blok grananja se crta kao romb sa jednom ulaznom i dve izlazne grane. Unutar romba je zapisan uslov koji određuje koja će od njegove dve izlazne grane biti izabrana. Uslov kombinuje jedan ili više ulaza konačnog automata u logički izraz čija vrednost može biti tačno (´1´ ili ´T´) ili netačno (´0´ ili ´F´). Na primer, w napisano u bloku grananja znači da je odluka zasnovana na vrednosti signala w (ako je w=´1´, nastavljamo putem *tačno* (T), a ako je w=´0´, putem *netačno* (F)). Slično, uslov $w_1 \cdot w_2$ bi značio da će grana *tačno* biti izabrana ako su oba signala jednaka ´1´, tj. w_1=w_2=´1´, a da će u svim ostalim slučajevima

automat slediti granu *netačno*. Osim logičkih, kao uslovi u bloku grananja mogu se koristiti i relacioni izrazi, kao npr. A > B ili C ≠ D (C različito od D).

Blok uslovnog izlaza se crta u vidu zaobljenog pravougaonika i sadrži izlaze Milijevog tipa, odnosno izlaze čije vrednosti zavise kako od tekućeg stanja, tako i od trenutnih vrednosti ulaza automata. Blok uslovnog izlaza uvek sledi posle jednog ili više blokova grananja, a nikad neposredno posle bloka stanja. Kao i u slučaju bloka stanja, u bloku uslovnog izlaza navode se samo aktivni signali. Na primer, da je u bloku grananja iz ASM bloka sa Sl. 6-5 upisano *w*, a u bloku uslovnog izlaza *z*, to bi značilo da će izlaz *z* imati vrednost ´1´ onda kada je automat u posmatranom stanju i pri tom važi *w*=´1´.

Ponašanje sinhronog konačnog automat se najbolje može objasniti kroz opis aktivnosti koje su obuhvaćene jednim ASM blokom:

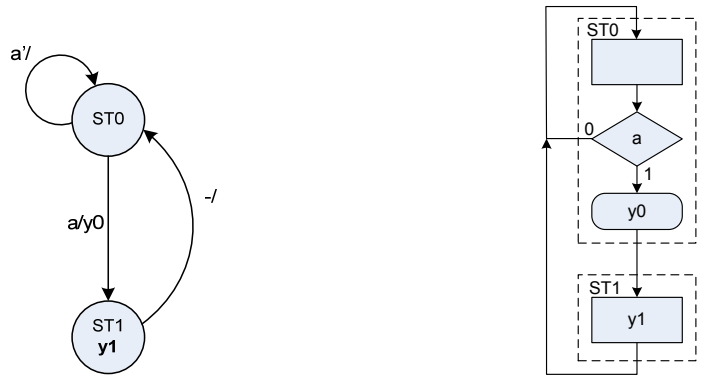
1. Sa svakom rastućom ivicom taktnog signala konačni automat ulazi u novo stanje, tj. u novi ASM blok.
2. U toku trajanja jedne periode takta, konačni automat obavlja nekoliko operacija: postavlja Murove izlaze iz bloka stanja, ispituje logičke uslove iz blokova grananja i shodno ishodu ovih ispitivanja postavlja Milijeve izlaze iz blokova uslovnog izlaza. Pri tom Murovi izlazi zadržavaju istu vrednost tokom celokupnog taktnog perioda, jer zavise samo od stanja, dok se Milijevi mogu menjati pod uticajem eventualni promena ulaza.
3. Sa sledećom rastućom ivicom takta (kojom se završava tekući taktni period), simultano se ispituju logički uslovi iz svih blokova grananja i određuje izlazna grana duž koje će konačni automat preći u novi ASM blok (ili ostati u istom).

Konverzija dijagrama stanja u ASM dijagram. Svaki dijagram stanja se može konvertovati u ekvivalentni ASM dijagram i obratno. Svakom ASM bloku (blok stanja sa pridruženim blokovima grananja i blokovima uslovnog izlaza) iz ASM dijagrama, u dijagramu stanja odgovara jedan čvor zajedno s potezima koji izviru iz tog čvora. Ključ za konverziju je transformacija logičkih izraza sa potega iz dijagrama stanja u blokove grananja u ASM dijagramu. Postupak konverzije biće ilustrovan kroz sledeća četiri primera.

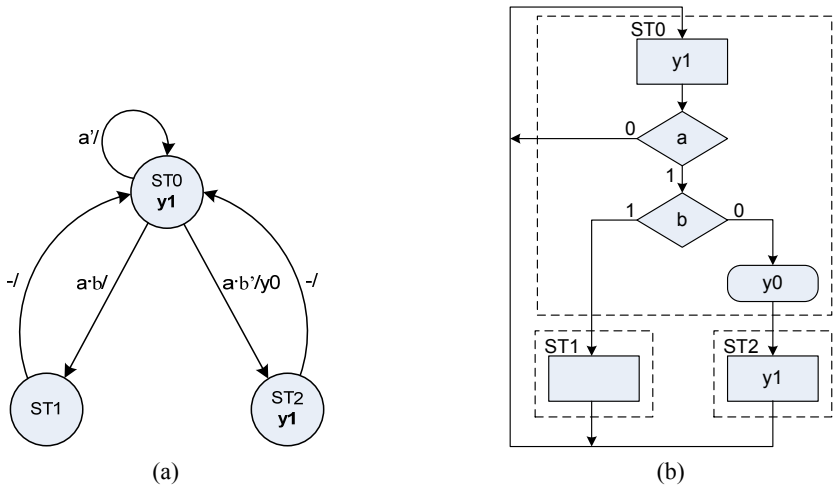
Prvi primer konverzije je prikazan na Sl. 6-6. Radi se o konačnom automatu u kome ne postoje grananja, a dijagram stanja i ASM dijagram su gotovo identični. Automat ″osciluje″ između stanja ST0 i ST1, tako što sa svakom rastućom ivicom takta bezuslovno prelazi iz jednog u ono drugo stanje. Automat poseduje samo jedan i to Murov izlaz, *y*, koji u stanju ST0 ima vrednost ´1´, a u stanju ST1, vrednost ´0´.

(a) (b)

Sl. 6-6 Prvi primer konverzije dijagrama stanja u ASM dijagram: (a) dijagram stanja; (b) ASM dijagram.



Sl. 6-7 Drugi primer konverzije dijagrama stanja u ASM dijagram: (a) dijagram stanja; (b) ASM dijagram.

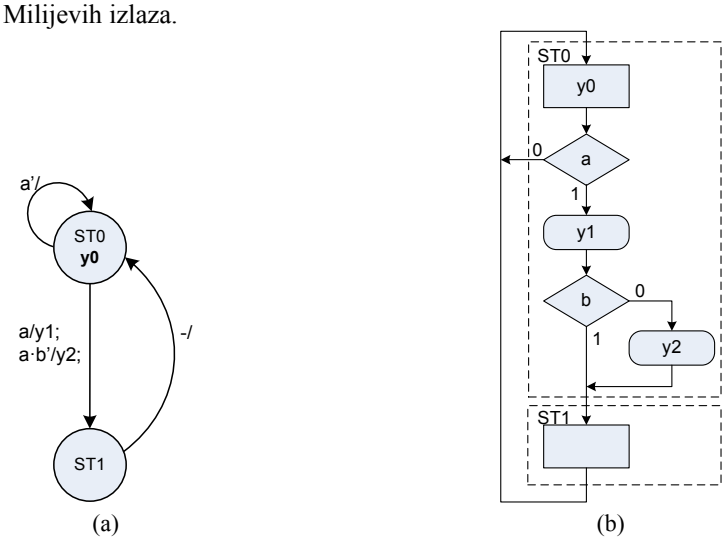


Sl. 6-8 Treći primer konverzije dijagrama stanja u ASM dijagram: (a) dijagram stanja; (b) ASM dijagram.

U konačnom automatu iz drugog primera postoje dva izlazna potega iz stanja ST0, a jednom od njih pridružen je Milijev izlaz (Sl. 6-7). Logički izrazi *a* i *a*' s ovih potega transformišu se u jedan blok grananja koji ispituje vrednost signala *a*. Uočimo da se dva stanja prevode u dva ASM bloka. Napomenimo da blok grananja i blok uslovnog izlaza iz ASM bloka stanja ST0 nisu nova stanja, već su to samo dodatne aktivnosti pridružene ovom stanju.

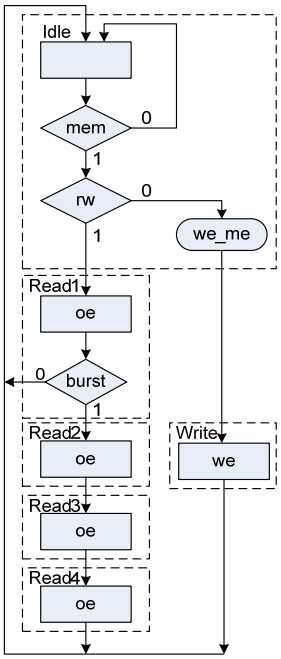
U trećem primeru, iz stanja ST0 izviru tri potega (Sl. 6-8). Logički izrazi s ovih potega, u ASM dijagramu su pokriveni sa dva bloka grananja koji su tako povezani da zajedno imaju tri izlaza od kojih svaki odgovara jednom potegu koja napušta čvor ST0. Automat ostaje u stanju ST0 za a=´1´ **i** b=´1´, a u stanje ST1 prelazi za a=´1´ **i** b=´1´, a u stanje ST2 za a=´1´ i b=´0´. Konačni automat sa Sl. 6-8 je primer automata koji poseduje izlaze oba tipa. Signal $y1$ je Murov izlaz koji je aktivan u stanjima ST0 i ST2, a neaktivan u stanju ST1. Signal $y0$ je Milijev izlaz koji je aktivan u stanju ST0 pod uslovom da važi a=´1´ i b=´0´. U svim ostalim stanjima i pri svim ostalim uslovima, izlaz $y0$ ima vrednost ´0´.

U četvrtom primeru konverzije dijagrama stanja u ASM dijagram (Sl. 6-9), Milijevi izlazi, $y1$ i $y2$, koji su pridruženi potegu iz stanja ST0 u ST1, zavise od različitih uslova. Automat prelazi iz stanja ST0 u stanje ST1 za a=´1´ i pri tom aktivira Miliev izlaz $y1$. Dodatno, ako važi b=´0´, izlaz $y2$ će imati vrednost ´1´. U ovakvim i sličnim situacijama, ASM dijagram predstavlja bolji izbor, jer može na jasniji način da prikaže složenije uslove prezala i postavljanja Milijevih izlaza.



Sl. 6-9 Četvrti primer konverzije dijagrama stanja u ASM dijagram: (a) dijagram stanja; (b) ASM dijagram.

Pr. 6-2 ASM dijagram kontrolera memorije



Sl. 6-10 ASM dijagram kontrolera memorije.

Na Sl. 6-10 je prikazan ASM dijagram koji je funkcionalno ekvivalentan dijagramu stanja kontrolera memorije sa Sl. 6-3(b). Iako to nije obavezno, ASM blokovi iz dijagrama sa Sl. 6-10 su uokvireni, baš kao i ASM blokovi iz prethodnih primera, a iz razloga da bi se na očigledan način obuhvatile sve aktivnosti koje se obavljaju u svakom pojedinačnom stanju. ASM blok stanja *Idle* je funkcionalno najsloženiji. U ovom ASM bloku ispituju se dva ulazna signala, *mem* i *rw*, i postavlja jedan uslovni (Milijev) izlaz, *we_me*. U stanju *Read1*, aktivan je izlazni signal *oe* i dodatno se ispituje ulazni signal *burst* da bi se odredio tip operacije čitanja. U stanjima *Write*, *Read2*, *Read3* i *Read4*, osim postavljanja izlaznog signala *we*, odnosno *oe*, ne obavljaju se nikakve druge aktivnost, a prelaz u sledeće stanje je bezuslovan.

Kao što je već napomenuto, dijagram stanja i ASM dijagram sadrže iste informacije o konačnom automatu. Zahvaljujući blokovima grananja i strukturi koja nalikuju softverskom dijagramu toka, ASM dijagram daje pregledniji prikaz složenih uslova koji regulišu prelaze između stanja i postavljanje Milijevih izlaza. S druge strane, konačni automati sa jednostavnim uslovima prelaza predstavljaju se na pregledniji način pomoću dijagrama stanja.

6.2. Opis konačnog automata u VHDL-u

Postupak kreiranja VHDL opisa konačnih automata sličan je postupku koji se koristi pri pisanju VHDL opisa regularnih sekvencijalnih kola. Slično kao kod regularnih sekvencijalnih kola, neophodni su kôdni segmenti za opis memorijskih elementa (registar stanja), logike sledećeg stanja i logike izlaza. Međutim, postoje sledeće dve bitne razlike. Prvo, kod konačnih automata se koriste simbolička imena stanja, koja se u VHDL-u predstavljaju nabrojivim tipom podataka. Drugo, logika sledećeg stanja konačnog automata se ne može svesti na ″regularnu″ kombinacionu mrežu (inkrementer, pomerač, multiplekser, ...), već se kôd piše u skladu sa dijagramom stanja ili ASM dijagramom.

Za opis konačnih automata u VHDL-u razvijeno je više standardnih pristupa ili kôdnih šablona. U ovom odeljku biće predstavljena dva takva pristupa na primeru konačnog automata kontrolera memorije.

6.2.1. Višesegmentni kôdni šablon

Prvi način za kreiranje VHDL opisa konačnog automata polazi od blok dijagrama sa Sl. 6-1. Svaki blok iz ovog dijagrama opisuje se u vidu zasebnog kôdnog segmenta. Iz perspektive VHDL-a, jasno je da će registar stanja zahtevati proces, dok preostala tri kombinaciona bloka neće. Međutim, kako procesom može biti opisana ne samo sekvencijalna već i kombinaciona logika, svi delovi automata se po pravilu opisuju pomoću zasebnih procesa. Sledi kompletan višesegmentni VHDL opis kontrolera memorije.

```
1 ----------------------------------------------------------------------
2 LIBRARY IEEE;
3 USE IEEE.STD_LOGIC_1164.ALL;
4 ----------------------------------------------------------------------
5 ENTITY mem_ctrl IS
6   PORT (clk, rst : IN STD_LOGIC;
7         mem, rw, burst : IN STD_LOGIC;
8         oe, we, we_me : OUT STD_LOGIC);
```

```
END mem_ctrl;
-------------------------------------------------------------------
ARCHITECTURE multi_seg_arch of mem_ctrl IS
  TYPE state IS (IDLE, READ1, READ2, READ3, READ4, WRITE);
  SIGNAL pr_state, nx_state : state;
BEGIN
  -- registar stanja ----------------------------------------------
  PROCESS(clk, rst)
  BEGIN
    IF(rst = '1') THEN
      pr_state <= IDLE;
    ELSIF( clk'EVENT AND clk = '1') THEN
      pr_state <= nx_state;
    END IF;
  END PROCESS;
  -- logika sledeceg stanja ---------------------------------------
  PROCESS(pr_state, mem, rw, burst)
  BEGIN
  CASE pr_state IS
    WHEN IDLE =>
      IF(mem = '1') THEN
        IF(rw = '1') THEN
          nx_state <= READ1;
        ELSE
          nx_state <= WRITE;
        END IF;
      ELSE
        nx_state <= IDLE;
      END IF;
    WHEN WRITE =>
      nx_state <= IDLE;
    WHEN READ1 =>
      IF(burst = '1') THEN
        nx_state <= READ2;
      ELSE
        nx_state <= IDLE;
      END IF;
    WHEN READ2 =>
      nx_state <= READ3;
    WHEN READ3 =>
      nx_state <= READ4;
    WHEN READ4 =>
      nx_state <= IDLE;
    END CASE;
  END PROCESS;
  -- logika za Murove izlaze --------------------------------------
  PROCESS(pr_state)
  BEGIN
    we <= '0';    -- podrazumevana vrednost
    oe <= '0';    -- podrazumevana vrednost
    CASE pr_state IS
      WHEN IDLE  =>
      WHEN WRITE =>
        we <= '1';
       WHEN READ1 =>
```

```
64      oe <= '1';
65    WHEN READ2 =>
66      oe <= '1';
67    WHEN READ3 =>
68      oe <= '1';
69    WHEN READ4 =>
70      oe <= '1';
71      END CASE;
72    END PROCESS;
73    -- logika za Milijeve izlaze ----------------------------------
74    PROCESS(pr_state, mem, rw)
75    BEGIN
76      we_me <= '0'; -- podrazumevana vrednost
77      CASE pr_state IS
78        WHEN IDLE =>
79          IF(mem = '1') AND (rw = '0') THEN
80            we_me <= '1';
81          END IF;
82        WHEN WRITE =>
83        WHEN READ1 =>
84        WHEN READ2 =>
85        WHEN READ3 =>
86        WHEN READ4 =>
87      END CASE;
88    END PROCESS;
89  END multi_seg_arch;
90  ------------------------------------------------------------------
```

U okviru deklarativnog dela arhitekture definisan je korisnički tip podataka *state* koji sadrži simbolička imena svih stanja konačnog automata (linija 12). Ovaj tip podataka se potom koristi za deklaraciju dva signala: *pr_state* i *nx_state* (linija 13). Signal *pr_state* (od engl. *present state* – tekuće stanje) igraće ulogu registra stanja, a signal *nx_state* (od engl. *next state* – sledeće stanje) ulogu izlaza iz logike sledećeg stanja (v. Sl. 6-1). Telo arhitekture sadrži četiri procesa. Prvi proces opisuje registar stanja (linije 15-23). Kôd nalikuje opisu regularnog registra, s tom razlikom što se za signal *pr_state* koristi korisnički definisan tip podataka, a ne tip *std_logic_vector*. To znači da se ovom signalu mogu dodeljivati samo one vrednosti koje su u vidu simboličkih imena stanja konačnog automata nabrojane u definiciji tipa *state*. Za inicijalizaciju registra stanja koristi se asinhrono resetovanje. Aktiviranjem signala *rst* konačni automata forsirano prelazi u inicijalno stanje, tj. u stanje *idle*.

Sledeći proces (linije 24-53) opisuje logiku sledećeg stanja. Zadatak ovog procesa je da na osnovu tekućeg stanja (*pr_state*) i trenutnih vrednosti ulaza (*mem, rw* i *burst*) odredi novo stanje automata (*nx_state*). Kôd je napisan u saglasnosti sa ASM dijagramom sa Sl. 6-10. Proces sadrži *case* naredbu u kojoj se tekuće stanje automata koristi kao uslov za grananje. Za svako stanje, u naredbi *case* postoji jedna grana u kojoj se nakon eventualnih dodatnih ispitivanja ulaza određuje novo stanje automata i njegovo simboličko ime dodeljuje signalu *nx_state*. Napomenimo da će u trenutku naredne ivice taktnog signala, vrednost signala *nx_state* biti upisana u registar stanja (*pr_state*) i tako postati novo stanje automata. Kôd za svaku granu naredbe *case* direktno sledi na osnovu ASM bloka odgovarajućeg stanja. Kod jednostavnih ASM blokova, kao što je npr. blok za stanje *Read4*, gde postoji samo jedan izlazni poteg, kôd za signal *nx_state* je trivijalan: *nx_state* <= *IDLE* (linija 51). Kod ASM blokova sa više izlaznih potega, za kôdiranje blokova grananja koristi se naredba *if.*

Kaskadno povezani blokovi grananja, poput onih iz ASM bloka za stanje *Idle*, prevode se u ugnježdene *if* naredbe:

```
IF(mem = '1') THEN
  IF(rw = '1') THEN
    nx_state <= READ1;
  ELSE
    nx_state <= WRITE;
  END IF;
ELSE
  nx_state <= IDLE;
END IF;
```

Uočimo da iz ASM blok za stanje *idle* izlaze tri potega i da se zbog toga signalu *nx_state*, a u zavisnosti od ulaza *mem* i *rw*, dodeljuje jedna od tri vrednosti: *read1*, *write* ili *idle*.

Treći proces (linije 54-72) opisuje logiku za Murove izlaze, *we* i *oe*. Pošto izlazni signali Murovog tipa zavise isključivo od tekućeg stanja, *pr_state* je jedini signal u listi senzitivnosti ovog proces. Kao i u prethodnom procesu, kriterijum za grananje u naredbi *case* zasnovan je na vrednosti signal *pr_state*. Uočimo da se na početku procesa signali *we* i *oe* postavljaju na podrazumevu vrednost (linija 57-58):

```
we <= '0';
oe <= '0';
```

Kad u nekom stanju treba aktivirati neki od ova dva izlazna signala, tada se u odgovarajućoj grani *case* naredbe podrazumevana vrednost tog signala prepisuje aktivnom (naredbom *we* <= '1' ili *oe* <= '1').

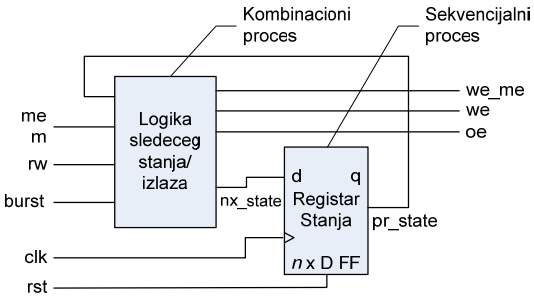
Četvrti proces (linije 73-88) opisuje logiku za Milijeve izlaze (signal *we_me*). Uočimo da se sada u listi senzitivnosti pored signala za tekuće stanje, *pr_state*, nalaze i neki od ulaznih signala. U procesu, naredba *case* služi za grananje na tekuće stanje, a naredba *if* za realizaciju blokova grananja koji vode do blokova uslovnog izlaza u kojima se aktiviraju Milijevi izlazi. U datom primeru, signal *we_me* je aktivan samo u stanju *Idle* i to pod uslovom da važi: *mem*=´1´ i *rw*=´0´. U svim ostalim slučajevima, *we_me* zadržava podrazumevanu vrednost, ´0´, koja mu je dodeljena na početku procesa.

Tri od četiri procesa iz višesegmentne arhitekture VHDL opisa kontrolera memorije, zasnovana su na naredbi *case.* To je učinjeno namerno, da bi se na što jasniji način pokazala veza između ASM dijagrama i VHDL kôda. Međutim, pojedine delove ovog opisa je moguće kôdirati i na kompaktniji način. Na primer, za postavljanje Milijevog izlaza *we_me* dovoljna je samo jedna konkurentna *when* naredbe:

```
we_me <='1' WHEN ((pr_state = IDLE) AND (mem='1') AND (rw='0')) ELSE
        '0';
```

6.2.2. Dvosegmentni kodni šablon

Kod dvosegmentnog načina opisivanja, konačni automat se deli na dva segmenta: sekvencijalni i kombinacioni. Sekvencijalni segment realizuje registar stanja, dok kombinacioni objedinjuje sve kombinacione delove konačnog automata: logiku sledećeg stanja, logiku Murovih i logiku Milijevih izlaza. Na Sl. 6-11 je prikazan blok dijagram kontrolera memorije koji je usklađen sa dvosegmentnim kodnim šablonom.



Sl. 6-11 Blok dijagram dvosegmentnog kontrolera memorije.

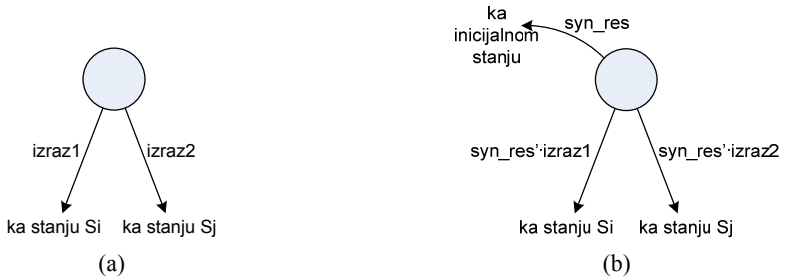
Ispod je dat VHDL opis kontrolera memorije koji je napisan u skladu sa dvosegmentnim kodnim šablonom. Dva kôdna segmenta su realizovana kao dva procesa. Prvi proces (linije 7-14) realizuje registar stanja i identičan je prvom procesu iz višesegmentnog šablona, dok drugi (linije 16-52) objedinjuje funkcije preostala tri kombinaciona procesa iz višesegmentnog opisa. Ovaj proces ima dvostruku namenu: (a) da postavlja Murove i Milijeve izlaze i (b) da određuje sledeće stanje automata. Centralno mesto u kôdu kombinacionog procesa zauzima naredba *case*, čija svaka *when* grana odgovara jednom stanju automata. Za svako stanje, ispituju se vrednosti ulaza i postavljaju odgovarajuće vrednost izlaza i sledećeg stanja, *nx_state*. Naredba *case* iz ovog procesa zapravo predstavlja formalni opis dijagrama stanja: svakom stanju (tj. čvoru iz dijagrama stanja) odgovara jedna *when* grana, a svakom potegu koji izvire iz datog stanja jedna *if* grana u odgovarajućoj *when* grani. Budući da su u listi senzitivnosti navedeni svi ulazni signali i da su sve ulazno-izlazne kombinacije pokrivene kôdom, ovaj proces se sintetiše u kombinacionu logiku.

```
1  ------------------------------------------------------------------
2  ARCHITECTURE two_seg_arch of mem_ctrl IS
3    TYPE state IS (IDLE, READ1, READ2, READ3, READ4, WRITE);
4    SIGNAL pr_state, nx_state : state;
5  BEGIN
6    -- registar stanja ------------------------------------------
7    PROCESS(clk, rst)
8    BEGIN
9      IF(rst = '1') THEN
10       pr_state <= IDLE;
11     ELSIF( clk'EVENT AND clk = '1') THEN
12       pr_state <= nx_state;
13     END IF;
14   END PROCESS;
15   -- logika sledeceg stanja/izlaza ----------------------------
16   PROCESS(pr_state, mem, rw, burst)
17   BEGIN
18     oe <= '0';we <= '0';
19     we_me <= '0';
20     CASE pr_state IS
21       WHEN IDLE =>
22         IF(mem = '1') THEN
23           IF(rw = '1') THEN
24             nx_state <= READ1;
25           ELSE
26             nx_state <= WRITE;
27             we_me <= '1';
```

```
28            END IF;
29          ELSE
30            nx_state <= IDLE;
31          END IF;
32        WHEN WRITE =>
33          nx_state <= IDLE;
34          we <= '1';
35        WHEN READ1 =>
36          IF(burst = '1') THEN
37           nx_state <= READ2;
38          ELSE
39           nx_state <= IDLE;
40          END IF;
41          oe <= '1';
42        WHEN READ2 =>
43          nx_state <= READ3;
44          oe <= '1';
45        WHEN READ3 =>
46          nx_state <= READ4;
47          oe <= '1';
48        WHEN READ4 =>
49          nx_state <= IDLE;
50          oe <= '1';
51        END CASE;
52    END PROCESS;
53 END two_seg_arch;
54 ------------------------------------------------------------------
```

6.2.3. Sinhrona inicijalizacija konačnog automata

U oba prethodno predstavljena kodna šablona, inicijalizacija automata se obavlja pomoću signala za asinhrono resetovanje registra stanja. Pod dejstvom ovog signala, u registar stanja se asinhrono, tj. nezavisno od taktnog signala, upisuje kôd inicijalnog (početnog) stanja. Međutim, u nekim situacijama je bolje, ili se to eksplicitno zahteva, da se inicijalizacija konačnog automata obavi sinhrono. To se postiže uvođenjem signala za sinhronu inicijalizaciju, ili sinhrono resetovanje, *syn_rst*. Ovaj signal se tretira kao bilo koji drugi ulazni signal automata, a postavljen na ´1´ inicira prelaz automata iz bilo kog stanja u inicijalno stanje. U kontekstu dijagrama stanja, uvođenje sinhrone inicijalizacije podrazumeva da se za svako stanje veže po jedna dodatna grana sa uslovom *syn_rst*=´1´ koja će biti usmerena ka inicijalnom stanju. Pri tom uslove svih regularnih grana treba proširiti članom *syn_rst'*, kao što je ilustrovano na Sl. 6-12.



Sl. 6-12 Uvođenje sinhrone inicijalizacije u dijagram stanja: (a) polazno stanje; (b) modifikovano stanje, sa sinhronim resetom.

Iako uvođenje sinhrone inicijalizacije značajno komplikuje dijagram stanja ili ASM dijagram, odgovarajuća intervencija na nivou VHDL opisa je gotovo trivijalna. Dovoljno je modifikovati samo proces registra stanja, tako da se resetovanje registra obavlja sinhrono, a ne asinhrono:

```
-- registar stanja ------------------------------------------------
PROCESS(clk)
BEGIN
  IF(clk'EVENT AND clk = '1') THEN
    IF(syn_rst = '1')
      pr_state <= IDLE;
    ELSE
      pr_state <= nx_state;
    END IF;
  END IF;
END PROCESS
```

6.3. Kodiranje stanja konačnog automata

Kodiranje stanja predstavlja obavezan korak u procesu sinteze konačnih automata. Cilj ovog koraka je zamena simboličkih imena stanja binarnim kombinacijama. Pri tom ne postoji neko striktno ograničenje koje se tiče dodele binarnih kodova stanjima osim zahteva da svako stanje treba da bude kodirano jedinstvenom binarnom kombinacijom. Zbog toga, broj različitih kodiranja stanja jednog istog automata može biti veoma veliki. Na primer, stanja konačnog automata sa četiri stanja S0, S1, S2 i S3 mogu se kodirati sa dva bita, odnosno binarnim kombinacijama: 00, 01, 10 ili 11. U ovom slučaju, postoje 4!=24 različita kodiranja, kao što je prikazano u tabeli T. 6-1. Iako funkcionalno ekvivalentna, različita kodiranja nisu ravnopravna u pogledu sinteze, jer promenom kodiranja dolazi i do promene funkcije kombinacione logike konačnog automata, koja za jedno kodiranje može biti složenija, a za neko drugo jednostavnija.

T. 6-1 Alternativna kondiranje stanja automata sa 4 stanja.

#	S0	S1	S2	S3		#	S0	S1	S2	S3		#	S0	S1	S2	S3
1	00	01	10	11		9	01	10	00	11		17	10	11	00	01
2	00	01	11	10		10	01	10	11	00		18	10	11	01	00
3	00	10	01	11		11	01	11	00	10		19	11	00	01	10
4	00	10	11	01		12	01	11	10	00		20	11	00	10	01
5	00	11	00	10		13	10	00	01	11		21	11	01	00	10
6	00	11	10	01		14	10	00	11	01		22	11	01	10	00
7	01	00	10	11		15	10	01	00	11		23	11	10	00	01
8	01	00	11	10		16	10	01	11	00		24	11	10	01	00

Pronalaženje optimalnog koda, tj. kodiranja pri kome će složenost kombinacione logike konačnog automata biti najmanja, predstavlja veoma težak problem. Zbog toga se prilikom sinteze konačnog automata projektanti obično odlučuju za jednu od nekoliko standardnih šema (ili heuristika) za kodiranja stanja. U suštini, kodiranje stanja konačnog automata može se razložiti na dva potproblema. Prvi se odnosi na izbor broja bita, *n*, koji će biti korišćeni za kodiranje, a drugi na pridruživanje *n*-bitnih kodova stanjima. Neke od često korišćenih šema kodiranja su:

- *Binarno kodiranje* – koristi se minimalan broja bita za kodiranje, koji za konačni automat sa *N* stanja iznosi $n = \lceil \log_2 N \rceil$. Na primer, za 4 stanja potrebna su dva, a za

pet tri bita. Pri tom redosled po kome se stanjima pridružuju *n*-bitni kodovi može biti proizvoljan. Ovaj pristup garantuje da će broj flip-flopova u sintetizovanom hardveru biti najmanji mogući, ali će zato kombinaciona logika, po pravilu, biti nešto složenija.

- *˝Onehot˝ kodiranje* - broj bita za kodiranje jednak je broju stanja (*n*=*N*). Svakom stanju se dodeljuje jedan bit iz *N* bitne binarne kombinacije koji kada je jednak 1 ukazuje da je konačni automat u tom stanju. Na primer, za kodiranje automat sa četiri stanja, koriste se sledeće četiri binarne kombinacije: 0001, 0010, 0100 i 1000. *Onehot* kodiranje zahteva veliki broj flip-flopova i zbog toga nije praktično za konačne automate sa velikim brojem stanja, ali je zato, s druge strane, kombinaciona logika rezultujuće hardverske realizacije obično značajno jednostavnija i brža nego kad se koristi binarnog kodiranja. *Onehot* kodiranje se preporučuje u slučajevima kad je broj stanja relativno mali, a za implementaciju se koristi tehnologija koja je ˝bogata˝ flip-flopovima, kao što je FPGA.
- *Grejov (Gary) kôd* – koristi se minimalan broj bita za kodiranje, kao kod binarnog kodiranja, ali je zato pravilo za dodelu kodova stanjima složenije, jer se teži da se susednim stanjima (tj. povezanim stanjima) dodeljuju binarni kodovi koje se razlikuju na što manjem broju bitskih pozicija. Na taj način se, indirektno, smanjuje složenost logike sledećeg stanja.

U svim prethodnim primerima VHDL opisa konačnih automata, stanja su kodirana na simbolički način, pomoću nabrojivog tipa podataka koji definiše imena stanja, ali ne i njihove binarne kodove. Ovakav stil modeliranja konačnih automata se oslanja na podršku alata za sintezu koji će prilikom kompajliranja VHDL kôda simbolička imena stanja zameniti binarnim kodovima u skladu sa izabranom šemom kodiranja. Na primer, u CAD sistemu ISE, koji se koristi za projektovanje na Xilinx FPGA platformama, heuristika kodiranja stanja se zadaje pomoću atributa koji se pridružuje nabrojivom tipu podataka za stanja. Ovaj atribut se definiše u deklarativnom delu arhitekture posle deklaracije nabrojivog tipa:

```
. . .
ARCHITECTURE <ime_arhitekture> OF <ime_entiteta> IS
   TYPE state IS (state_0,state_1,state_2 ...);
   attribute fsm_encoding : string;
   attribute fsm_encoding of state is "gray"
   SIGNAL pr_state,nx_state : state;
BEGIN . . .
```

Prva od dve linije kôda posvećenih specifikaciji heuristike kodiranja stanja deklariše atribut imena *fsm_enciding*, a druga pridružuje ovaj atribut tipu podataka *state* i dodeljuje mu vrednost ˝gray˝, koja odgovara Grejovom kodu. Ako se u VHDL kôdu heuristika eksplicitno ne navede, alat će koristi podrazumevanu opciju (kod Xilinx ISE to je *onehot*). Način specifikacije heuristike kodiranja može biti drugačiji za neki drugi CAD sistem. Ako projektant želi da samostalno kodira stanja konačnog automata, tada se binarne kombinacije pridružuju stanjima takođe pomoću atributa koji se vezuje za nabrojivi tip podataka *state*, na sledeći način:

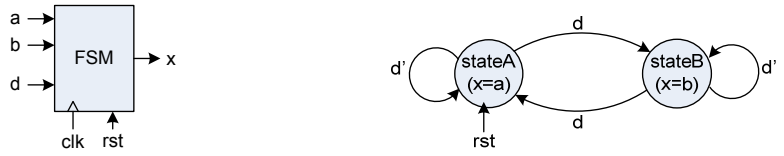
```
. . .
ARCHITECTURE FSM OF detektor_bit_oblika IS
   TYPE state IS (S0, S1, S2, S3);
   attribute enum_encoding of state: type is "00 10 11 01";
   SIGNAL pr_state, nx_state : state;
BEGIN . . .
```

String dodeljen atributu *enum_encoding* sadrži niz od četiri 2-bitne binarne kombinacije koje se redom dodeljuju stanjima S0, S1, S2 i S3. Napomenimo da ovakav način ručnog kodiranja važi za Xilinx ISE sistem.

6.4. Primeri konačnih automata

Pr. 6-3 Jednostavan konačni automat

Sl. 6-13 prikazuje blok dijagram i dijagram stanja jednog jednostavnog konačnog automata Murovog tipa. Konačni automat ima tri ulaza, *a, b* i *d*, jedan izlaz, *x*, i dva stanja, *stateA* i *stateB*. Do promene stanja dolazi uvek kad je *d*=´1´. Konačni automat ostaje u zatečenom stanju za sve vreme dok važi *d*=´0´. U stanju *stateA* na izlaz *x* se prenosi vrednost ulaza *a* (*x*=*a*), a u stanju *stateB* vrednost ulaza *b* (*x*=*b*). Početno stanje konačnog automata je stanje *stateA*. Sledi VHDL opis konačnog automata sa Sl. 6-13. Ovaj opis je usklađen sa dvosegmentnim kôdnim šablonom.



(a) (b)

Sl. 6-13 Konačni automat iz Pr. 6-3: (a) blok dijagram; (b) dijagram stanja.

```
1  ----------------------------------------------------------------------
2  ENTITY fsm_1 IS
3    PORT (a,b,c,d,clk,rst : IN BIT;
4          x: OUT BIT);
5  END fsm_1;
6  ----------------------------------------------------------------------
7  ARCHITECTURE fsm_1 OF fsm_1 IS
8    TYPE state IS (stateA, stateB);
9    SIGNAL pr_state, nx_state : state;
10 BEGIN
11   -- registar stanja ------------------------------------------------
12   PROCESS (clk,rst)
13   BEGIN
14      IF(rst='1')THEN
15         pr_state <=stateA;
16      ELSIF(clk'EVENT AND clk='1') THEN
17         pr_state <= nx_state;
18      END IF;
19   END PROCESS;
20   -- kombinacioni deo -----------------------------------------------
21   PROCESS(a,b,d,pr_state)
22   BEGIN
23      CASE pr_state IS
24         WHEN stateA =>
25            x<=a;
26            IF(d='1')THEN
27               nx_state <= stateB;
28            ELSE
29               nx_state <= stateA;
```

```
30            END IF;
31         WHEN stateB =>
32            x<=b;
33            IF(d = '1')THEN
34               nx_state <= stateA;
35            ELSE
36               nx_state <= stateB;
37            END IF;
38      END CASE;
39   END PROCESS;
40 END fsm_1;
```

Pr. 6-4 Detektor ivice

Sledi VHDL opis detektora ivice shodno dijagramu stanja sa Sl. 6-4(a). Kôd je napisan shodno višesegmentnom stilu.

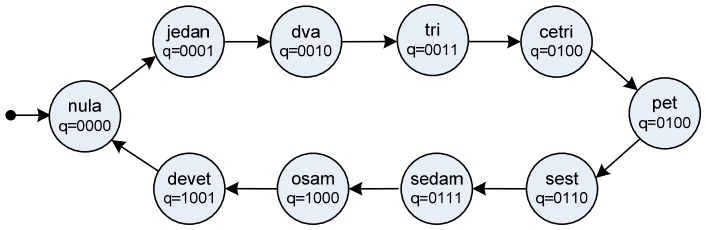
```
1  ----------------------------------------------------------------------
2  LIBRARY IEEE;
3  USE IEEE.STD_LOGIC_1164.ALL;
4  ----------------------------------------------------------------------
5  ENTITY edge_detector IS
6     PORT (clk, rst: IN STD_LOGIC;
7           strobe: IN STD_LOGIC;
8           p: OUT STD_LOGIC);
9  END edge_detector;
10 ----------------------------------------------------------------------
11 ARCHITECTURE moore_arch of edge_detector IS
12    TYPE state IS (zero, edge, one);
13    SIGNAL pr_state, nx_state : state;
14 BEGIN
15    -- registar stanja ------------------------------------------
16    PROCESS(clk, rst)
17    BEGIN
18       IF(rest = '1') THEN
19          pr_state <= zero;
20       ELSIF(clk'EVENT AND clk = '1') THEN
21          pr_state <= nx_state;
22       END IF;
23    END PROCESS;
24    -- logika sledeceg stanja -----------------------------------
25    PROCESS(pr_state, strobe)
26    BEGIN
27       CASE pr_state IS
28          WHEN zero =>
29             IF strobe = '1' THEN
30                nx_state <= edge;
31             ELSE
32                nx_state <= zero;
33             END IF;
34          WHEN edge =>
35             IF strobe = '1' THEN
36                nx_state <= one;
37             ELSE
38                nx_state <= zero;
```

```
39              END IF;
40           WHEN one =>
41              IF strobe = '1' THEN
42                 nx_state <= one;
43              ELSE
44                 nx_state <= zero;
45              END IF;
46        END CASE;
47     END PROCESS;
48     -- Murova izlazna logika ------------------------------------
49     p <= '1' WHEN pr_state = edge ELSE
50          '0';
51  END moore_arch;
```

Pr. 6-5 BCD brojač

Ponašanje bilo kog sekvencijalnog kola može se opisati u vidu konačnog automata. Međutim, konačni automat ne mora biti uvek i najbolji (najjednostavniji) način za modeliranje sekvencijalnih kola. To je obično slučaj sa regularnim sekvencijalnim komponentama, kao što su to npr. brojači i pomerački registar. U ovom primeru biće predstavljen VHDL opis BCD brojača koji je kreiran na osnovu predstave brojač u vidu konačnog automata Murovog tipa.

BCD brojač je 4-bitni binarni brojač sa osnovom brojanja 10. Svako stanje ovog brojača se interpretira kao jedna binarno-kodirana decimalna cifra (0, 1, .., 9), pa otuda i naziv BCD (prema engl. ***B**inary **C**oded **D**ecimal*) brojač. Na Sl. 6-14 je prikazan dijagram stanja BCD brojača. Konačni automat sadrži 10 stanja koja su povezana u cikličnu sekvencu. Vrednosti 4-bitnog izlaza *q* koje su pridružene stanjima odgovaraju BCD ciframa. Početno stanje brojača, označeno strelicom, je stanje *nula*. Posle stanja *nula* slede redom stanja od *jedan* do *devet*, a onda ponovo *nula*.



Sl. 6-14 Dijagram stanja BCD brojača

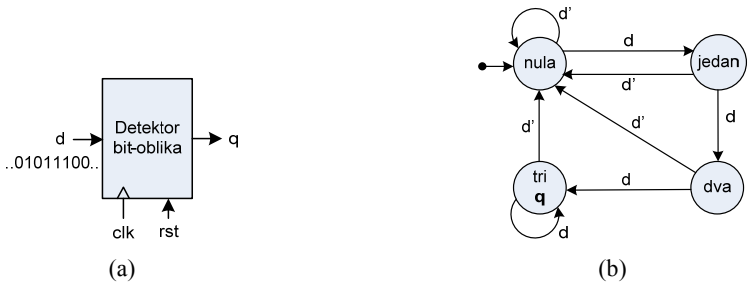
VHDL kôd je napisan u skladu sa dvosegmentnim kôdnim šablonom. Obimnost procesa za kombinacioni deo automata potiče od zahteva da se svako stanje mora eksplicitno obraditi u *case* naredbi. Očigledno, ovakav opisa brojača biće obimniji što je broj stanja veći. Kompaktniji i prirodniji opis brojača je svakako onaj koji je zasnovan na aritmetičkoj operaciji sabiranja, ″+″, poput opisa binarnog brojača iz Pr. 5-19.

```
1  ----------------------------------------------------------------
2  LIBRARY IEEE;
3  USE IEEE.STD_LOGIC_1164.ALL;
4  ----------------------------------------------------------------
5  ENTITY BCD IS
6    PORT (clk,rst : IN STD_LOGIC;
```

```
7         q : OUT STD_LOGIC_VECTOR(3 DOWNTO 0));
8  END BCD;
9  ------------------------------------------------------------------
10 ARCHITECTURE FSM OF BCD IS
11   TYPE state IS (nula, jedan, dva, tri, cetiri,
12                  pet, sest, sedam, osam, devet);
13   SIGNAL pr_state, nx_state : state;
14 BEGIN
15   -- registar stanja ---------------------------------------------
16   PROCESS (clk,rst)
17   BEGIN
18      IF(rst='1')THEN
19         pr_state <=nula;
20      ELSIF(clk'EVENT AND clk='1') THEN
21         pr_state <= nx_state;
22      END IF;
23   END PROCESS;
24   -- kombinacioni deo --------------------------------------------
25   PROCESS(pr_state)
26   BEGIN
27      CASE pr_state IS
28         WHEN nula =>
29            q <= "0000";
30            nx_state <= jedan;
31         WHEN jedan =>
32            q <= "0001";
33            nx_state <= dva;
34         WHEN dva =>
35            q <= "0010";
36            nx_state <= tri;
37         WHEN tri =>
38            q <= "0011";
39            nx_state <= cetiri;
40         WHEN cetiri =>
41            q <= "0100";
42            nx_state <= pet;
43         WHEN pet =>
44            q <= "0101";
45            nx_state <= sest;
46         WHEN sest =>
47            q <= "0110";
48            nx_state <= sedam;
49         WHEN sedam =>
50            q <= "0111";
51            nx_state <= osam;
52         WHEN osam =>
53            q <= "1000";
54            nx_state <= devet;
55         WHEN devet =>
56            q <= "1001";
57            nx_state <= nula;
58      END CASE;
59   END PROCESS;
60 END FSM;
```

Pr. 6-6 Detektor bit-oblika

Detektor bit-oblika je sekvencijalno kolo koje prepoznaje zadatu podsekvencu (tj. bit-oblik) u ulaznoj binarnoj sekvenci. U ovom primeru je predstavljen detektor bit-oblika ″111″, tj. kolo koje na svom izlazu generiše ´1´ uvek kada na ulazu dobije tri uzastopne jedinice. Pri tom preklapanje bit-oblika se uzima u obzir; na primer, ako se na ulazu javi sekvenca ″…0111110…″, na izlazu detektora će biti postavljeno ´1´ u tri uzastopna taktna ciklusa.



Sl. 6-15 Detektor bit-oblika ″111″.

Sl. 6-15 prikazuje blok dijagram i dijagram stanja detektora bit-oblika. Na ulaz *d* se dovodi novi bit ulazne sekvence u svakom taktnom ciklusu. Kolo poseduje četiri stanja koja su nazvana tako da njihova imena ukazuju na trenutni broj detektovanih uzastopnih 1-ca. Radi se o konačnom automatu Murovog tipa čiji izlaz *q* ima vrednost ´1´ u stanju *tri* (primljene su tri uzastopne jedinice), a vrednost ´0´ u svim ostalim stanjima. Svako novo 1 na ulazu *d* vodi automat u sledeće stanje, sve do stanja *tri* u kome kolo ostaje sve dok su na ulazu prisutne 1-ce. Pojava nule na ulazu, u bilo kom stanju, vraća automat u stanje *nula*.

U nastavku su data dva VHDL opisa detektora bit-oblika. Oba rešenja su napisana u skladu sa dvosegmentnim kôdnim šablonom, a razlikuju se po načinu pisanja kôda za kombinacioni proces. U prvom rešenju (arhitektura *arch_v1*), za svako stanje se u odgovarajućoj *when* grani naredbe *case* (linije 26-47) najpre izlazu *q* dodeljuje vrednost koju on treba da ima u tom stanju, a zatim se, u zavisnosti od vrednosti ulaza *d*, određuje sledeće stanje. Drugo rešenje (arhitektura *arch_v2*) je kompaktnije jer koristi podrazumevane vrednosti za izlaz *q* i sledeće stanje, *nx_state*. Na početku kombinacionog procesa, pre *case* naredbe, u linijama 67 i 68, signalima *q* i *nx_state* dodeljene su podrazumevane vrednosti, tj. ´0´ za *q* i *nula* za *nx_state*. U naredbi *case* (linije 69-83) signalima *q* i *nx_state* se dodeljuje vrednost samo ako se njihova nova vrednost razlikuje od podrazumevane. Kao podrazumevane vrednosti izabrane su one koje se najčešće dodeljuju signalima *q* i *nx_state*.

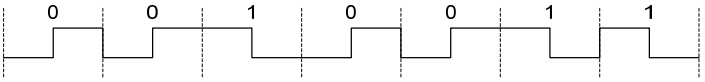
```
1  ----------------------------------------------------------------
2  LIBRARY IEEE;
3  USE ieee.STD_LOGIC_1164.ALL;
4  ----------------------------------------------------------------
5  ENTITY detektor_bit_oblika IS
6    PORT (d,clk,rst : IN STD_LOGIC;
7          q: OUT STD_LOGIC);
8  END detektor_bit_oblika;
9  -- Resenje 1 -----------------------------------------------------
10 ARCHITECTURE arch_v1 OF detektor_bit_oblika IS
11   TYPE state IS (nula, jedan, dva, tri);
12   SIGNAL pr_state, nx_state : state;
```

```
BEGIN
  -- sekvencijalni deo ------------------------------------------
  PROCESS (clk,rst)
  BEGIN
     IF(rst='1')THEN
        pr_state <=nula;
     ELSIF(clk'EVENT AND clk='1') THEN
        pr_state <= nx_state;
     END IF;
  END PROCESS;
  -- kombinacioni deo -------------------------------------------
  PROCESS(d,pr_state)
  BEGIN
     CASE pr_state IS
        WHEN nula =>
           q<='0';
           IF(d='1')THEN nx_state <= jedan;
           ELSE nx_state <= nula;
           END IF;
        WHEN jedan =>
           q<='0';
           IF(d='1')THEN nx_state <= dva;
           ELSE nx_state <= nula;
           END IF;
        WHEN dva =>
           q<='0';
           IF(d='1')THEN nx_state <= tri;
           ELSE nx_state <= nula;
           END IF;
        WHEN tri =>
           q<='1';
           IF(d='0')THEN nx_state <= nula;
           ELSE  nx_state <= tri;
           END IF;
     END CASE;
  END PROCESS;
END arch_v1;
-- Resenje 2 ---------------------------------------------------
ARCHITECTURE arch_v2 OF detektor_bit_oblika IS
  TYPE state IS (nula, jedan, dva, tri);
  SIGNAL pr_state, nx_state : state;
BEGIN
  -- sekvencijalni deo ------------------------------------------
  PROCESS (clk,rst)
  BEGIN
     IF(rst='1')THEN
        pr_state <=nula;
     ELSIF(clk'EVENT AND clk='1') THEN
        pr_state <= nx_state;
     END IF;
  END PROCESS;
  -- kombinacioni deo -------------------------------------------
  PROCESS(d,pr_state)
  BEGIN
     q<='0';             -- podrazumevane
```

```
68      nx_state <= nula;   -- vrednosti
69      CASE pr_state IS
70         WHEN nula =>
71            IF(d='1')THEN nx_state <= jedan;
72            END IF;
73         WHEN jedan =>
74            IF(d='1')THEN nx_state <= dva;
75            END IF;
76         WHEN dva =>
77            IF(d='1')THEN nx_state <= tri;
78            END IF;
79         WHEN tri =>
80            q<='1';
81            IF(d='0')THEN nx_state <= nula;
82            END IF;
83      END CASE;
84   END PROCESS;
85 END arch_v2;
```

Pr. 6-7 Koder za Mančester kôd

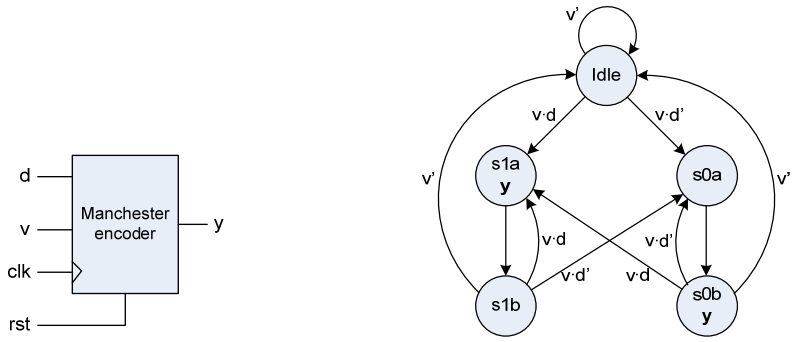
Mančester kôd se često koristi za kodiranje binarnih sekvenci za prenos preko serijske komunikacione linije. U Mančester kôdu, binarna 0 se predstavlja prelazom ´0´→´1´, odnosno sa ´0´ u prvoj i ´1´ u drugoj polovini bitskog intervala. Slično, binarno 1 se predstavlja prelazom ´1´→´0´, tj. sa ´1´ u prvoj i ´0´ u drugoj polovini bitskog intervala. Na Sl. 6-16 je prikazano kako se binarna sekvenca ″0010011″ kodira u Mančester kôdu.



Sl. 6-16 Primer talasnog oblika mančester-kodirane bitske sekvence.

Mančester koder je kolo koje transformiše niz (povorku) bitova u Mančester-kôdiranu bitsku sekvencu (Sl. 6-17(a)). Koder poseduje dva ulazna, *d* i *v*, i jedan izlazni, *y*, signal. Signal *d* predstavlja ulaznu bitsku sekvencu, tj. sekvencu koja se kodira, dok je *v* signal za dozvolu rada kodera. Za *v*=´1´, koder ″radi″ i konvertuje signal *d* u Mančester kôd. U suprotnom slučaju, za *v*=´0´, koder ″ne radi″, a na izlazu *y* je permanentno prisutna ´0´. Budući da se jedan ulazni bit konvertuje u dva bita na izlazu, ″01″ ili ″10″, postoji zahtev da svaki bit ulazne sekvence treba da traje tačno dva taktna intervala.

Na Sl. 6-17(b) je prikazan dijagram stanja Mančester kodera. Sve dok važi *v*=´0´, automat je ″zakočen″ u stanju *idle*, a na njegovom izlazu je *y*=´0´. Transformacije ulazne u izlaznu bitsku sekvencu počinje onda kad *v* postane ´1´ i traje sve dok se *v* ponovo ne vrati na ´0´. U toku kodiranja, a u zavisnosti od vrednosti bita koji se kodira, automat prolazi kroz stanja *s1a* i *s1b*, za *d*=´1´, odnosno kroz stanja *s0a* i *s0b*, za *d*=´0´, generišući tako na izlazu *y* sekvencu ″10″, odnosno ″01″. U stanjima *s1b* i *s0b*, ispituje se ulazni signal *v* i ako je njegova vrednost još uvek ´1´, koder, bez povratka u stanje *idle*, prelazi na kodiranje sledećeg ulaznog bita. Sledi kompletan VHDL opis mančester kodera.



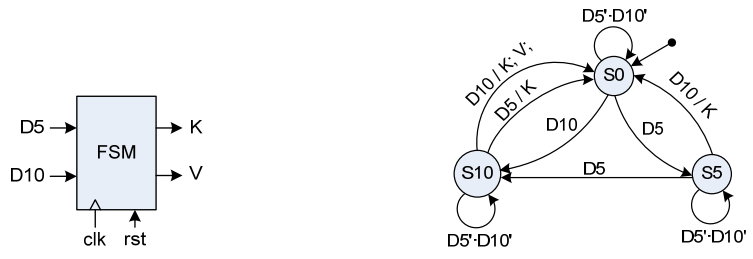
Sl. 6-17 Mančester koder: (a) blok dijagram; (b) dijagram stanja.

```
1  ----------------------------------------------------------------------
2  LIBRARY IEEE;
3  USE IEEE.STD_LOGIC_1164.ALL;
4  ----------------------------------------------------------------------
5  ENTITY manchester_encoder IS
6     PORT (clk, rst: IN STD_LOGIC;
7           d, v: IN STD_LOGIC;
8           y: OUT STD_LOGIC);
9  END manchester_encoder;
10 ----------------------------------------------------------------------
11 ARCHITECTURE moore_arch of manchester_encoder IS
12    TYPE state IS (idle, s1a, s1b, s0a, s0b);
13    SIGNAL pr_state, nx_state : state;
14 BEGIN
15    -- registar stanja ------------------------------------------
16    PROCESS(clk, rst)
17    BEGIN
18       IF(rst = '1') THEN
19          pr_state <= idle;
20       ELSIF(clk'EVENT AND clk = '1') THEN
21          pr_state <= nx_state;
22       END IF;
23    END PROCESS;
24    -- logika sledeceg stanja -----------------------------------
25    PROCESS(pr_state, v, d)
26    BEGIN
27       CASE pr_state IS
28          WHEN idle =>
29             IF v = '0' THEN
30                nx_state <= idle;
31             ELSE
32                IF d = '0' THEN
33                   nx_state <= s0a;
34                ELSE
35                   nx_state <= s1a;
36                END IF;
37             END IF;
38          WHEN s0a =>
39             nx_state <= s0b;
40          WHEN s1a =>
```

```
41              nx_state <= s1b;
42           WHEN s0b =>
43              IF v = '0' THEN
44                 nx_state <= idle;
45              ELSE
46                 IF d = '0' THEN
47                    nx_state <= s0a;
48                 ELSE
49                    nx_state <= s1a;
50                 END IF;
51              END IF;
52           WHEN s1b =>
53              IF v = '0' THEN
54                 nx_state <= idle;
55              ELSE
56                 IF d = '0' THEN
57                    nx_state <= s0a;
58                 ELSE
59                    nx_state <= s1a;
60                 END IF;
61              END IF;
62        END CASE;
63     END PROCESS;
64     -- Murova izlazna logika ---------------------------------------
65     y <= '1' WHEN (pr_state = s1a OR pr_state = s0b) ELSE
66          '0';
67  END moore_arch;
```

Pr. 6-8 Automat za prodaju karata

U ovom primeru, projektujemo upravljačku jedinicu automata za prodaju karata u gradskom autobusu. Cena karte je 15 din., a u automat se ubacuju kovanice od 5 i 10 din. Kad iznos ubačenog novca postane jednak ili veći od cene karte, automat izdaje kartu i ako je potrebno vraća kusur. Na Sl. 6-18(a) je prikazan blok dijagram upravljačke jedinice jednog ovakvog uređaju. Osim upravljačke jedinice, uređaj poseduje i elektromehanički sklop koji prihvata ubačene kovanice, izdaje kartu i vraća kusur. Pretpostavka je da ovaj sklop generiše impuls trajanja jednog taktnog ciklusa za svaku ubačenu kovanicu od 5 din. (signal *D*5) i 10 din. (signal *D*10). Takođe, elektromehanički sklop reaguje na impulse *K* i *V*, trajanja jednog taktnog ciklusa, tako što izdaje kartu (signal *K*), odnosno vraća kusur od 5 din. (signal *V*).



(a) (b)

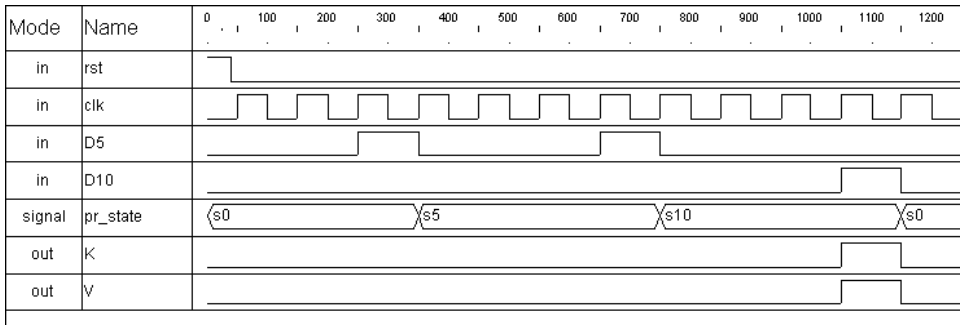
Sl. 6-18 Automat za prodaju karata: (a) blok dijagram; (b) dijagram stanja.

Na Sl. 6-18(b) je prikazan dijagram stanja konačnog automata kojim je modelirano ponašanje upravljačke jedinice sa Sl. 6-18(a). Kao što se može videti, radi se o automatu Milijevog tipa, jer su izlazi *K* i *V* pridruženi prelazima, a ne stanjima. Svako stanje odgovara ukupnom iznosu do tog momenta ubačenog novca: S0 je početno stanje, koje važi sve dok se ne ubaci prva kovanica (odnosno ubačeno je 0 din.). Slično, S5 znači da je ubačeno 5 din., a stanje S10 da je ubačeno ukupno 10 din. Automat ostaje u zatečenom stanju sve dok su oba ulaza, *D*5 i *D*10, jednaki ′0′. Ovaj uslov je označen logičkim izrazom *D*5'·*D*10' na petljama koje postoje u svakom stanju. Iz stanja S0 automat prelazi u stanje S5 ako je kao prva ubačena kovanica od 5 din., odnosno prelazi u stanje S10 ako je ubačena kovanica od 10 din. Iz stanja S5, automat prelazi u stanje S10 ako je kao druga ubačena kovanica od 5 din (uslov *D5*=′1′). Međutim, ako je u stanju S5 ubačena kovanica od 10 din. (*D10*=′1′), ukupan iznos ubačenog novca postaje jednak ceni karte i automat se vraća u početno stanje, uz aktiviranje signal za izdavanje karte, *K*. Pošto nema potrebe za vraćanjem kusura, signal *V* ostaje neaktivan. Nakon ubacivanja kovanice u stanju S10 takođe su moguća dva ishoda, pri čemu oba rezultuju u izdavanju karte (*K*=′1′). Pri tome, ako je ubačena kovanica od 5 din., kusur se ne vraća (*V*=′0′), a ako je ubačeno 10 din. kusur se vraća (*V*=′1′), jer je vrednost ubačenog novca (20 din.) postala veća od cene karte (15 din.).

VHDL opis je usklađen sa dvosegmentnim kôdnim šablonom, a razlika u odnosu na opis konačnog automata Murovog tipa ogleda se u tome što naredbe za dodelu vrednosti izlaznim signalima iz *when* grana naredbe *case* nisu bezuslovne, već su deo *if* naredbi koje ispituje ulaze. Na Sl. 6-19 su prikazani rezultati simulacije datog VHDL kôda za slučaj kada se u automat ubacuju redom: 5 din., 5 din. i 10 din. Pošto je ubačeno ukupno 20 din., automat vraća kusur. Uočimo da impuls koji se javlja na ulaznom signalu *D10* trenutno postavlja izlaze *V* i *K*, dok se stanje automata menja tek sa sledećom aktivnom ivicom takta (odlika konačnih automata Milijevog tipa).

```
1  ------------------------------------------------------------------
2  LIBRARY IEEE;
3  USE IEEE.STD_LOGIC_1164.ALL;
4  ------------------------------------------------------------------
5  ENTITY controller IS
6    PORT (D5,D10,clk,rst : IN STD_LOGIC;
7          K,V: OUT STD_LOGIC);
8  END controller;
9  ------------------------------------------------------------------
10 ARCHITECTURE FSM OF controller IS
11   TYPE state IS (S0, S5, S10);
12   SIGNAL pr_state, nx_state : state;
13 BEGIN
14   -- registar stanja ---------------------------------------------
15   PROCESS (clk,rst)
16   BEGIN
17      IF(rst='1')THEN
18         pr_state <=S0;
19      ELSIF(clk'EVENT AND clk='1') THEN
20         pr_state <= nx_state;
21      END IF;
22   END PROCESS;
23   -- logika sledeceg stanja/izlaza -------------------------------
24   PROCESS(D5,D10,pr_state)
25   BEGIN
```

```
26      nx_state <= pr_state;
27      K<='0';V<='0';
28      CASE pr_state IS
29         WHEN S0 =>
30            IF(D5='1')THEN
31               nx_state <= S5;
32            ELSIF(D10='1') THEN
33               nx_state <= S10;
34            END IF;
35         WHEN S5 =>
36            IF(D5='1')THEN
37               nx_state <= S10;
38            ELSIF(D10='1') THEN
39               K<='1';
40               nx_state <= S0;
41            END IF;
42         WHEN S10 =>
43            IF(D5='1')THEN
44               K<='1';
45               nx_state <= S0;
46            ELSIF(D10='1') THEN
47               K<='1';V<='1';
48               nx_state <= S0;
49            END IF;
50      END CASE;
51   END PROCESS;
52 END FSM;
53 ------------------------------------------------------------------
```

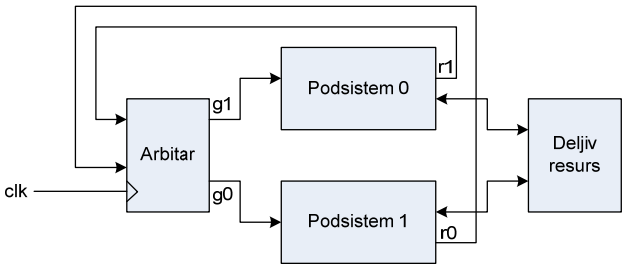


Sl. 6-19 Rezulati simulacije VHDL opisa automata za prodaju karata.

Pr. 6-9 Arbitar

U složenim digitalnim sistemima čest je slučaj da više podsistema dele isti resurs. Na primer, nekoliko procesora mogu deliti isti blok memorije, ili nekoliko periferijskih jedinica mogu deliti istu magistralu. Arbitar je kolo koje sprečava konflikte i koordinira pristup deljivom resursu. U ovom primeru, razmotrićemo realizaciju arbitar za sistem sa dva podsistema i jednim deljivim resursom. Podsistemi komuniciraju s arbitrom posredstvom dva signala, *request* (zahtev) i *grant* (dozvola), koji su označeni sa *r*1 i *g*1 za podsistem 1, odnosno sa *r*0 i *g*0 za podsistem 0, kao što je prikazano na Sl. 6-20. Podsistem kome je potreban resurs aktivira signal *request*. Arbitar vodi računa o dostupnosti resursa, nadgleda zahteve i aktiviranjem odgovarajućeg signala *grant* daje dozvolu onom

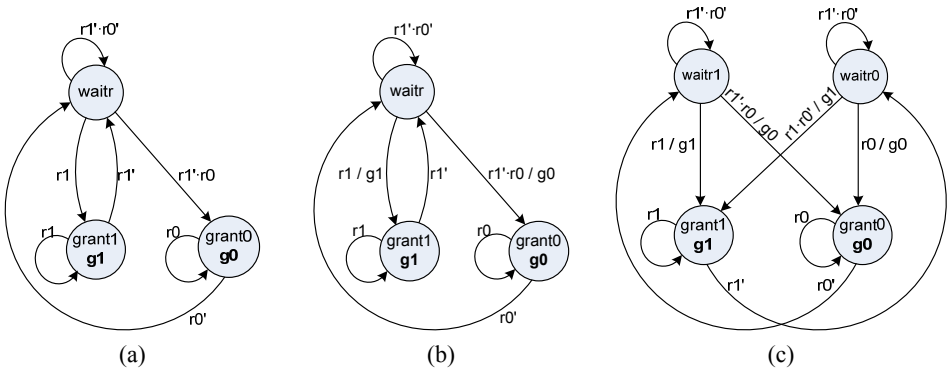
podsistemu koji je uputio zahtev. Podsistem kome je upućen signal *grant* dobija pravo da koristi resurs. Nakon što je obavio svoj zadatak i resurs mu više nije potreban, podsistem deaktivira signal *request*. Ako neki podsistem uputi zahtev arbitru, a resurs je trenutno zauzet, signal dozvole ostaće neaktivan sve dok se resurs ne oslobodi.



Sl. 6-20 Blok dijagram arbitra.

Razmotrićemo tri realizacije arbitra u vidu konačnog automata. Prva realizacija je prikazana na Sl. 6-21(a). Osnovna karakteristika ovog rešenja je ta da se viši prioritet daje podsistemu 1. Dijagram stanja sadrži tri stanja: *waitr*, *grant1* i *grant0*. U stanju *waitr* resurs je dostupan za korišćenje (nezauzet), a arbitar čeka na zahtev. Stanja *grant1* i *grant0* ukazuju da je resurs zauzet i trenutno dodeljen podsistemu 1, odnosno podsistemu 0, respektivno. Na početku rada, arbitar je u stanju *waitr*. Ako se aktivira signal *r*1, arbitar prelazi u stanje *grant1*. U ovom stanju aktivan je signal *g*1 koji informiše podsistem 1 da je resurs slobodan. Nakon što završi korišćenje resursa, podsistem 1 deaktivira signal *r*1, a arbitar se vraća u stanje *waitr* da čeka na novi zahtev. Ako se u stanju *waitr*, umesto *r*1 aktivira signal *r*0, arbitar će, prelaskom u stanje *grant0*, dati dozvolu podsistemu 0. Uočimo da s obzirom na to kako su formirani uslovi na izlaznim granama iz stanja *waitr*, u slučaju istovremenog aktiviranja oba zahteva, *r*1 i *r*0, prednost dobija podsistem 1.

Rešenje sa Sl. 6-21(b) predstavlja Milijevu varijantu arbitra, jer se sada izlazni signali aktiviraju ne samo u stanjima *grant0* i *grant1* već i na prelazima iz stanja *waitr* u ova dva stanja. Učinjena modifikacija omogućava da podsistemi dobiju dozvolu korišćenja resursa jedan taktni ciklus ranije (signal dozvole se javlja u istom taktnom ciklusu u kome je podsistem uputio zahtev, a ne u sledećem, kao u Murovoj varijanti arbitra). Međutim, i u ovom, kao i u prvom rešenju, u situacijama kad oba podsistema istovremeno upute zahtev, viši prioritet se daje podsistemu 1.



Sl. 6-21 Tri realizacije arbitar: (a) Murovi izlazi – fiksni prioriteti; (b) Milijevi izlazi – fiksni prioriteti; (c) Milijevi izlazi - ″fer″ politika.

Fiksni prioriteti mogu stvoriti probleme u slučaju kada podsistem 1 neprekidno zahteva resurs. Pod takvim okolnostima, podsistem 2 nikad neće dobiti priliku da pristupi resursu. Treće i konačno rešenje uvodi ″fer″ politiku arbitraže, tako što vodi računa o tome koji podsistem je poslednji koristio resurs i u slučajevima istovremenog upućivanja zahteva prednost daje onom drugom podsistemu. Dijagram stanja ovog rešenje prikazan je na Sl. 6-21(c). Umesto jednog sada uočavamo dva stanja čekanja, *waitr1* i *waitr0*. U stanju *waitr1* viši prioritet ima podsistem 1, a u stanju *waitr0* podsistem 0. Ako je podsistem 0 poslednji koristio resurs, arbitar čeka na novi zahtev u stanju *waitr1*. Ako je podsistem 1 poslednji koristio resurs, arbitar je u stanju *waitr0*. U nastavku je dat VHDL opis trećeg rešenja.

```
1  ----------------------------------------------------------------------
2  LIBRARY IEEE;
3  USE IEEE.STD_LOGIC_1164.ALL;
4  ----------------------------------------------------------------------
5  ENTITY arbiter IS
6     PORT (clk, rst : IN STD_LOGIC;
7           r : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
8           g : OUT STD_LOGIC_VECTOR(1 DOWNTO 0));
9  END arbiter;
10 ----------------------------------------------------------------------
11 ARCHITECTURE two_seg_arch of arbiter IS
12   TYPE state IS (WAITR0, WAITR1, GRANT0, GRANT1);
13   SIGNAL pr_state, nx_state : state;
14 BEGIN
15    -- registar stanja ------------------------------------------
16    PROCESS(clk, rst)
17    BEGIN
18      IF(rst = '1') THEN
19        pr_state <= WAITR0;
20      ELSIF( clk'EVENT AND clk = '1') THEN
21        pr_state <= nx_state;
22      END IF;
23    END PROCESS;
24    -- logika sledeceg stanja/izlaza ----------------------------
25    PROCESS(pr_state, r)
26    BEGIN
27      g <= "00";   -- podrazumevane vrednosti izlaza
28      nx_state <= pr_state;
29      CASE(pr_state) IS
30        WHEN WAITR0 =>
31          IF r(0) = '1' THEN
32            g(0) <= '1';
33            nx_state <= GRANT0;
34          ELSIF r(1) = '1' THEN
35            g(1) <= '1';
36            nx_state <= GRANT1;
37          END IF;
38        WHEN WAITR1 =>
39          IF r(1) = '1' THEN
40            g(1) <= '1';
41            nx_state <= GRANT1;
42          ELSIF r(0) = '1' THEN
43            g(0) <= '1';
44            nx_state <= GRANT0;
```

```
45          END IF;
46        WHEN GRANT0 =>
47          IF r(0) = '0' THEN
48            nx_state <= WAITR1;
49          END IF;
50        WHEN GRANT1 =>
51          IF r(1) = '0' THEN
52            nx_state <= WAITR0;
53          END IF;
54      END CASE;
55    END PROCESS;
56 END two_seg_arch;
```

7. HIJERARHIJSKO PROJEKTOVANJE

Složeni digitalni sistemi se projektuju na hijerarhijski način. To znači da se sistem koji treba projektovati postepeno i rekurzivno deli na module manje složenosti. Moduli dobijeni podelom se potom nezavisno realizuju i konačno, međusobno povezuju. Pojam rekurzivno znači da se proces podele može sprovesti i nad modulima koji su nastali prethodnom podelom, obično do nivoa modula koji su dovoljno jednostavni da se lako mogu direktno realizovati, ili modula koji su raspoloživi u vidu gotovih, ranije projektovanih gradivnih jedinica.

Beneficije koje proističu iz primene hijerarhijskog projektovanja su brojne, a neke od bitnijih su sledeće:

- Pažnja projektanta se skreće sa celokupnog sistem na pojedinačne module koje može nezavisno da projektuje, analizira i verifikuje.
- Nakon izvršene podele, zadatak projektovanja se može raspodeliti između više projektanata, koji će nezavisno i istovremeno raditi na različitim delovima istog sistema. Na taj način se skraćuje ukupno vreme projektovanja.
- Složeni sistemi ne predstavljaju problem samo projektantima, već i softverima za sintezu. Automatska sinteza obimnog i komplikovanog VHDL kôda zahteva utrošak enormnog računarskog vremena, uz povećane zahteve za radnom memorijom. Uz to, konačni rezultat sinteze često nije zadovoljavajući, kako u pogledu performansi, tako i u pogledu cene sintetizovanog hardvera (broja upotrebljenih gejtova). Razbijanjem sistema na manje module i njihovom zasebnom sintezom, proces sinteze celokupnog sistema se može učiniti bržim i efikasnijim.
- Mnogi sistemi poseduju zajedničke ili slične funkcionalnosti. Nakon što je modul projektovan i verifikovan za potrebe jednog sistema, isti taj modul se može iskoristiti i u budućim projektima.

Obezbeđivanje podrške za modularno i hijerarhijsko projektovanje jedan je od osnovnih ciljeva VHDL-a. U VHDL-u postoje brojne jezičke konstrukcije za tu namenu, od kojih su, sa stanovišta sinteze, najznačajnije:

- COMPONENT
- GENERIC
- CONFIGURATION

- LIBRARY
- PACKAGE
- FUNCTION
- PROCEDURE

Konstrukcije *componet*, *generic* i *configuration* su osnovni mehanizmi za kreiranje hijerarhijski organizovanog VHDL kôda. Konstrukcije *package* i *library* pomažu u organizaciji obimnog kôda, dok *funcion* i *procedure* služe za kreiranje potprograma. Konkretno, često korišćeni delovi kôda se pišu u obliku komponenti (*component*), funkcija i procedura (*function*, *procedure*) i smeštaju u pakete (*package*). Jedan ili više paketa se mogu smestiti u biblioteku (*library*), koja se potom može uključivati u druge, buduće projekte ili razmenjivati s drugim projektantima.

7.1. PACKAGE

Kako VHDL kôd postaje obimniji i složeniji, tako raste broj i raznovrsnost deklaracija (npr. deklaracije konstanti, tipova podataka, komponenti, funkcija itd.) koje moraju biti sadržane u deklarativnim sekcijama različitih projektnih jedinica (entiteta, arhitektura i procesa). Kad se sistem podeli na više manjih podsistema, neke deklaracije moraju biti kopirane u različite projektne jedinice, a eventualna naknadna promena bilo koje deklaracije zahtevala bi neposrednu intervenciju u kôdu svake jedinici. Svrha paketa je upravo grupisanje zajedničkih i često korišćenih deklaracija na jednom mestu u projektu. Osim deklaracija komponenti, funkcija i procedura, paket može da sadrži i deklaracije drugih jezičkih celina, kao što su tipovi podataka, konstante i globalno vidljivi signali. U bilo kom delu projekta gde je neophodna neka deklaracija iz paketa, dovoljno je naredbom *use* uključiti odgovarajući paket, a svaka naknadna izmena bilo koje deklaracije iz paketa biće po automatizmu vidljiva u svim delovima projekta.

Paket ima sledeću sintaksu:

```
PACKAGE ime_paketa IS
   (deklaracije)
END ime_paketa
[PACKAGE BODY ime_paketa IS
   (funkcije i procedure)
END ime_paketa;]
```

Paket čine dve celine: deklaracija paketa (*package*) i telo paketa (*package body*). Prva celina je obavezna i sadrži sve deklaracije. Telo paketa je opciono i neophodno samo ako u deklaraciji paketa postoji jedna ili više deklaracija potprograma (funkcija ili procedura) i pri tom sadrži kompletne opise (zaglavlja i tela) svih potprograma. Sekcije *package* i *package body* moraju imati isto ime.

Pr. 7-1 Primer jednostavnog paketa

VHDL kôd iz ovog primera definiše paketa imena *nas_paket*. S obzirom na to što ovaj paket sadrži samo deklaracije *type* i *constant* (dva tipa podataka i jednu konstantu), telo paketa nije potrebno.

```
1 -- Paket bez funkcija i procedura --------------------------------
2 LIBRARY IEEE;
3 USE IEEE.STD_LOGIC_1164.ALL;
```

```
4  -- Deklaracija paketa ---------------------------------------------
5  PACKAGE nas_paket IS
6    TYPE state IS (st1,st2,st3,st4);
7    TYPE boja  IS (crvena, bela, plava);
8    CONSTANT niz: STD_LOGIC_VECTOR(7 DOWNTO 0) := "11111111";
9  END nas_paket;
```

Pr. 7-2 Paket s funkcijom

Paket iz ovog primera, pored deklaracije dva tipa podataka i jedne konstante, sadrži i jednu funkciju. Iz tog razloga, telo paketa je neophodno. Deklaracija funkcije je smeštena u deklaraciji paketa, a kompletan kôd funkcije u telu paketa.

```
1  -- Primer paketa s funkcijom -------------------------------------
2  LIBRARY IEEE;
3  USE IEEE.STD_LOGIC_1164.ALL;
4  -- Deklaracija paketa ---------------------------------------------
5  PACKAGE nas_paket IS
6    TYPE state IS (st1,st2,st3,st4);
7    TYPE boja  IS (crvena, bela, plava);
8    CONSTANT niz: STD_LOGIC_VECTOR(7 DOWNTO 0) := "11111111";
9    FUNCTION rastuca_ivica(SIGNAL s : STD_LOGIC) RETURN BOOLEAN;
10 END nas_paket;
11 -- Telo paketa ----------------------------------------------------
12 PACKAGE BODY nas_paket IS
13   FUNCTION rastuca_ivica(SIGNAL s : STD_LOGIC) RETURN BOOLEAN IS
14   BEGIN
15     RETURN (s'EVENT AND s='1');
16   END rastuca_ivica;
17 END nas_paket;
```

Deklaracija i telo paketa su projektne jedinice VHDL-a, koje se prilikom kompajliranja nezavisno analiziraju i smeštaju u biblioteku. Da bi neka konkretna deklaracija iz paketa postala vidljiva u nekom delu VHDL projekta, neophodno je koristiti naredbu *use*, čija je sintaksa sledećeg oblika:

```
USE ime_biblioteke.ime_paket.ime_deklaracije;
```

Član *ime_deklaracije* se najčešće zamenjuje ključnom rečju *all*, čime se postiže vidljivost **svih** deklaracija sadržanih u navedenom paketu. Na primer, ispod je pokazano kako se u VHDL kôd uključuje paket iz Pr. 7-2, pod pretpostavkom da je paket *nas_paket* deo istog projekta i da se uzima iz podrazumevane, radne biblioteke *work*:

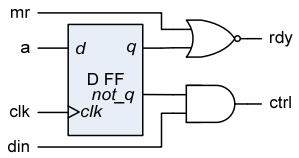
```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE work.nas_paket.all;

ENTITY . . .
. . .
ARCHITECTURE . . .
. . .
```

7.2. COMPONENT

Metodologija hijerarhijskog projektovanja se, u osnovi, zasniva na podeli sistema na manje module. U gotovo svim ranijim primerima VHDL opisa korišćen je upravo ovakav pristup, iako to nije eksplicitno naglašeno. Pisanju VHDL kôda, čak i za jednostavna kola, po pravilu je prethodilo crtanje skice konceptualnog dijagrama s prikazom nekoliko glavnih delova kola i njihovih veze. Nakon toga, svaki blok iz konceptualnog dijagrama opisivan je jednim segmentom VHDL kôda (procesom, konkurentnim naredbama i dr.). Komponente iz VHDL-a upravo predstavljaju formalan način za modularno, a onda i hijerarhijsko organizovanje složenijih VHDL opisa.

Komponenta, kao pojam, odnosi se celoviti VHDL kôd (entitet plus arhitektura) koji opisuje neko kolo. Ako se jedna takva celina deklariše kao *component*, ona može postati dostupna za korišćenje u drugim delovima VHDL kôda za konstrukciju složenijih kola ili u novim projektima, bez potrebe ponovnog pisanje istog kôda.

Pr. 7-3 Jednostavan strukturni opis



Sl. 7-1 Kolo iz Pr. 7-3.

Razmotrimo kolo sa Sl. 7-1 i njegov strukturni VHDL opis:

```
-- Primer strukturnog opisa -------------------------------------
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
----------------------------------------------------------------------
ENTITY gating IS
  PORT (a,clk,mr,din : STD_LOGC;
        rdy,ctrl : OUT STD_LOGC);
END gating;
----------------------------------------------------------------------
ARCHITECTURE struktura OF gating IS
  COMPONENT AND2
    PORT(x,y: IN STD_LOGC; z OUT STD_LOGC);
  END COMPONENT;
  COMPONENT DFF
    PORT(d,clock: IN STD_LOGC; q, not_q OUT STD_LOGC);
  END COMPONENT;
  COMPONENT NOR2
    PORT(x,y: IN STD_LOGC; z OUT STD_LOGC);
  END COMPONENT;
  SIGNAL s1, s2 : STD_LOGC;
BEGIN
  D1: DFF PORT MAP(a,clk,s1,s2);
  A1: AND2 PORT MAP(s2,din,ctrl);
  N1: NOR2 PROT MAP(s1,mr,rdy);
END stuktura;
----------------------------------------------------------------------
```

U deklarativnom delu arhitekture deklarisane su tri komponente: *and2* (dvoulazno I kolo, linije 11-13), *dff* (D flip-flop, linije 14-16) i *nor2* (dvoulazno NILI kolo, linije 17-19). (Pretpostavka je da se VHDL opisi ovih komponenti nalaze u nekoj drugoj datoteci istog projekta, da su kompajlirani i na taj način smešteni u biblioteku *work*.) Komponente *dff*, *and2* i *nor2* potom su *instancirane* u telu arhitekture (linije 22-24) i povezane s odgovarajućim portovima entiteta *gating* kao i međusobno pomoću dva signala, *s1* i *s2*. Naredbe za instanciranje komponente spadaju u grupu konkurentnih naredbi i otuda redosled njihovog navođenja u telu arhitekture nije od značaja. Jedna ista komponenta se može instancirati proizvoljan broj puta, ali svaka instanca mora imati jedinstvenu labelu.

Konstrukcija *component* predstavlja osnovu strukturnog i hijerarhijskog projektovanja u VHDL-u. Strukturno projektovanje znači da se složenija kola konstruišu povezivanjem jednostavnijih komponenti. Hijerarhijsko projektovanje se ostvaruje tako što se strukturno opisana kola deklarišu takođe kao komponente i zatim koriste za realizaciju još složenijih kola. Na primer, često korišćena kola, poput flip-flopova i logičkih kola, mogu se jednom opisati VHDL kôdom i deklarisati kao komponente, a zatim proizvoljan broj puta koristiti (*instancirati*) za realizaciju složenijih struktura, kao što su registri, sabirači itd.

Korišćenje komponenti u VHDL kôdu zahteva dva koraka. Prvi korak se odnosi na deklarisanje, a drugi na instanciranje komponente.

7.2.1. Deklaracija komponente

Deklaracija komponente sadrži informacije o interfejsu komponente, što podrazumeva ulazne i izlazne portove i druge relevantne parametre komponente. Informacije o komponenti sadržane u deklaraciji komponente su identične onim iz deklaracije entiteta. Sintaksa deklaracije komponente je sledećeg oblika:

```
COMPONENT ime_komponente IS
  GENERIC(ime_parametra : tip_parametra;
          ime_parametra : tip_parametra;
          . . . );
  PORT (ime_porta : mode_signala tip_signala;
        ime_porta : mode_signala tip_signala;
        . . . );
END COMPONENT;
```

Deo *generic* iz deklaracije komponente je opcion i sadrži spisak parametara koji treba proslediti komponenti prilikom njenog instanciranja. Deo za portove, slično kao kod entiteta, sadrži spisak imena portova zajedno s njihovim smerom (*in*, *out*, *bufer* ili *inout*) i tipom (*std_logic*, *bit*, *integer* i sl.).

Pr. 7-4 Sličnost između *entity* i *component*

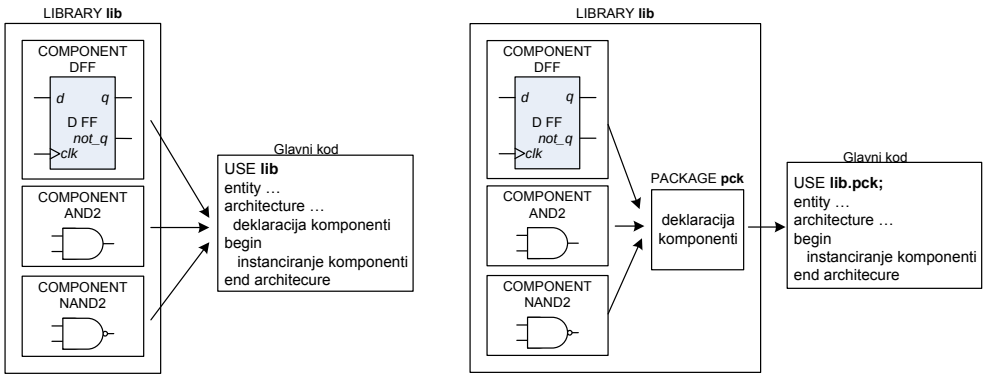
Pretpostavimo da raspolažemo kompletnim VHDL opisom D flip-flopa (entitet *dff* je prikazan na Sl. 7-2(a)) i da ovaj opis, u vidu komponente, želimo da koristimo u drugim delovima projekta. U tom cilju, potrebno je kreirati odgovarajuću deklaraciju komponente, koja će potom biti kopirana u deklarativne sekcije onih arhitektura u kojima će *dff* biti instanciran. Deklaracija komponente *dff* prikazana je na Sl. 7-2(b). Kao što vidimo, deklaracija komponente je gotovo identična deklaraciji entiteta. Službena reč *entity* je zamenjena rečju *component*, reč *is* je izostavljena, a ponovljeno ime entiteta iz poslednje linije zamenjeno je službenom rečju *component*.

```
ENTITY DFF IS                          COMPONENT DFF
 PORT(d : IN STD_LOGC;                  PORT(d : IN STD_LOGC;
      clock: IN STD_LOGC;                    clock: IN STD_LOGC;
      q : IN STD_LOGC            →           q : IN STD_LOGC
      not_q: OUT STD_LOGC);                  not_q : OUT STD_LOGC);
END DFF;                               END COMPONENT;
```

(a) (b)

Sl. 7-2 *Entity* v.s. *Component*: (a) etitet i (b) odgovarajuća deklaracija komponente.

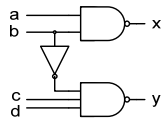
Na Sl. 7-3 je ilustrovan princip rada s komponentama. Pre korišćenja, komponenta mora biti projektovana, kompajlirana i smeštena u biblioteku. Postoje dva načina za uključivanje komponenti u VHDL kôd (kazaćemo, u glavni kôd). Pristup koji je korišćen u Pr. 7-3 podrazumeva da se komponente deklarišu u deklarativnom delu arhitekture (ovaj princip je ilustrovan na Sl. 7-3(a)). Drugi način je taj da se deklaracije svih komponenti prvo smeste u paket, koji se potom uključuje u glavni kôd (Sl. 7-3(b)). Na taj način, eliminiše se potreba da se komponenta eksplicitno deklariše u svakoj arhitekturi gde se koristi.



(a) (b)

Sl. 7-3 Načini za deklaraciju komponenti: (a) deklaracija u arhitekturi; (b) deklaracija u paketu.

Pr. 7-5 Deklaracija komponenti u arhitekturi



Sl. 7-4 Kolo iz Pr. 7-5.

U ovom primeru je predstavljen *strukturni* VHDL opis kola sa Sl. 7-4, odnosno opis u kome se koriste isključivo komponente (*inv*, *nand2* i *nand3*) i to na način ilustrovan na Sl. 7-3(a), gde se paket ne koristi. Iako to nije obavezan zahtev, preporučljivo je da se VHDL kôd nekog složenijeg, hijerarhijski organizovanog projekta raspodeli na više datoteka, tako da svaka datoteka sadrži opis samo jednog modula (entitet i arhitektura) ili paketa. U konkretnom primeru, potrebne su četiri datoteke, po jedna datoteka za svaku komponentu plus datoteka za glavni kôd. Sadržaji sve četiri datoteke dati su ispod. Pošto se paket ne koristi, komponente moraju biti deklarisane u deklarativnoj sekciji arhitekture glavnog kôda (datoteka *projekat.vhd*).

```
------ Datoteka inv.vhd ----------------------------------------
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY inv IS
  PORT ( a: IN STD_LOGIC; b : OUT STD_LOGIC);
END inv;
ARCHITECTURE inv OF inv IS
BEGIN
   b <= NOT a;
END inv;
```

```
------ Datoteka nand2.vhd ---------------------------------------
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY nand2 IS
  PORT ( a,b: IN STD_LOGIC; c : OUT STD_LOGIC);
END nand2;
ARCHITECTURE nand2 OF nand2 IS
BEGIN
   c <= a NAND b;
END nand2;
```

```
------ Datoteka nand3.vhd ---------------------------------------
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY nand3 IS
  PORT ( a,b,c: IN STD_LOGIC; d : OUT STD_LOGIC);
END nand3;
ARCHITECTURE nand3 OF nand3 IS
BEGIN
   d <= NOT(a AND b AND c);
END nand3;
```

```
------ Datoteka projekat.vhd ------------------------------------
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY projekat IS
  PORT ( a,b,c,d: IN STD_LOGIC;
         x,y : OUT STD_LOGIC);
END projekat;
ARCHITECTURE struktura OF projekat IS
  COMPONENT inv IS
     PORT(a: IN STD_LOGIC; b: OUT STD_LOGIC);
  END COMPONENT;
  COMPONENT nand2 IS
     PORT(a,b: IN STD_LOGIC; c: OUT STD_LOGIC);
  END COMPONENT;
  COMPONENT nand3 IS
     PORT(a,b,c: IN STD_LOGIC; d: OUT STD_LOGIC);
  END COMPONENT;
  SIGNAL w: STD_LOGIC;
BEGIN
   K1: inv PORT MAP(b,w);
   K2: nand2 PORT MAP(a,b,x);
   K3: nand3 PORT MAP(w,c,d,y);
END struktura;
```

Pr. 7-6 Deklaracije komponenti u paketu.

U ovom primeru, ponovo je realizovan projekat iz prethodnog primera (Sl. 7-4), ali sada na način kao na Sl. 7-3(b), koji podrazumeva kreiranje paketa s deklaracijama korišćenih komponenti (*inv*, *nand2* i *nand3*). Sada je potrebno pet datoteka: tri za VHDL kôd komponenata i po jedna za paket i glavni kôd, odnosno vršni modul sistema. Iako u odnosu na pristup iz Pr. 7-5 sada postoji još jedna dodatna datoteka (za paket), paket se kreira samo jedanput, a koristi proizvoljan broj puta u različitim projektnim datotekama. Na taj način, izbegava se ponavljanje deklaracija komponenti u svakoj arhitekturi gde se koriste. Sledi VHDL kôd kompletnog projekta. Datoteka *nase_komponente.vhd* sadrži istoimeni paket s navedenim deklaracijama komponenti. Paket se uključuje u glavni kôd (datoteka *projekat.vhd*) u liniji 4.

```
1  ------ Datoteka inv.vhd ----------------------------------------
2  LIBRARY IEEE;
3  USE IEEE.STD_LOGIC_1164.ALL;
4  ENTITY inv IS
5     PORT ( a: IN STD_LOGIC; b : OUT STD_LOGIC);
6  END inv;
7  ARCHITECTURE inv OF inv IS
8  BEGIN
9     b <= NOT a;
10 END inv;
```

```
1 ----- Datoteka nand2.vhd ----------------------------------------
2  LIBRARY IEEE;
3  USE IEEE.STD_LOGIC_1164.ALL;
4  ENTITY nand2 IS
5    PORT ( a,b: IN STD_LOGIC; c : OUT STD_LOGIC);
6  END nand2;
7  ARCHITECTURE nand2 OF nand2 IS
8  BEGIN
9     c <= a NAND b;
10 END nand2;
```

```
1  ------ Datoteka nand3.vhd ----------------------------------------
2  LIBRARY IEEE;
3  USE IEEE.STD_LOGIC_1164.ALL;
4  ENTITY nand3 IS
5    PORT ( a,b,c: IN STD_LOGIC; d : OUT STD_LOGIC);
6  END nand3;
7  ARCHITECTURE nand3 OF nand3 IS
8  BEGIN
9    d <= NOT(a AND b AND c);
10 END nand3;
```

```
1 ------ Datoteka nase_komponente.vhd -----------------------------
2  LIBRARY IEEE;
3  USE IEEE.STD_LOGIC_1164.ALL;
4  PACKAGE nase_komponente IS
5    COMPONENT inv IS
6       PORT(a: IN STD_LOGIC; b: OUT STD_LOGIC);
7    END COMPONENT;
8    COMPONENT nand2 IS
9       PORT(a,b: IN STD_LOGIC; c: OUT STD_LOGIC);
10   END COMPONENT;
```

```
11   COMPONENT nand3 IS
12      PORT(a,b,c: IN STD_LOGIC; d: OUT STD_LOGIC);
13   END COMPONENT;
14 END nase_komponente;
```

```
1 ------ Datoteka projekat.vhd ------------------------------------
2  LIBRARY IEEE;
3  USE IEEE.STD_LOGIC_1164.ALL;
4  USE work.nase_komponente.ALL;
5  ENTITY projekat IS
6     PORT ( a,b,c,d: IN STD_LOGIC;
7            x,y : OUT STD_LOGIC);
8 END projekat;
9  ARCHITECTURE struktura OF projekat IS
10    SIGNAL w: STD_LOGIC;
11 BEGIN
12    K1: inv PORT MAP(b,w);
13    K2: nand2 PORT MAP(a,b,x);
14    K3: nand3 PORT MAP(w,c,d,y);
15 END struktura;
```

7.2.2. Instanciranje komponente

Nakon što je deklarisana, komponenta se može instancirati u telu arhitekture pomoću naredbe sledeće sintakse:

```
labela_instance: ime_komponente
   PORT MAP(lista_portova)
   GENERIC MAP(lista_parametara);
```

Naredba za instanciranje počinje labelom, posle koje sledi ime komponente koja se instancira, a onda i dve deklaracije: *port map* i *generic map*. Labela identifikuje konkretnu instancu komponente i mora biti jedinstvena u okviru arhitekture. Lista portova iz deklaracije *port map* ostvaruje veze (ili ″asocijacije″) između portova komponente (tzv. *formalnih* signala) i spoljnih signala (tzv. *stvarnih* signala). Konstrukcija *generic map* dodeljuje stvarne vrednosti generičkim parametrima komponente. Ukoliko komponenta nema generičke parametre, sekcija *generic map* se ne navodi u naredbi za instanciranje.

Razmotrimo još jedanput Pr. 7-3. Deklaracija komponente *and2* (linije 11-13) tumači se ovako: komponenta ima dva ulazna, *x* i *y* i jedan izlazni port, *z*, svi tipa *std_logic*. U telu arhitekture, ova komponenta je jedanput instancirana (linija 23). Labela *A1* definiše ime ove instance u arhitekturi *struktura*. Lista portova, *port map*(*s2, din, ctrl*), povezuje interni signal *s2* s portom *x* komponente *and2*, kao i portove *din* i *ctrl* entiteta *gating* s portovima *y* i *z* komponente *and2*. Uočimo da se redosled signala u listi portova podudara sa redosledom portova iz deklaracije komponente (linija 12). Napomenimo da se ovakav način uspostavljanja korespondencije između formalnih i stvarnih signala naziva *poziciono povezivanje*.

Port map. Prilikom instanciranja komponente, postoje dva načina za povezivanje njenih portova sa spoljnim signalima: *poziciono* i *nominalno*. Razmotrimo sledeći primer:

```
COMPONENT inv IS
  PORT(a: IN STD_LOGIC; b: OUT STD_LOGIC);
END COMPONENT;
. . .
K1: inv PORT MAP(x,y);
```

Primenjeno je *poziciono* povezivanje: signali *x* i *y* se vezuju na portove *a* i *b* komponente *inv*, respektivno. Povezivanje je implicitno, određeno redosledom po kojem su portovi navedeni u deklaraciji komponente.

Identično povezivanje može se postići i tzv. *nominalnim* povezivanjem, na sledeći način:

```
K1: inv PORT MAP(a=>x,b=>y);
```

Zapis (*a=>x, b=>y*) znači da se port *a* komponente *inv* povezuje sa signalom *x*, a port *b* sa signalom *y*. Povezivanje je sada eksplicitno, jer se za svaki port komponente navodi ime signala s kojim se taj port povezuje. Kod nominalnog povezivanja, redosled povezivanja nije bitan. Sledeća naredba za instanciranje komponenta ima isti efekat kao prethodna:

```
K1: inv PORT MAP(b=>y,a=>x);
```

Poziciono povezivanje je kompaktnije, ali je zato nominalno preglednije. Problem s pozicionim povezivanjem nastaje ako se naknadno, iz bilo kog razloga, promeni redosled portova u entitetu komponente. Takva jedna modifikacija, koja je inače minorna sa stanovišta komponente, zahtevala bi intervenciju u svim arhitekturama u kojima je komponenta instancirana i to radi preuređenja liste portova, što je svakako podložno lakom pravljenju grešaka.

Dopušteno je da neki portovi komponente ostanu nepovezani prilikom instanciranja. Za označavanje nepovezanih portova koristi se službena reč *open*, npr.:

```
K2: nase_kolo PORT MAP(x, y, open, z);
```

Ili, za nominalno povezivanje:

```
K2: nase_kolo PORT MAP(a=>x, b=>y, c=>open, d=>z);
```

Nepovezani portovi su po pravilu izlazni portovi komponente koji se ne koriste (nisu potrebni) u arhitekturi koja se projektuje. Softveri za sintezu poseduju mogućnost da prilikom optimizacije hardvera odstrane iz komponente automatski one delove logike koji su isključivo namenjeni za generisanje signala nepovezanih izlaznih portova. Ako se u naredbi za instanciranje koristi nominalno povezivanje, ″otvoreni″ portovi mogu biti izostavljeni iz liste portova, a efekat će biti isti kao da je korišćena opcija *open*:

```
K2: nase_kolo PORT MAP(x=>a, y=>b, z=>d);
```

Međutim, radi bolje čitljivosti kôda, dobra je praksa da se u listi portova navedu svi portovi, a oni koji treba da ostanu nepovezani eksplicitno označe sa *open*.

VHDL dozvoljava da se prilikom instanciranja komponente i ulazni portovi označe sa *open*. U tom slučaju, na ulazni port se fiksno postavlja podrazumevanu vrednost (´0´ ili ´1´). Međutim, dobra je praksa ne ostavljati ulazni port nepovezanim, već eksplicitno, u naredbi za instanciranje, definisati vrednost koja će fiksno biti postavljena na tom portu, kao u sledećem primeru:

```
brojac: BCD_brojac PORT MAP(din=>d, q=>q, en=>´1´);
```

U prethodnoj naredbi, instancirana je komponenta *BCD_brojač* i pri tom je njen ulaz za dozvolu rada, *en*, fiksno postavljen na vrednost ´1´. Na taj način je ukinuta mogućnost kontrole rada instanciranog brojača, a brojanje je uvek dozvoljeno. Međutim, treba naglasiti da pojedini softveri za sintezu ne dozvoljavaju korišćenje konstantnih vrednosti u naredbi za instanciranje komponente. U takvim slučajevima, rešenje se sastoji u tome da se u

arhitekturi deklariše interni signal koji će najpre biti postavljen na željenu konstantnu vrednost, a zatim, u naredbi za instanciranje, povezan sa odgovarajućim ulaznim portom:

```
ARCHITECTURE ...
  SIGNAL ena : STD_LOGIC;
BEGIN
  ena <= '1';
  ...
  brojac: BCD_brojac PORT MAP(din=>d, q=>q, en=>ena);
  ...
END ...
```

Generic map. Generički parametri se koriste za kreiranje tzv. generičkih, odnosno parametrizovanih VHDL opisa, tj. opisa koji modeliraju ne samo jedno, konkretno kolo, već čitavu klasu srodnih kola koja, tipično, poseduju istu funkcionalnosti, a razlikuju se po dimenzijama portova. Na primer, na Sl. 7-5(a) je prikazan entitet generičkog binarnog brojača. Kao što vidimo, opseg izlaznog porta *q* nije definisan konkretnom brojnom vrednošću, već je izražen preko *generičkog* parametra *N*. Odgovarajuća deklaracija generičke komponente prikazana je na Sl. 7-5(b).

```
ENTITY binarni_brojac IS
  GENERIC(N : NATURAL);
  PORT(clk, rst : IN STD_LOGIC;
       q : OUT STD_LOGIC_VECTOR(N-1 DOWNTO 0);
END bin_brojac;
```

(a)

```
COMPONENT binarni_brojac
  GENERIC(N : NATURAL);
  PORT(clk, rst : IN STD_LOGIC;
       q : OUT STD_LOGIC_VECTOR(N-1 DOWNTO 0);
END COMPONENT;
```

(b)

Sl. 7-5 Generički parametri: (a) deklaracija generičkog entiteta; (b) deklaracija generičke komponente.

Naredba za instanciranje generičke komponente pored *port map* sadrži i konstrukciju *generic map*, kojom se postavljaju stvarne vrednosti generičkih parametra. Na primer, sledeća naredba kreira jednu instancu dvobitnog binarnog brojača:

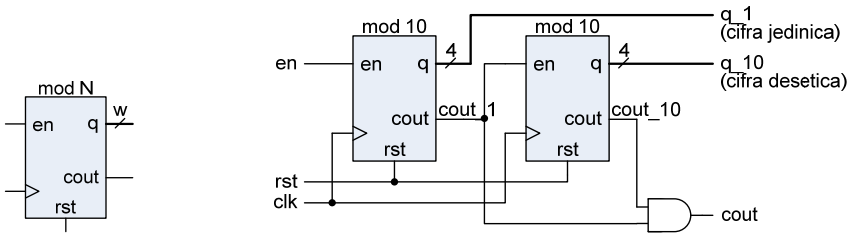
```
dvo_bit_brojac: binarni_brojac
  GENERIC MAP(N => 2)
  PORT MAP(clk => clock, rst => reset, q => q_out);
```

Generičke komponente mogu imati proizvoljan broj generičkih parametara, a u konstrukciji *generi map*, kao i u *port map*, može se koristiti bilo poziciono bilo nominalno povezivanje.

Pr. 7-7 Dvocifarski dekadni brojač

U ovom primeru, projektovaćemo dvocifarski dekadni brojač na osnovu generičkog mod_n brojača (tj . brojača po modulu *n*). U Pr. 5-22 je predstavljen **programabilni** brojač po modulu *n*. Ono što brojač iz Pr. 5-22 čini programabilnim jeste mogućnost izbora osnove brojanja posredstvom namenskog porta, *m*. Prvi od dva VHDL opisa iz ovog primera modelira **generički** mod_n brojač, čija se osnova brojanja i dužina (broj bita) definišu generičkim parametrima, *n* i *w* (Sl. 7-6(a)). Ovaj brojač odbrojava od 0 do *n*-1, a zatim se

vraća na početak. Zapazimo da parametri *n* i *w* nisu nezavisni, jer su za brojanje do *n*-1 u binarnom kodu potrebna $w=\lceil \log_2 n \rceil$ bita. Interfejs brojača čine signali takta (*clk*), asinhronog reseta (*rst*), dozvole rada (*en*), *w*-bitni izlaz za tekuće stanje brojača (*q*) i signal izlaznog prenosa (*cout*), koji je postavljen na ´1´ za sve vreme dok je brojač u stanju *n*-1. Generički opis mod_n brojača se može sintetizovati u brojač proizvoljne osnove, ali za razliku od brojača iz Pr. 5-22, nakon što je sintetizovan, on neće imati mogućnost programiranja osnove.



Sl. 7-6 (a) Generički mod_n brojač; (b) blok dijagram dvocifarskog dekadnog brojača.

```
-- Genericki mod_n brojac ---------------------------------------
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.NUMERIC_STD.ALL;
-----------------------------------------------------------------
ENTITY mod_n_counter IS
  GENERIC (N : NATURAL;
           W : NATURAL);
  PORT (clk,rst, en : IN STD_LOGIC;
        cout : OUT STD_LOGIC;
        q : OUT STD_LOGIC_VECTOR(W-1 DOWNTO 0));
END mod_n_counter;
-----------------------------------------------------------------
ARCHITECTURE arch OF mod_n_counter IS
  SIGNAL r_reg : UNSIGNED(W-1 DOWNTO 0);
  SIGNAL r_next : UNSIGNED(W-1 DOWNTO 0);
BEGIN
------ Registar -------------------------------------------------
  PROCESS(clk,rst)
  BEGIN
     IF(rst = '1') THEN
        r_reg <= (OTHERS => '0');
     ELSIF(clk'EVENT AND clk = '1') THEN
          IF(en = '1') THEN
           r_reg <= r_next;
          END IF;
     END IF;
  END PROCESS;
---- Logika sledeceg stanja i izlaza ----------------------------
  r_next <= (OTHERS => '0') WHEN r_reg = (N - 1) ELSE
            r_reg + 1;
  q <= STD_LOGIC_VECTOR(r_reg);
  cout <= '1' WHEN r_reg = (N-1) ELSE
          '0';
END arch;
```

Na Sl. 7-6(b) je prikazan blok dijagram dvocifarskog dekadnog brojača koji je realizovan povezivanjem dve instance brojača po modulu *n* s generičkim parametrima postavljenim na vrednosti: *n*=10 i *w*=4. Sledi odgovarajući VHDL opis.

```
-- Dvocifarski dekadni brojac ------------------------------------
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
----------------------------------------------------------------------
ENTITY two_digit_dec_counter IS
  PORT (clk,rst : IN STD_LOGIC;
        cout : OUT STD_LOGIC;
        q_1, q_10 : OUT STD_LOGIC_VECTOR(WIDTH-1 DOWNTO 0));
END two_digit_dec_counter;
----------------------------------------------------------------------
ARCHITECTURE generic_arch OF two_digit_dec_counter IS
   COMPONENT mod_n_counter IS
     GENERIC (N : NATURAL;
              WIDTH : NATURAL);
     PORT (clk,rst, en : IN STD_LOGIC;
           cout : OUT STD_LOGIC;
           q : OUT STD_LOGIC_VECTOR(WIDTH-1 DOWNTO 0));
   END COMPONENT;
   SIGNAL cout_1, cout_10 : STD_LOGIC;
BEGIN
 one_digit: mod_n_counter
   GENERIC MAP(N => 10, W => 4)
   PORT MAP (clk => clk, rst => rst, en => en,
             cout => cout_1, q => q_1);
 ten_digit: mod_n_counter
   GENERIC MAP(N => 10, W => 4)
   PORT MAP (clk => clk, rst => rst, en => en,
             cout => cout_10, q => q_10);
 cout <= cout_1 AND cout_10;
END generic_arch;
----------------------------------------------------------------------
```

Ovaj primer jasno ukazuje na potencijale kombinovanja generičkih parametara i komponenti. Umesto da kreiramo čitav niz brojača različitih osnova brojanja, koristimo samo jednu, ali parametrizovanu komponentu binarnog brojača, čija se osnova brojanja definiše tek prilikom instanciranja, posredstvom generičkog parametra. Uopšteno govoreći, u odnosu na neparametrizovane (tj. fiksne) komponente, generičke komponente su fleksibilnije i univerzalnije, s većom šansom da budu ponovno korišćene u budućim projektima. Parametrizovano projektovanje je tema glave 8.

7.3. CONFIGURATION

Deklaracija komponente sadrži samo osnovne informacije o komponenti, koje se tiču njenog interfejsa i eventualno generičkih parametara. Ove informacije su dovoljne za korišćenje komponente u novom kôdu. Međutim, deklaracija komponente ne sadrži nikakvu referencu na kôd komponente (tj. na entitet i arhitekturu komponente). U kontekstu VHDL-a, pod pojmom *konfiguracija* podrazumeva se povezivanje komponente sa odgovarajućim entitetom i arhitekturom. Proces konfiguracije se sprovodi u toku faze elaboracije dizajna (v. 2.4) i uključuje dva koraka: a) povezivanje komponente i entiteta i b)

povezivanje entiteta i arhitekture. U ovom pogledu, VHDL je veoma fleksibilan, jer kao što dozvoljava da za jedan isti entitet može postojati više različitih arhitektura, tako isto dozvoljava da se jedna ista komponenta, po potrebi, može povezati s jednim od, eventualno, više dostupnih entiteta. Projektant ostvaruje eksplicitnu kontrolu nad konfiguracijom (šta se sa čim povezuje) korišćenjem jezičke konstrukcije *configuration*, koja se piše kao nezavisna projekta jedinica. Sintaksa ove konstrukcije je složena. Međutim, u kôdu za sintezu se koristi samo mali deo mogućnosti koje ona pruža. Šta više, eksplicitna konfiguracija nije uvek neophodna. Na primer, ni u jednom primeru VHDL opisa iz ove glave konstrukcija za konfiguraciju nije korišćena, iako su u svim primerima korišćene komponente. Uvek kada za neku komponentu nije definisana konfiguracija, primenjuje se *podrazumevano* povezivanje, koje podrazumeva sledeće:

- Komponenta se povezuje sa istoimenim entitetom.
- Portovi komponente se povezuju sa istoimenim portovima entiteta.
- Ako postiji više arhitektura za isti entitet, tada se sa entitetom povezuje prva arhitektura koju je kompajler analizirao.

Treba naglasiti da u VHDL projektima namenjenim za sintezu, za jedan entitet po pravilu postoji samo jedna arhitektura. To znači da će podrazumevano povezivanje biti dovoljno u najvećem broju slučajeva ukoliko se poštuju pravila iz prve dve tačke.

Međutim, i u sintezi se povremeno javljaju situacije kad je potrebno baratati s više od jedne arhitekture za isti entitet. Na primer, prilikom projektovanja digitalnog hardvera često je potrebno pronaći optimalan kompromis između performansi i složenosti realizacije. Da bi postigao zadovoljavajući rezultat, projektant je u poziciji da realizuje nekoliko alternativnih rešenja istog modula i da svako rešenje ugrađuje u sistem radi procene njegovih performansi. Najefikasniji način kako se to može ostvariti jeste da se svako takvo rešenje realizuje u vidu zasebne arhitekture istog entiteta. Drugi primer višestrukih arhitektura se javlja kod testbenča (v. 2.4.1), gde se sistem koji se projektuje najpre opisuje na visokom apstraktnom nivou, a zatim postepeno razrađuje. Svaka nova, razrađenija varijanta sistema predstavlja novu arhitekturu koja se ugrađuje u isti testbenč.

Premda postoje i drugi načini za definisanje konfiguracije u VHDL-u, razmotrićemo samo jedan, koji je karakterističan za kôd namenjen za sintezu, a koji koristi tzv. *deklaraciju konfiguracije*. Deklaracija konfiguracije se piše kao nezavisna projektna jedinica (poput entiteta ili arhitekture), a njena pojednostavljena sintaksa je sledećeg oblika:

```
CONFIGURATION ime_konfiguracije OF ime_entiteta IS
  FOR ime_arhitekture
    FOR labela_instance : ime_komponente
      USE ENTITY ime_biblioteke.ime_entiteta(ime_arhitekture);
    END FOR;
    FOR labela_instance : ime_komponente
      USE ENTITY ime_biblioteke.ime_entiteta(ime_arhitekture);
    END FOR;
    . . .
  END FOR;
END ime_konfiguracije;
```

U gornjoj sintaksi, *ime_konfiguracije* je jedinstveno ime konfiguracione jedinice. *Ime_entiteta* iz prve i *ime_arhitekture* iz druge linije ukazuju na entitet i arhitekturu na koje se konfiguracija odnosi. *Labela_instance* ukazuje na jednu konkretnu instancu komponente *ime_komponente*, dok je u naredbi *use* koja sledi naveden par entitet-arhitektura koji se

povezuje s tom instancom. *Ime_biblioteke* je ime biblioteke u kojoj su smešteni entitet i arhitektura. Umesto konkretnog imena instance (*labela_instance*) dozvoljeno je korišćenje službene reči *all*. U tom slučaju, povezivanje definisano naredbom *use* se odnosi na sve instance date komponente.

Pr. 7-8 Konfigurabilni dvocifarski dekadni brojač

Razmotrimo ponovo projekat dvocifarskog dekadnog brojača iz Pr. 7-7. U arhitekturi dvocifarskog brojača, *generic_arch*, koriste se dve instance mod_n brojača koje su, pomoću generički parametara *n* i *w*, podešene za brojanje u osnovi 10. Pretpostavimo sada da je projekat proširen još jednom arhitekturom koja opisuje takođe brojač po modulu *n*, ali takav da za razliku od realizacije opisane postojećom arhitekturom, *arch*, ne broji naviše (0, ..., *n*-1, 0, 1...), već naniže (0, *n*-1, ..., 1, 0, *n*-1, ...). Kôd ove nove arhitekture, nazvane *arch_down*, dat je ispod. Zapazimo da je u odnosu na opis brojača naviše, jedina razlika u logici sledećeg stanja (linije 22 i 23) koja reguliše redosled stanja brojača i u logici izlaza (linije 25 i 26), jer se sada signal izlaznog prenosa javlja dok je brojač u stanju 0, a ne u stanju *n*-1 kao kod brojanja naviše.

```
1  -- Genericki mod_n brojac nanize ---------------------------------
2  LIBRARY IEEE;
3  USE IEEE.STD_LOGIC_1164.ALL;
4  USE IEEE.NUMERIC_STD.ALL;
5   ------------------------------------------------------------------
6  ARCHITECTURE arch_down OF mod_n_counter IS
7    SIGNAL r_reg : UNSIGNED(W-1 DOWNTO 0);
8    SIGNAL r_next : UNSIGNED(W-1 DOWNTO 0);
9  BEGIN
10 ----- Registar ------------------------------------------------
11   PROCESS(clk,rst)
12   BEGIN
13      IF(rst = '1') THEN
14        r_reg <= (OTHERS => '0');
15      ELSIF(clk'EVENT AND clk = '1') THEN
16        IF(en = '1') THEN
17          r_reg <= r_next;
18        END IF;
19      END IF;
20   END PROCESS;
21 --- Logika sledeceg stanja i izlaza ----------------------------
22   r_next <= N-1 WHEN r_reg = 0 ELSE
23             r_reg - 1;
24   q <= STD_LOGIC_VECTOR(r_reg);
25   cout <= '1' WHEN r_reg = 0 ELSE
26           '0';
27 END arch;
28  ------------------------------------------------------------------
```

S obzirom na to što sada u okviru istog projekta raspolažemo dvema arhitekturama, *arch* i *arch_down*, za isti entitet *mod_n_counter*, neophodno je napisati konfiguracionu jedinicu kako bi se izabrala verzija brojača koja će se koristiti u arhitekturi *generic_arch* entiteta *two_digit_dec_counter*. Ako je cilj da se realizuje dvocifarski dekadni brojač naniže, odgovarajuća konfiguraciona jedinica je oblika:

```
1 CONFIGURATION down_config OF two_digit_dec_counter IS
2   FOR generic_arch
3     FOR one_digit: mod_n_counter
4       USE ENTITY work.mod_n_counter(down_arch);
5     END FOR;
6     FOR ten_digit: mod_n_counter
7       USE ENTITY work.mod_n_counter(down_arch);
8     END FOR;
9   END FOR;
10 END down_config;
```

Budući da se obe instance mod_n brojača povezuju s istom arhitekturom, kôd konfiguracione jedinice se može uprostiti korišćenjem službene reči *all*:

```
1 CONFIGURATION down_config OF two_digit_dec_counter IS
2   FOR generic_arch
3     FOR ALL : mod_n_counter
4       USE ENTITY work.mod_n_counter(down_arch);
5     END FOR;
6   END FOR;
7 END down_config;
```

Naglasimo još jedanput da je se izbor između dve varijante dvocifarskog brojača vrši isključivo posredstvom konfiguracione jedinice i da se pri tom ne zahteva bilo kakva intervencija u kôdu brojača. Takođe, naglasimo da se sintezom ovog projekta ne kreira dvocifarski brojač s mogućnošću izbora smera brojanja, već da se radi o projektu koji se u zavisnosti od zahteva konkretne primene može sintetizovati bilo u brojač naviše, bilo u brojač naniže.

7.4. LIBRARY

Kao što je već rečeno u poglavlju 2.4, procesiranje VHDL projekta podrazumeva: razlaganje kôda na projektne jedinice (deklaracije entiteta, arhitekture, deklaracije konfiguracije itd.), prevođenje projektnih jedinica u međukôd i smeštanje međukôda u biblioteku. VHDL biblioteka se može razumeti kao neka vrsta virtuelnog skladišta procesiranih projektnih jedinica. Podrazumevano, za smeštanje projektnih jedinica koristi se tzv. radna biblioteka, tj. biblioteka *work*. Ova biblioteka je vidljiva samo u tekućem projektu i ne može se prenositi u druge projekte. Na primer, biblioteka *work* je korišćena u deklaraciji konfiguracije iz Pr. 7-8:

```
USE ENTITY work.mod_n_counter(down_arch);
```

Osim kao skladište za projektne jedinice tekućeg projekta, VHDL biblioteke se mogu koristiti i za prenošenje u druge projekte i ponovno korišćenje jednom napisanih i verifikovanih projektnih jedinica. Svakako je racionalnije i efikasnije često korišćene projektne jedinice čuvati u zajedničkoj biblioteci nego uvek iznova kopirati izvorni kôd takvih jedinica u svaki novi projekat.

Biblioteka se uključuje u VHDL kôd naredbom *library*, čije je sintaksa:

```
LIBRARY ime_biblioteke, ime_biblioteke, ..., ime_biblioteke;
```

Na primer, pretpostavimo da VHDL projekat dvocifarskog dekadnog brojača iz Pr. 7-7 ne sadrži kôd za mod_n brojač, već da su entitet i dve arhitekture ovog brojača smešteni u eksternoj biblioteci imena *c_lib*. U tome slučaju, kôd konfiguracione jedinice dvocifarskog brojača bi izgledao ovako:

```
1  LIBRARY c_lib;
2  CONFIGURATION down_config OF two_digit_dec_counter IS
3   FOR generic_arch
4    FOR one_digit: mod_n_counter
5      USE ENTITY c_lib.mod_n_counter(down_arch);
6    END FOR;
7    FOR ten_digit: mod_n_counter
8      USE ENTITY c_lib.mod_n_counter(down_arch);
9    END FOR;
10  END FOR;
11 END down_config;
```

Biblioteka *c_lib* je uključena naredbom *library* (linija 1). Time ova biblioteka postaje ″vidljiva″, a njen sadržaj dostupan za korišćenje u kôdu koji sledi. U linijama 5 i 8, umesto *work* sada piše *c_lib*, što znači da se komponenta *mod_n_counter* više ne povezuje sa entitetom *mod_n_counter* i arhitekturom *down_arch* iz tekućeg projekta, već sa istoimenim entitetom i arhitekturom iz biblioteke *c_lib*.

U vezi sa naredbom *library* je i naredba *use*, kojom se u projekat mogu selektivno uključivati pojedine projekte jedinice iz biblioteke, tipično one koje će često biti korišćene. Sintaksa naredbe *use*, onda kad se ona koristi za uključivanje projektnih jedinica, oblika je:

```
USE ime_biblioteke.ime_jedinice;
```

Efekat naredbe *use* je u tome da se ime projektne jedinice može pisati u kôdu konfiguracione jedinice bez navođenja imena biblioteke. Ako se u naredbi *use* umesto imena jedinice napiše službena reč *all*, direktan pristup biće omogućen svim projektnim jedinicama iz biblioteke. Na primeru deklaracije konfiguracije dvocifarskog dekadnog brojača, to izgleda ovako:

```
1  LIBRARY c_lib;
2  USE c_lib.mod_n_counter;
3  CONFIGURATION down_config OF two_digit_dec_counter IS
4    FOR generic_arch
5     FOR one_digit: mod_n_counter
6       USE ENTITY mod_n_counter(down_arch);
7     END FOR;
8     FOR ten_digit: mod_n_counter
9       USE ENTITY mod_n_counter(down_arch);
10    END FOR;
11   END FOR;
12 END down_config;
```

Kao što se može videti, u linijama 6 i 9 više nema imena biblioteke - zahvaljujući naredbi *use* iz linije 2, podrazumeva se da se entitet *mod_n_counter* i arhitektura *down_arch* uzimaju iz biblioteke *c_lib*.

Za razliku od eksternih biblioteka, koje se u projekat uključuju naredbom *library*, biblioteka *work* je implicitno uključena u svaki projekat. Zato u kôdu konfiguracione jedinice iz Pr. 7-7 ne postoji naredba *″library work″*.

7.5. Potprogrami

U VHDL-u, kao u većini programskih jezika, postoje dve vrste potprograma: funkcije i procedure. Osnovna razlika među njima je u tome što procedura može da vrati više od jedne izlazne vrednosti, dok funkcija vraća samo jednu. Takođe, svi parametri funkcije su ulazni, dok procedura može imati ulazne, izlazne i ulazno/izlazne parametre. Potprogrami u VHDL-u mogu da sadrže isključivo sekvencijalne naredbe i po načinu rada se mnogo ne razlikuju od potprograma iz tradicionalnih programskih jezika. Međutim, za razliku od entiteta i arhitektura, potprogrami se ne tretiraju kao projektne jedinice i zbog toga ne mogu nezavisno da se procesiraju. Na primer, funkciju nije moguće sintetizovati izolovano od ostatka VHDL kôda. Napomenimo da iako osnovni mehanizmi za kreiranje hijerarhije u softveru, funkcije i procedure nisu pogodno sredstvo za modeliranje hardverske hijerarhije.

7.5.1. FUNCTION

Funkcije se koriste, između ostalog, da bi se realizovale neke tipične složenije logičke operacije ili aritmetička izračunavanja, obavila konverzija iz jednog u neki drugi tip podataka, ili kreirali novi operatori i atributi. Jednom napisana, funkcija se može koristiti (pozivati) u VHDL kôdu proizvoljan broj puta. Na taj način, kôd postaje kompaktniji i razumljiviji. Funkcija, kao i proces, može sadržati samo sekvencijalne naredbe (*if*, *case* i *loop*). Izuzetak je *wait* naredba koja nije dozvoljena u funkciji ako se ona koristi u kôdu za sintezu. Takođe, u funkciji nije dozvoljeno deklarisati signale i instancirati komponente.

Deklaracija funkcije. Sintaksa deklaracije funkcije je sledećeg oblika:

```
FUNCTION ime_funkcije[<lista_parametara>] RETURN tip_podataka IS
   [deklaracije]
BEGIN
   sekvencijalna naredba;
   sekvencijalna naredba;
   . . .
   RETURN (izraz);
END ime_funkcije;
```

Deklaracija funkcije se sastoji iz zaglavlja i tela. Zaglavlje definiše ime funkcije, spisak ulaznih parametara i tip izlazne vrednosti funkcije, dok je u telu sadržan kôd funkcije. U gornjoj sintaksi, *lista_parametara* definiše ulazne parametre funkcije. Broj parametara može biti proizvoljan (uključujući i nulu). Parametri mogu biti konstante ili signali, dok varijable nisu dozvoljene, tj.:

```
<parametar> = CONSTANT ime_konstante : tip_konsatne;
<parametar> = SIGNAL ime_signala : tip_signala;
<parametar> = ime_konst_ili_sign : tip_konst_ili_sign;
```

Reč *conastant* napisana ispred imena parametra postavlja ograničenje da se pri pozivu funkcije kao stvarni parametar može koristiti samo konstanta. Slično, reč *signal* ispred imena parametra uvodi ograničenje da stvarni parametar može biti samo signal. Ako ispred imena parametra ne piše ni *constant* ni *signal*, ograničenja ovog tipa ne postoje, a stvarni parametar može biti bilo konstanta bilo signal. Tip parametra može biti bilo koji tip podatak koji se može sintetizovati (*boolean*, *std_logic*, *integer* itd.). Pri tom, za vektorske tipove ne treba navoditi opseg (npr. *range* za *integer* ili *to*/*downto* za *std_logic_vector*). S druge strane, funkcija može da vrati samo jednu vrednost, čiji je tip naveden iza ključne reči *return*. Na primer, razmotrimo zaglavlje sledeće funkcije:

```
FUNCTION f1 (a,b : INTEGER; SIGNAL c : STD_LOGIC_VECTOR) RETURN
BOOLEAN IS
   [deklaracije]
BEGIN
   (sekvencijalne naredbe)
END f1;
```

Ime funkcije je *f*1. Funkcija ima tri ulazna parametra, *a*, *b* i *c*. Parametri *a* i *b* su tipa *integer*, a *c* tipa *std_logic_vector*. Uočimo da uz *std_logic_vector* nije naveden opseg (*to* ili *downto*). Pri pozivu ove funkcije, kao parametri *a* i *b* se mogu koristiti i konstantne vrednosti i signali, dok se na mestu parametra *c* može naći samo signal. Izlazna vrednosti funkcije *f*1 je tipa *boolean*.

***Poziv funkcije*.** Poziv funkcije uvek je deo nekog složenijeg izraza, npr.:

```
x <= conv_integer(a);          -- konvertuje a u integer
y <= maximum(a,b);             -- vraca vece od a i b
IF(x > maximum(a,b)) THEN ... -- poredi x sa vecim od a i b
```

Pr. 7-9 Funkcija *positive_egde*()

Funkcija iz ovog primera detektuje rastuću (pozitivnu) ivicu takta. Slična je naredbi *if(clk'event and clk='1')*, koja se koristi za modeliranje sinhronih sekvencijalnih kola. Data funkcija ima jedan parametar, signal tipa *std_logic*, i vraća vrednost tipa *boolean*.

```
-- Deklaracija funkcije ---------------------------------------------
FUNCTION positive_edge (SIGNAL s : STD_LOGIC) RETURN BOOLEAN IS
BEGIN
   RETURN (s'EVENT AND s='1');
END positive_edge;
-- Poziv funkcije ---------------------------------------------------
. . .
IF positive_edge(clk) THEN ...
. . .
```

Pr. 7-10 Funkcija *conv_integer()*

Funkcija iz ovog primera konvertuje vrednost signala tipa *std_logic_vector* u odgovarajuću vrednost tipa *integer*. Na primer, za ulaz ″0110″ funkcija vraća 6. Kôd funkcije je u potpunosti generički, tj. ″radi″ za bilo koji opseg (*range*) i poredak (*to*/*downto*) ulaznog parametra *vector* koji je tipa *std_logic_vector*. Kao pomoćna promenljiva, u funkciji se koristi varijabla *b* tipa *integer*. Njen opseg odgovara opsegu binarnih brojeva dužine *vector'length* bita (linija 3). Na početku, *b* dobija vrednost bita najveće težine ulaznog vektora (linije 5-7). Zatim, u petlji, počev od pozicije prve manje težine od najveće, pa do pozicije najmanje težine (linija 8), *b* se množi sa 2 (linija 9) i uvećava za vrednost bita sa tekuće pozicije, *i*, u ulaznom vektoru (linije 10 i 11). Po izlasku iz petlje, vrednost varijable *b* jednaka je celobrojnom ekvivalentu ulaznog binarnog broja. Ova vrednost se vraća iz funkcije naredbom *return* (linija 13).

```
1 ---- deklaracija funkcije ----------------------------------------
2 FUNCTION conv_integer(SIGNAL vector: STD_LOGIC_VECTOR)
  RETURN INTEGER IS
3    VARIABLE b: INTEGER RANGE 0 TO 2**vector'LENGTH-1;
4 BEGIN
```

```
5     IF(vector(vector'HIGH)='1') THEN b:=1;
6     ELSE b:=0;
7     END IF;
8     FOR i IN(vector'HIGH-1) DOWNTO (vector'LOW) LOOP
9        b:=b*2;
10       IF(vector(i)='1')THEN b:=b+1;
11       END IF;
12    END LOOP;
13    RETURN b;
14 END conv_integer;
---- poziv funkcije -------------------------------------------------
. . .
y <= conv_integer(a);
. . .
```

Zapazimo da je ulazni parametar funkcije *conv_integer()*, *vector*, označen kao signal. To znači da bi poziv funkcije u kome bi se kao stvarni parametar navela konstanta umesto signala, npr. *y<=conv_integer(″00110011″)*, bio sintaksno neispravan. Funkcija za konverziju konstantnih vektora tipa *std_logic_vector* u *integer* imala bi zaglavlje:

```
FUNCTION conv_integer(CONSTANT vector: STD_LOGIC_VECTOR) RETURN
INTEGER IS
```

ili

```
FUNCTION conv_integer(vector: STD_LOGIC_VECTOR) RETURN INTEGER IS
```

Druga forma deklaracije parametra je opštija, jer dozvoljava da se pri pozivu funkcije kao ulazni parametar navede bilo konstanta bilo signal. Napomenimo da ne postoji oblik funkcije *conv_integer()* koji bi omogućio konverziju **varijable** tipa *std_logic_vector* u *integer* (zato što varijabla ne može biti ulazni parametar funkcije).

Pr. 7-11 Logaritam za osnovu 2

Funkcija iz ovog primera izračunava vrednost funkcije $\lceil \log_2 N \rceil$, gde je N ceo broj, a $\lceil\ \rceil$ označava ″veće celo od″. Za N=4, rezultat je 2, za N=8, rezultat je 3, za N=9 rezultat je 4 itd. Drugim rečima, funkcija vraća broj bita potrebnih za predstavljanje celog broja N u binarnom brojnom sistemu.

```
1  -- f-ja log2c(N) ------------------------------------------------
2    FUNCTION log2c(N : INTEGER) RETURN INTEGER IS
3      VARIABLE m, p : INTEGER;
4    BEGIN
5      m := 0; p := 1;
6      WHILE(p < N) LOOP
7        m := m + 1;
8        p := p * 2;
9      END LOOP;
10     RETURN m;
11   END log2c;
12 ------------------------------------------------------------------
```

Funkcija *log2c* se često koristi za određivanje dužine vektorskog signala. Na primer, u opisu parametrizovanog mod_n brojača iz Pr. 7-7 koriste se dva generička parametra, *n* koji definiše osnovu brojanja i *w* koji definiše broj izlaznih bita brojača. Očigledno, ova dva parametra nisu nezavisni s obzirom da važi $w = \lceil \log_2 n \rceil$. Ispod je ponovljena deklaracija

entiteta mod_n brojača iz Pr. 7-7, ali sada samo s jednim generičkim parametrom, *n*, dok se broj bita izlaznog porta *q* određuje pomoću funkcije *log2c*.

```
ENTITY mod_n_counter IS
  GENERIC (N : NATURAL);
  PORT (clk,rst, en : IN STD_LOGIC;
        cout : OUT STD_LOGIC;
        q : OUT STD_LOGIC_VECTOR(log2c(N)-1 DOWNTO 0));
END mod_n_counter;
```

Funkcije (i procedure) se najčešće pišu u paketu. Napisana u paketu, funkcija je vidljiva (tj. može se pozivati) svuda u projektu gde je paket uključen. Takođe, ako je to potrebno, funkcija se može pisati i unutar arhitekture ili entiteta. Međutim, u tom slučaju ona će biti vidljiva samo u arhitekturi/entitetu u kome je napisana. Kao što je već rečeno, ako paket sadrži potprograme, tada je neophodno da takav paket ima i telo, u kome će biti sadržana tela svih funkcija i procedura navedenih u deklarativnom delu paketa. U nastavku su dati primeri oba ova slučaja.

Pr. 7-12 Funkcija u arhitekturi

U VHDL kôdu koji sledi, funkcija *positiv_edge()* iz Pr. 7-10 napisana je u deklarativnom delu arhitekture (linije 11-14) i potom iskorišćena za detekciju rastuće ivice takta u telu arhitekture koja opisuje D flip-flop (linija 19).

```
1  ----------------------------------------------------------------------
2  LIBRARY IEEE;
3  USE IEEE.STD_LOGIC_1164.ALL;
4  ----------------------------------------------------------------------
5  ENTITY dff IS
6     PORT (d,clk,rst : IN STD_LOGIC;
7           q : OUT STD_LOGIC);
8  END dff;
9  ----------------------------------------------------------------------
10 ARCHITECTURE d_flip_flop OF dff IS
11   FUNCTION positive_edge (SIGNAL s : STD_LOGIC) RETURN BOOLEAN
12   BEGIN
13     RETURN (s'EVENT AND s='1');
14   END positive_edge;
15 BEGIN
16   PROCESS(clk,rst)
17   BEGIN
18     IF(rst='1') THEN q <= '0';
19     ELSIF positive_edge(clk) THEN q <= d;
20     END IF;
21   END PROCESS;
22 END d_flip_flop;
```

Pr. 7-13 Funkcija u paketu

Ovaj primer je sličan prethodno, s jedinom razlikom što je sada funkcija *positive_edge()* locirana u paketu. U sekciji *package*, funkcija se samo deklariše, dok se kompletan kôd funkcije (zaglavlje i telo funkcije) nalazi u sekciji *package body*. Da bi funkcija postala dostupna za pozivanje iz glavnog kôda, dovoljno je naredbom *use* uključiti odgovarajući paket.

```
----- paket: ------------------------------------------------------
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
-------------------------------------------------------------------
PACKAGE nas_paket IS
   FUNCTION positive_edge (SIGNAL s : STD_LOGIC) RETURN BOOLEAN;
END nas_paket;
-------------------------------------------------------------------
PACKAGE BODY nas_paket IS
   FUNCTION positive_edge (SIGNAL s : STD_LOGIC) RETURN BOOLEAN IS
   BEGIN
      RETURN (s'EVENT AND s='1');
   END positive_edge;
END nas_paket;
```

```
---- glavni kod ---------------------------------------------------
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE work.nas_paket.all;
-------------------------------------------------------------------
ENTITY dff IS
   PORT (d,clk,rst : IN STD_LOGIC;
         q : OUT STD_LOGIC);
END dff;
-------------------------------------------------------------------
ARCHITECTURE d_flip_flop OF dff IS
BEGIN
   PROCESS(clk,rst)
   BEGIN
      IF(rst='1') THEN q <= '0';
      ELSIF positive_edge(clk) THEN q <= d;
      END IF;
   END PROCESS;
END d_flip_flop;
```

Pr. 7-14 Preklapanje operatora ″+″

Funkcija iz ovog primera, nazvana ″+″, preklapa predefinisani operator sabiranja (″+″). Predefinisano ″+″ se može primenjivati samo nad sabircima tipa *integer*, *signed* ili *unsigned* (v. 3.3). Nova funkcija omogućiće sabiranje dve vrednosti tipa *std_logic_vector*. Funkcija je smeštena u paketu *nas_paket*. Primer korišćenja funkcije dat je u kôdu koji sledi posle paketa. Funkcija ima dva ulazna parametra koji su kao i izlazna vrednost tipa *std_logic_vector*. Pretpostavićemo da su dužine oba ulazna i izlaznog vektora jednake. Funkcija obavlja sabiranje ulaznih binarnih vektora bit-po-bit, počev od bita najmanje težine. Za svaku bitsku poziciju, određuje se bit sume, *sum*(*i*), i bit prenosa, *carry*, koji se pridodaje sledećoj poziciji.

```
---- paket -------------------------------------------------------
 LIBRARY IEEE;
 USE IEEE.STD_LOGIC_1164.ALL;
-------------------------------------------------------------------
 PACKAGE nas_paket IS
    FUNCTION "+" (a,b: STD_LOGIC_VECTOR) RETURN STD_LOGIC_VECTOR;
 END nas_paket;
```

```
8  ------------------------------------------------------------------
9  PACKAGE BODY nas_paket IS
10   FUNCTION "+" (a,b: STD_LOGIC_VECTOR) RETURN STD_LOGIC_VECTOR IS
11      VARIABLE sum: STD_LOGIC_VECTOR(a'RANGE);
12      VARIABLE carry: STD_LOGIC;
13   BEGIN
14      carry := '0';
15      FOR i IN a'REVERSE_RANGE LOOP
16         sum(i) := a(i) XOR b(i) XOR carry;
17         carry  := (a(i) AND b(i)) OR (a(i) AND carry)
18                   OR (b(i) AND carry);
19      END LOOP;
20      RETURN sum;
21   END "+";
22 END nas_paket;
1  ------adder.vhd ------------------------------------------------
2  LIBRARY IEEE;
3  USE IEEE.STD_LOGIC_1164.ALL;
4  USE WORK.NAS_PAKET.ALL;
5  ------------------------------------------------------------------
6  ENTITY adder IS
7    PORT ( a : IN STD_LOGIC_VECTOR(3 DOWNTO 0);
8           y : OUT STD_LOGIC_VECTOR(3 DOWNTO 0));
9  END adder;
10 ------------------------------------------------------------------
11 ARCHITECTURE adder OF adder IS
12    CONSTANT b : STD_LOGIC_VECTOR(3 DOWNTO 0) := "0110";
13 BEGIN
14      y <= a + b;
15 END adder;
16 ------------------------------------------------------------------
```

Treba napomenuti da postojanje funkcije za preklapanje operatora ″+″ ne znači da je predefinisani operator ″+″ zamenjen novim, već da je njegova primena proširena na novi tip podataka. To znači da i tamo gde je paket *nas_paket* uključen i dalje možemo koristiti ″+″ za sabiranje brojeva tipa *integer*, *signed* ili *unsigned*. Koja od dve funkcije će biti izvršena, predefinisana ili preklopljena, zavisi od tipa operanada na koje je operator ″+″ primenjen.

7.5.2. PROCEDURE

Procedura je slična funkciji i u osnovi ima istu namenu. Međutim, za razliku od funkcije, procedura može da vrati više od jedne vrednosti. Kao i funkcija i procedura se prvo deklariše, a zatim poziva u kôdu.

Deklaracija procedure. Sledi sintaksa deklaracije procedure.

```
PROCEDURE ime_procedure [<lista_parametara>] IS
   [deklaracije]
BEGIN
   (sekvencijalne naredbe)
END ime_procedure;
```

U gornjoj sintaksi, *lista_parametara* sadrži ulazne i izlazne parametre procedure:

```
<parametar> = [CONSTANT] ime_konstante : smer tip_konstante;
```

```
<parametar> = SIGNAL ime_signala : smer tip_signala;
<parametar> = VARIABLE ime_signala : smer tip_varijable;
```

Procedura može imati proizvoljan broj ulaznih (*smer=in*), izlaznih (*smer=out*) i ulazno/izlaznih (*smer =inout*) parametara. Svi oni mogu biti signali, varijable ili konstante. Kod funkcija koje se koriste u kôdu za sintezu, deklaracije *wait*, *signal* i *component* nisu dozvoljene. Isto važi i za procedure, s izuzetkom da signali mogu biti deklarisani u proceduri, ali samo pod uslovom da je ona deklarisana u procesu. Osim toga, kao ni *wait*, tako ni bilo koji drugi način za detekciju ivice signala ne može biti sintetizovan ako je deo procedure (npr. konstrukcija tipa *if(clk'event and clk='1')* nije dozvoljena u proceduri).

Na primer, procedura čije je zaglavlje prikazano ispod ima tri ulazna parametra, *a*, *b* i *c* (*smer =in*). Parametar *a* je konstanta tipa *bit*, dok su *b* i *c* signali istog tog tipa. Uočimo da za ulazne parametre ključna reč *constant* nije neophodna. Procedura vraća dve vrednosti kroz signale *x* (*smer=out* i *tip=bit_vector*) i *y* (*smer=inout*, *tip=integer*).

```
PROCEDURE nasa_procedura(a: IN BIT; SIGNAL b,c: IN BIT;
                         SIGNAL x: OUT BIT_VECTOR(7 DOWNTO 0);
                         SIGNAL y: INOUT INTEGER RANGE 0 TO 99) IS
BEGIN
. . .
END nasa_procedura;
```

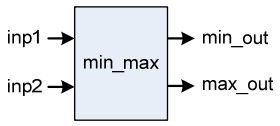
Poziv procedure. Za razliku od funkcija, čiji je poziv je uvek deo nekog izraza, pozivi procedura se mogu tretirati kao nezavisne naredbe, koje mogu egzistirati samostalno ili biti pridružene nekoj drugoj naredbi, npr.:

```
min_max(in1, in2, in3, out1, out2);
deljenje(deljenik, delilac, kolicnik, ostatak);
IF(a>b) THEN min_max(in1,in2,in3,out1,out2);
```

Procedure se pišu na istim mestima u VHDL kôdu gde i funkcije. Kao i za funkcije, tipično mesto za procedure je u paketu, ali po potrebi procedura može biti locirana i u glavnom kôdu (u deklarativnim sekcijama entiteta, arhitekture i procesa). Paket koji sadrži proceduru mora imati telo (*package body*), u kome će biti napisana tela svih procedura deklarisanih u deklarativnom delu paketa.

Pr. 7-15 Procedura u arhitekturi

VHDL kôd iz ovog primera opisuje *min_max* kolo (Sl. 7-7) uz pomoć procedure koja se zove *sort*. Procedura *sort* (linije 11-21) ima dva ulazna, *inp1* i *inp2*, i dva izlazna parametra, *min_out* i *max_out*, svi tipa *std_logic_vector*. U telu procedure, dva ulazna bit-vektori se najpre porede, konvertovani u neoznačene cele brojeve, a zatim se na izlaz *min_out* prenosi manji, a na izlaz *max_out* veći bit-vektor. Zapazimo da je procedura locirana u deklarativnom delu arhitekture.



Sl. 7-7 min_max kolo.

```
1  --- Min-max kolo: primer procedure u arhitekturi -----------------
2  LIBRARY IEEE;
3  USE IEEE.STD_LOGIC_1164.ALL;
4  USE IEEE.NUMERIC_STD.ALL;
```

```
5 ----------------------------------------------------------------------
6 ENTITY min_max IS
7   PORT (inp1,inp2 : IN STD_LOGIC_VECTOR(7 downto 0);
8         min_out, max_out: OUT STD_LOGIC_VECTOR(7 downto 0));
9 END min_max;
10 ----------------------------------------------------------------------
11 ARCHITECTURE nasa_arhitektura OF min_max IS
12  PROCEDURE sort(SIGNAL in1,in2 : IN STD_LOGIC_VECTOR(7 downto 0);
13                 SIGNAL min, max: OUT STD_LOGIC_VECTOR(7 downto 0))
14 IS
15  BEGIN
16   IF(UNSIGNED(in1) > UNSIGNED(in2)) THEN
17      max <= in1; min <= in2;
18   ELSE
19      max <= in2; min <= in1;
20   END IF;
21  END sort;
22 BEGIN
23  sort(inp1,inp2,min_out,max_out);
24 END nasa_arhitektura;
```

Pr. 7-16 Procedura u paketu

Ovaj primer je sličan prethodnom, s jedinom razlikom što je sada procedura *sort* smeštena u paketu, a ne u arhitekturi. Na taj način, ova procedura se može koristiti u različitim delovima projekta. VHDL kôd je raspodeljen na dve datoteke, jedna sadrži kôd za paket, a druga glavni kôd.

```
1 ----- Paket: ------------------------------------------------------
2 LIBRARY IEEE;
3 USE IEEE.STD_LOGIC_1164.ALL;
4 USE IEEE.NUMERIC_STD.ALL;
5 ----------------------------------------------------------------------
6 PACKAGE nas_paket IS
7  PROCEDURE sort(SIGNAL in1,in2 : IN STD_LOGIC_VECTOR(7 downto 0);
8            SIGNAL min, max: OUT STD_LOGIC_VECTOR(7 downto 0));
9 END nas_paket;
10 ----------------------------------------------------------------------
11 PACKAGE BODY nas_paket IS
12  PROCEDURE sort(SIGNAL in1,in2 : IN STD_LOGIC_VECTOR(7 downto 0);
13                 SIGNAL min, max: OUT STD_LOGIC_VECTOR(7 downto 0))
14  IS
15  BEGIN
16   IF(UNSIGNED(in1) > UNSIGNED(in2)) THEN
17      max <= in1; min <= in2;
18   ELSE
19      max <= in2; min <= in1;
20   END IF;
21  END sort;
22 END nas_paket;
```

```
1 -- Glavni kod ------------------------------------------------------
2 LIBRARY IEEE;
3 USE IEEE.STD_LOGIC_1164.ALL;
4 USE WORK.NAS_PAKET.ALL;
```

```
5  ----------------------------------------------------------------------
6  ENTITY min_max IS
7    PORT (inp1,inp2 : IN STD_LOGIC_VECTOR(7 downto 0);
8          min_out, max_out: OUT STD_LOGIC_VECTOR(7 downto 0));
9  END min_max;
10 ----------------------------------------------------------------------
11 ARCHITECTURE nasa_arhitektura OF min_max IS
12 BEGIN
13   sort(inp1,inp2,min_out,max_out);
14 END nasa_arhitektura;
```

7.5.3. Poređenje funkcija i procedura

- Funkcija može imati nijedan, jedan ili više ulaznih parametara i samo jednu izlaznu vrednost. Ulazni parametri funkcije mogu biti konstante i signali, ali ne i varijable. Procedura može imati proizvoljan broj *in*, *out* ili *inout* parametara, koji mogu biti signali, konstante ili varijable.
- Poziv funkcije je uvek deo nekog izraza, dok je poziv procedure naredba za sebe.
- Za funkcije, kao i za procedure, važi da se naredbe *wait* i *component* ne mogu sintetizovati. U funkciji je dozvoljeno korišćenje jezičkih konstrukcija za detekciju ivice signala, dok u proceduri nije.
- Procedure i funkcije se mogu pisati na istim mestima u kôdu. Uobičajeno su smeštene u paketu (iz razloga ponovnog korišćenja), ali mogu biti locirane i u glavnom kôdu (u deklarativnim sekcijama arhitekture, entiteta ili procesa). Paket koji sadrži funkcije ili procedure, mora imati i telo (*package body*) u kome će biti smeštena tela svih funkcija/procedura deklarisanih deklarativnom delu paketa.

Procedure u VHDL-u se mogu razumeti kao neka vrsta alternative komponentama. Međutim, komponente su prirodniji i efikasniji način za modeliranje hardverske hijerarhije, pa se iz tog razloga procedure retko koriste u kôdu za sintezu. Za razliku od procedura, funkcije nalaze širu primenu, koja je uglavnom u vezi sa konverzijom tipova, preklapanjem operatora i jednostavnim numeričkim izračunavanjima.

8. PARAMETRIZOVANO PROJEKTOVANJE

Kako digitalni sistemi postaju složeniji, tako je sve teže projektovati svaki novi digitalni sistem iz početka. Međutim, mnogi sistemi, iako svaki sa svojim specifičnostima, sadrže delove identične ili slične funkcionalnosti. Imajući to u vidu, jedan od glavnih ciljeva digitalnog projektovanja danas postoje ″projektovanje za ponovno korišćenje″ (engl., *design reuse*). Od projektanta se očekuje da za realizaciju sistema, svuda tamo gde je to moguće, koristi ranije projektovane module, a da module koje samostalno projektuje realizuje na takav način da se lako mogu ponovno koristiti. Pretpostavka projektovanja za ponovno korišćenje jeste u mogućnosti parametrizacije projektnih modula. Umesto da se projektuju komponente fiksnih karakteristika i funkcionalnosti, racionalnije je projektovati ″univerzalne″ komponente, koje će pri ugradnji u novi sistem moći u izvesnoj meri i na lak način da se prilagode specifičnim zahtevima nove primene. Na primer, brojač je komponenta koja se može naći u gotovo svakom digitalnom sistemu. Iako postoje različite vrste brojača, svi oni poseduju identičnu bazičnu konstrukciju, a tipično se razlikuju po osnovi i smeru brojanja. Šanse za ponovno korišćenje 17-bitnog brojača unazad su svakako male. Međutim, brojač koji je projektovan na način da se može konfigurisati za rad u željenoj osnovi i smeru brojanja, naći će daleko širu primenu. Ovo se postiže tako što se pojedini aspekti rada kola (poput osnova brojanja brojača) opisuju u zavisnosti od eksternih parametara.

Ova glava je posvećena principima parametrizovanog projektovanja u VHDL-u. VHDL poseduje brojne jezičke konstrukcije i mehanizme za parametrizovano projektovanje. S nekim od njih smo se upoznali u prethodnim glavama (*generic*, atributi i konstante), dok će drugi, kao npr. naredba *generate*, biti uvedeni u ovoj glavi.

8.1. Vrste parametara

Iz perspektive parametrizovanog projektovanja, parametri se mogu grubo podeliti na dimenzione i funkcionalne. Kod različitih digitalnih sistema postoje različiti zahtevi u pogledu broja bita koji se koriste za predstavljanje podataka. Tako na primer možemo govoriti o 8-, 16- ili 32-bitnoj ALU, odnosno o jedinici koja obavlja aritmetičke i logičke operacija nad 8-, 16- i 32-bitnim podacima. Dimenzioni parametri upravo služe za definisanje veličine, odnosno broja bita višebitnih portova i signala koji se koriste za razmenu podataka sa sistemom, odnosno za prenos i čuvanje podataka unutar sistema. Pri tom sistem može posedovati jedan ili više dimenzionih parametara. Na primer, RAM se

dimenzioniše s dva parametra, od kojih jedan određuje broj bita u memorijskoj reči, a drugi ukupan broj memorijskih reči. Osim toga što služe za definisanje veličine ulaznih i izlaznih portova, dimenzioni parametri direktno određuju i veličinu unutrašnjih jedinica sistema, poput registara, sabirača i sl., a da pri tom ne utiču na strukturnu organizaciju sistema (tj. na to kako su unutrašnje jedinice međusobno povezane).

S druge strane, funkcionalni parametri se koriste za specifikaciju strukture ili organizacije sistema s ciljem da se omogući uključivanje/isključivanje pojedinih funkcija ili izabor između nekoliko raspoloživih varijanti implementacije sistema. Po pravili, pomoću funkcionalnih parametara postižu se male varijacije u funkcionalnosti/strukturi sistema. Na primer, uz pomoć funkcionalnog parametra moguće je regulisati da će se u sistemu koristiti sinhroni ili asinhroni reset i sl. U principu, funkcionalne parametre je moguće koristiti i za izbor između nekoliko potpuno različitih realizacija istog kola. Na primer, da li koristiti sabirač s rednim prenosom ili sabirač sa ubrzanim prenosom. Da bi se to postiglo, odgovarajući VHDL kôd bi morao da sadrži dva gotovo potpuno nezavisna opisa u okviru iste arhitekture. U takvim situacijama, bolje je alternativne realizacije opisati u vidu zasebnih arhitektura, a za izbor jedne od njih koristiti konfiguraciju (v. 7.3).

8.2. Specifikacija parametara

VHDL poseduje nekoliko jezičkih konstrukcija za specifikaciju parametara, od kojih su sa stanovišta sinteze najbitniji: generički parametri i atributi vektora.

Generički parametri. Generički parametri su razmatrani u poglavlju 7.2, u kontekstu komponenti. Generički parametri se deklarišu u entitetu, a koriste u arhitekturi, u vidu simboličkih konstanti. Ako se kasnije entitet koristi kao komponenta, stvarne vrednosti generičkih parametara će biti navedene prilikom instanciranja komponente.

Pr. 8-1 Parametrizovan opis generatora bita parnosti

VHDL opis 8-bitnog generatora parnosti predstavljen je u Pr. 5-14 iz poglavlja 5.4. Opis iz Pr. 5-14 se može lako parametrizovati, ako se numeričke konstante iz deklaracije entiteta i kôda arhitekture zamene generičkim parametrom, na način kao u kôdu koji sledi.

```
1  ----------------------------------------------------------------------
2  LIBRARY IEEE;
3  USE IEEE.STD_LOGIC_1164.ALL;
4  ----------------------------------------------------------------------
5  ENTITY pargen IS
6    GENERIC(N: INTEGER := 8);
7    PORT (a : IN STD_LOGIC_VECTOR(N-1 DOWNTO 0);
8          y : OUT STD_LOGIC);
9  END pargen;
10 ----------------------------------------------------------------------
11 ARCHITECTURE param_arch OF pargen IS
12 BEGIN
13  PROCESS(a)
14    VARIABLE p : STD_LOGIC;
15  BEGIN
16    p := a(0);
17    FOR i IN 1 TO (N-1) LOOP
18      p := p XOR a(i);
```

```
19    END LOOP;
20    y <= p;
21  END PROCESS;
22 END param_arch;
23 ------------------------------------------------------------------
```

Generički parametar *N*, deklarisan u liniji 6, iskorišćen je dvaput: u liniji 7, za definisanje ″širine″ ulaznog porta *a* i u liniji 17, kao gornje granica indeksa u *for loop* petlji. Na taj način, opis je postao univerzalan, u smislu da više ne modelira neki konkretan generator bita parnosti, već generator parnosti za proizvoljan broj ulaznih bita. Konkretizacija se viši prilikom instanciranja komponente izvedene na osnovu generičkog opisa.

Zapazimo da deklaracija iz linije 7 sadrži podrazumevanu vrednost generičkog parametra *N* (konkretno, *N:*=8). Podrazumevana vrednost, ako postoji, koristi se onda kada naredba za instanciranje komponente ne sadrži konstrukciju *generic map*, kojom se postavljaju stvarne vrednosti generičkih parametara. Takođe, pri direktnoj sintezi generičkog VHDL opisa, biće korišćene podrazumevane vrednosti generičkih parametara.

Atributi vektora. VHDL atributi su uvedeni u poglavlju 3.5. Za primenu u parametrizovanom projektovanju od značaja su atributi vektora, koji pružaju informaciju o opsegu i graničnim vrednostima indeksa vektora. Radi podsećanja, u Pr. 8-2 određene su vrednosti atributa za dva vektorska signala. U Pr. 8-3 je pokazano kako se atributi vektora mogu koristiti kao parametri u VHDL kôdu.

Pr. 8-2 Atributi vektora

Za signale:

```
SIGNAL s1 : STD_LOGIC_VECTOR(15 DOWNTO 0);
SIGNAL s2 : STD_LOGIC_VECTOR(8 TO 15);
```

važi:

```
s1'LEFT = 15; s1'RIGHT = 0;       S2'LEFT = 8; s2'RIGHT = 15;
s1'LOW = 0; s1'HIGH = 15;         s2'LOW = 8; s2'HIGH = 15;
s1'LENGTH = 16;                   s2'LENGTH = 8;
s1'RANGE = 15 DOWNTO 0;           s2'RANGE = 8 TO 15;
S1'REVERSE_RANGDE = 0 TO 15;      s2'REVERSE_RANGDE = 15 DOWNTO 8;
```

Pr. 8-3 Parametrizovan opis generatora parnosti korišćenjem atributa vektora

U arhitekturu iz opisa generatora parnosti koji sledi iskorišćen je atribut *'length*, koji vraća dužinu vektora. Praktično, *a'length* igra ulogu parametra *N* iz Pr. 8-1.

```
1  ------------------------------------------------------------------
2  LIBRARY IEEE;
3  USE IEEE.STD_LOGIC_1164.ALL;
4  ------------------------------------------------------------------
5  ENTITY pargen IS
6    GENERIC(N: INTEGER := 8);
7    PORT (a : IN STD_LOGIC_VECTOR(N-1 DOWNTO 0);
8          y : OUT STD_LOGIC);
9  END pargen;
10 ------------------------------------------------------------------
```

```
11 ARCHITECTURE param_arch OF pargen IS
12 BEGIN
13  PROCESS(a)
14    VARIABLE p : STD_LOGIC;
15  BEGIN
16    p := a(0);
17    FOR i IN 1 TO (a'length-1) LOOP
18      p := p XOR a(i);
19    END LOOP;
20    y <= p;
21  END PROCESS;
22 END param_arch;
23 ------------------------------------------------------------------
```

Opseg indeksa u naredbe *for loop* (linija 17) mogao je biti izražen i pomoću drugih atributa vektora *a*, npr.:

```
FOR i IN (a'low + 1) TO a'high LOOP
```
ili
```
FOR i IN (a'right + 1) TO a'left LOOP
```

Vektori s nedefinisanim opsegom. Upotreba atributa vektora u Pr. 8-3 je donekle redundantna, jer ako je generički parametar *N* već deklarisan u entitetu generatora parnosti, parametrizovane vrednosti iz arhitekture se jasnije mogu izraziti preko *N* (kao u Pr. 8-1). Pravu primenu atributi vektora nalaze u kombinaciji s vektorima sa nedefinisanim opsegom.

Za pisanje VHDL-a za sintezu najčešće se koriste sledeća tri tipa podataka: *std_logic_vector*, *unsigned* i *signed*. Ovi tipovi podataka su definisani u paketima *std_logic_1164* i *numeric_std*, i to kao vektori (nizovi) sa nedefinisanim opsegom indeksa (engl. *unconstrained array*), npr.:

```
TYPE STD_LOGIC_VECTOR IS ARRAY(NATURAL RANGE <>) OF STD_LOGIC;
```

Iz prethodne deklaracije se vidi da indeksi vektora tipa *std_logic_vector* moraju biti tipa *natural*, ali tačne granice opsega indeksa nisu definisane. U takvim slučajevima, granice se navode prilikom deklaracije objekta (signala, varijable ili konstante) ovog tipa, npr.:

```
SIGNAL s1 : STD_LOGIC_VECTOR(15 DOWNTO 0);
```

Deklaracija porta je jedini izuzetak od ovog pravila. Port, za razliku od signala, može biti deklarisan bez navođenja opsega, kao u sledećem primeru.

Pr. 8-4 Sabirač bez granica

VHDL kôd koji sledi predstavlja opis sabirača neoznačenih binarnih brojeva. Kôd je ispravan iako u deklaraciji portova granice opsega indeksa nisu navedene.

```
1  ------------------------------------------------------------------
2  LIBRARY IEEE;
3  USE IEEE.STD_LOGIC_1164.ALL;
4  USE IEEE.NUMERIC_STD.ALL;
5  ------------------------------------------------------------------
6  ENTITY unconstrain_adder IS
7     PORT(a : IN STD_LOGIC_VECTOR;
8          b : IN STD_LOGIC_VECTOR;
9          c : OUT STD_LOGIC_VECTOR);
10 END unconstrain_adder;
```

```
11 ------------------------------------------------------------------
12 ARCHITECTURE arch OF unconstrain_adder IS
13 BEGIN
14    c <= STD_LOGIC_VECTOR(UNSIGNED(a) + UNSIGNED(b));
15 END arch;
```

Sabirač iz Pr. 8-4 se naravno ne može sintetizovati, ali se može koristiti kao komponenta, jer će prilikom instanciranja, portovi sabirača preuzeti opsege stvarnih signala. Na primer, u sledećem segmentu VHDL kôda instancira se jedan 8-bitni sabirač.

```
. . .
SIGNAL a1, b1, c1: STD_LOGIC_VECTOR(7 DOWNTO 0);
. . .
add8 : unconstrain_adder
  PORT MAP(a=>a1, b=>b1, c=>c1);
```

Opis sabirača iz Pr. 8-4 je parametrizovan iako bez eksplicitno navedenih parametara. Možemo razumeti da je dimenzioni parametar *N* sadržan u stvarnom signalu koji se prosleđuje deklaraciji entiteta u momentu instanciranja odgovarajuće komponente.

U sledećem primeru, ilustrovana je upotreba atributa vektora za pisanje parametrizovanog kôda u slučaju kad portovi nemaju definisan opseg.

Pr. 8-5 Generator bita parnosti s portovima bez definisanog opsega

U kôdu koji sledi, ulazni port *a* generatora bita parnosti je deklarisan bez navođenja opsega indeksa (linija 6). Kao što možemo videti, deklaracija entiteta više ne sadrži deklaraciju generičkog parametra *N*. Umesto toga, informacija o dužini je ekstrahovana iz signala *a* atributom *length* i dodeljena konstanti *N* (linija 11), koja se potom u telu arhitekture koristi na identičan način kao istoimeni generički parametar u kôdu iz Pr. 8-1.

```
1  ------------------------------------------------------------------
2  LIBRARY IEEE;
3  USE IEEE.STD_LOGIC_1164.ALL;
4  ------------------------------------------------------------------
5  ENTITY unconstrain_pargen IS
6    PORT (a : IN STD_LOGIC_VECTOR;
7          y : OUT STD_LOGIC);
8  END unconstrain_pargen;
9  ------------------------------------------------------------------
10 ARCHITECTURE arch OF unconstrain_pargen IS
11  CONSTANT N : NATURAL := a'LENGTH;
12  VARIABLE p : STD_LOGIC;
13 BEGIN
14  PROCESS(a)
15  BEGIN
16    p := a(0);
17    FOR i IN 1 TO (N-1) LOOP
18      p(i) := p XOR a(i);
19    END LOOP;
20    y <= p;
21  END PROCESS;
22 END arch;
```

Generički parametri i vektori bez definisanog opsega indeksa su u osnovi mehanizmi za specifikaciju dimenzionih parametara. Pri tom, vektori bez definisanog opsega su opštiji i fleksibilniji od generičkih parametar, zato što se informacija o dužini automatski ekstrahuje iz signala, bez potrebe eksplicitnog uvođenja dimenzionog parametra. Međutim, korišćenje vektora bez definisanog opsega može da dovede do grešaka koje su posledica neusklađenosti opsega stvarnih signala i opsega koji je pretpostavljen prilikom pisanja parametrizovanog kôda.

Primera radi, u kôdu koji sledi namera je da se instancira 8-bitni generator bita parnosti iz Pr. 8-5 i koristiti za određivanje parnosti višeg bajta dvobajtnog signala *a*1. Iako je kôd sintaksno ispravan, greška će se javiti tokom sinteze. Problem je u opsegu indeksa signala *a1* koji se vezuje na port *a* generatora parnosti. Pošto port *a* preuzima opseg stvarnog signala *a1*, tj. (15 *downto* 8), greška će se javiti u naredbi dodele iz linije 16, kao i u *for loop* petlji u kôdu iz Pr. 8-5, zato što *a* ne sadrži indekse od 0 do 7.

```
. . .
SIGNAL a1 : STD_LOGIC_VECTOR(15 DOWNTO 0);
SIGNAL y1 : STD_LOGIC;
. . .
pargen8 : unconstrain_pargen
  PORT MAP(a=>a1(15 DOWNTO 8), y=>y1);
. . .
```

Opisana greška se ne bi javila da je kôd arhitekture iz Pr. 8-5 napisan na nešto drugačiji način:

```
ARCHITECTURE arch OF unconstrain_pargen IS
BEGIN
   PROCESS(a)
     VARIABLE p : STD_LOGIC;
   BEGIN
     p := a(a'LOW);
     FOR i IN (a'LOW+1) TO a'HIGH LOOP
       p := p XOR a(i);
     END LOOP;
     y <= p;
   END PROCESS;
END arch;
```

Sada u kôdu više ne figuriše atribut za dužinu signala, već se umesto njega koriste atributi za granične vrednosti indeksa, *low* i *high*. Zbog toga, opseg indeksa stvarnog signala može biti proizvoljan.

U kôdu koji sledi pokazan je jedan jednostavniji način kako se parametrizovan opis generatora parnosti iz Pr. 8-5 može učiniti univerzalnijim. Kao što vidimo, uveden je pomoćni, interni signal *aa* koji ima dužinu ulaznog porta *a*, ali sa opsegom indeksa (*N-1 downto* 0). U telu arhitekture, najpre se signalu *aa* dodeljuje vrednost ulaznog signala *a,* a onda se u nastavku umesto sa *a*, manipuliše sa *aa*.

```
ARCHITECTURE arch OF unconstrain_pargen IS
  CONSTANT N : NATURAL := a'LENGTH;
  SIGNAL aa : STD_LOGIC_VECTOR(N-1 DOWNTO 0);
BEGIN
  aa <= a;
  PROCESS(aa)
```

```
  VARIABLE p : STD_LOGIC;
BEGIN
    p := aa(0);
    FOR i IN 1 TO (N-1) LOOP
       p := p XOR aa(i);
    END LOOP;
    y <= p;
  END PROCESS;
END arch;
```

Logičke i aritmetičke operacija nad vektorskim tipovima *std_logic_vector*, *unsigned* i *signed* definisane su u paketima *std_logic_1164* i *numeric_std* kao preklopljeni operatori nad vektorima bez definisanog opsega. Zbog toga, operandi ovih operatora mogu biti i vektori proizvoljne dužine, kao u sledećem kôdu:

```
a <=  b + c WHEN b < c ELSE
      b - c;
```

Pošto u prethodnoj naredbi ne figuriše informacija o dužini vektora, može se smatrati da je kôd ″implicitno″ parametrizovan. Prethodna naredba, izolovano posmatrana, ne pruža informaciju o dužini, ali je ta informacija, koja je inače neophodna za sintezu, poznata iz konteksta u kojem se naredba nalazi.

U zaključku, parametrizovan opis koji se oslanja na vektore bez definisanog opsega, iako fleksibilan za korišćenje, mora biti pažljivo pisan i analiziran kako bi se predupredile eventualne greške koje mogu nastati zbog neusklađenosti formata signala. Iz tog razloga, za pisanje parametrizovanog kôda preporučuju se generički parametri, osim ukoliko se ne radi o krajnje generalnim strukturama, poput sabirača iz Pr. 8-4.

8.3. Manipulacije s vektorima

Zamena fiksnih referenci na elemente ili delove vektora, atributima i generičkim parametrima, jedna od osnovnih tehnika za parametrizovano projektovanja. U ovom poglavlju, na nekoliko primera biće ukazano na mogućnosti VHDL-a koje se tiču pisanja parametrizovanog kôda za manipulaciju vektorima.

Pr. 8-6 Korišćenje agregacija

U sekvencijalnom kôdu često postoji potreba za dodelom konstante vrednosti signalu ili varijabli. Na primer, resetovanje 8-bitnog brojača zahteva sledeću naredbu:

```
q <= "00000000";
```

Jasno je da se prethodna naredba uvek mora modifikovati kad se, iz bilo kog razloga, javi potreba za promenom dužine brojača. Korišćenje binarne konstante fiksne dužine se lako može izbeći upotrebnom agregacije, tj. konstrukcije *others* (v. 3.3.4):

```
q <= (OTHERS => '0');
```

Kao što znamo, konstrukcija (*others => '0'*), dodeljuje vrednost '0' **svim** elementima vektora čije je ime navedeno s leve strane znaka dodele. Pošto prethodna naredba ne sadrži informaciju o dužini vektora, ona će ostati nepromenjena i nakon eventualne promene dužine signala *q*. Tipični primeri korišćenja konstrukcije *others* su još i oni kad vektoru treba dodeliti vrednost ″sve jedinice″ ili kad bit najmanje težine vektora treba postaviti na '1', a sve ostale bitove na '0':

```
q <= (OTHERS => '1');
q <= (0 => '1', OTHERS => '0');
```

S obzirom na to što agregacija, u obliku koji je predstavljen u prethodnim primerima, ne sadrži informaciju o dužini, ona se može koristi samo za dodelu vrednosti vektoru poznate dužine, ali se ne može koristiti u aritmetičkim, logičkim ili relacionim izrazima. Na primer, sledeći kôd, koji je napisan s namerom da se ispita da li je vektor *a* jednak nuli, nije ispravan:

```
SIGNAL a : STD_LOGIC_VECTOR(N-1 DOWNTO 0);
. . .
y <= '1' WHEN (a = (OTHERS => '0')) ELSE . . .
```

Problem se može korigovati ako se u agregaciji umesto *others* koristi atribut *'range*:

```
y <= '1' WHEN (a = (a'RANGE => '0')) ELSE . . .
```

Agregacija (*a'range => '0'*) kreira bit-vektor ispunjen nulama koji je istog formata (opsega indeksa) kao vektor *a*. Još jedno rešenje za ovu situaciju jeste da se deklariše konstanta ″sve nule″ odgovarajuće dužine, koja bi se potom koristila u ispitivanjima:

```
CONSTANT zero : STD_LOGIC_VECTOR(N-1 DOWNTO 0) := (OTHERS => '0');
SIGNAL a : STD_LOGIC_VECTOR(N-1 DOWNTO 0);
. . .
y <= '1' WHEN (a = zero) ELSE . . .
```

Pr. 8-7 Korišćenje *integer*-a i funkcija za konverziju

Paket *numeric_std*, u kome su definisani tipovi podataka *unsigned* i *signed*, takođe sadrži i definicije funkcija za preklapanje aritmetičkih i relacionih operatora, koje omogućavaju da se u istom aritmetičkom ili relacionom izraz kombinuju podaci tipa *unsigned*/*signed* i *integer*, odnosno *natural* (v. 3.3.5). Na primer, pretpostavimo da je signal *a* iz sledeće *when* naredbe tipa *unsigned*:

```
y <= '1' WHEN (a = "00000101") ELSE . . .
```

Naredba ispituje da li je *a* jednako 5, pri čemu je konstanta 5 predstavljana u obliku bit-vektora. Međutim, umesto u obliku bit-vektora, konstanta se može izraziti i u obliku celog broja:

```
y <= '1' WHEN (a = 5) ELSE . . .
```

Efekat je isti jer će konstanta 5, zahvaljujući preklopljenom operatoru jednakosti iz paketa *numeric_std*, biti automatski konvertovana u bit-vektor dužine jednake dužini vektora *a*. Korišćenje celobrojne konstante umesto bit-vektora nije samo konciznije već i opštije rešenje jer ne zahteva modifikaciju kôda nakon promene dužine vektora *a*. Međutim, u slučajevim kada konstantu tipa *integer* želimo da dodelimo objektu tipa *unsigned/signed*, konverzija je neophodna:

```
y <= TO_UNSIGNED(5, N);
```

Prvi parametar funkcije *to_unsigned* je vrednost koja se konvertuje, a drugi broj bita koji će biti korišćen za predstavljanje ove vrednosti u binarnom obliku. Ako pak želimo da vektoru tipa *std_logic_vector* dodelimo *integer* konstantu, biće neophodna dupla konverzija:

```
y <= STD_LOGIC_VECTOR(TO_UNSIGNED(5, N));
```

Pr. 8-8 Referenciranje delova i elemenata vektora korišćenjem alijasa i atributa

U VHDL kôdu, često se javlja potreba za referenciranjem pojedinačnih elemenata vektora ili delova vektora. Tipičan primer je obraćanje bitu najveće (MSB) ili bitu najmanje težine (LSB) nekog vektora. Pretpostavimo da je *a* signal tipa *signed*(7 *downto* 0). Pošto se radi o tipu za označene binarne brojeve, MSB ovog vektora je ujedno i njegov bit znaka. Neka je za bit znaka uveden tzv. *alijas*:

```
ALIAS sign : STD_LOGIC IS a(7);
```

Konstrukcija *alias* definiše alternativno (obično skraćeno) ime za neki objekat ili deo objekta. U konkretnom primeru, *sign* je drugo ime za bit znaka vektora *a*. Napomenimo da alijas nije novi signal, već samo novo ime za isti signal i uvodi se radi kreiranja čitljivijeg kôda.

U prethodnoj deklaraciji, za referenciranje MSB-a, možemo koristiti atribut umesto konstantnog indeksa:

```
ALIAS sign : STD_LOGIC IS a(a'LEFT);
```

U slučaju da je signal *a* već parametrizovan kao *signed*(*N-1 downto 0*), isti efekat imaće i sledeća deklaracija:

```
ALIAS sign : STD_LOGIC IS a(N-1);
```

Slično, umesto da rotaciju za jednu poziciju udesno izrazimo na sledeći način:

```
q <= q(0) & q(7 DOWNTO 0);
```

to možemo učiniti i na opštiji način:

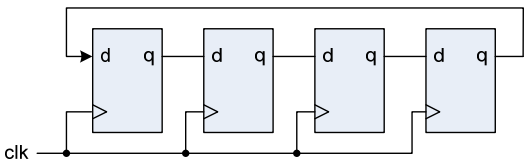
```
q <= q(q'RIGHT) & q(q'LEFT DOWNTO q'RIGHT+1);
```

Prethodna naredba će biti korektna ukoliko je dužina signala *q* veća od dva i njen opseg indeksa je opadajući (*downto*). Ako je signal *q* već parametrizovan, npr. kao *std_logic_vector*(*N*-1 *downto* 0), operacija rotacije se može izraziti na sledeći, očigledniji način:

```
q <= q(0) & q(N-1 DOWNTO 0);
```

Pr. 8-9 Kružni brojač

Kružni brojač je registarska komponenta koja se konstruiše povezivanje sirijskog izlaza na serijski ulaz pomeračkog registra. Ovaj princip je ilustrovan na Sl. 8-1. U normalnom režimu rada, kružni brojač sadrži samo jedno 1 koje se pod dejstvom taktnog signala pomera duž registra sve do poslednjeg flip-flopa, a onda se vraća na početak. Dakle, regularna stanja 4-bitnog kružnog brojača su: ″0001″, ″1000″, ″0100″ i ″0010″.



Sl. 8-1 Kružni brojač – princip.

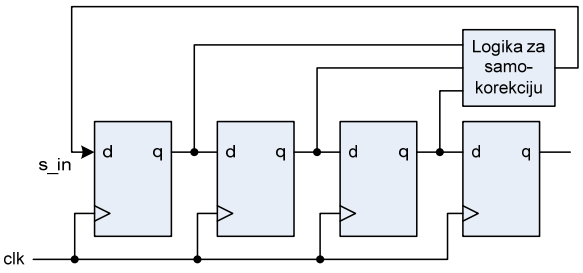
Postoje dva metoda za realizaciju kružnog brojača. Prvi podrazumeva da se prilikom inicijalizacije u registar paralelno upiše jedno od regularnih stanja. Za ovu namenu se može koristiti reset signal koji će postaviti registar ne u stanje ″0000″, kako je to uobičajeno, već npr. u stanje ″0001″. Nakon što se reset signal deaktivira, registar ulazi u normalan, ciklični režim rada.

Ispod je dat parametrizovan VHDL opis kružnog brojača koji koristi reset signal za inicijalizaciju. Uveden je generički parametar *N* koji definiše dužinu registra (linija 6). Parametar *N* se najpre koristi za dimenzionisanje izlaznog porta i internih signala (linije 7, 12 i 13), a zatim i u naredbi kojom se opisuje rotacija (linija 25). Treba obratiti pažnju i na liniju 19, gde se vrši inicijalizacija registra naredbom dodele u kojoj je početno stanje registra, ″00...01″, predstavljeno u vidu agregacije.

```
1 ------------------------------------------------------------------
2 LIBRARY IEEE;
3 USE IEEE.STD_LOGIC_1164.ALL;
4 ------------------------------------------------------------------
5 ENTITY para_ring_counter IS
6   GENERIC(N : NATURAL);
7   PORT (q : OUT STD_LOGIC_VECTOR(N-1 DOWNTO 0);
8         clk, rst : IN STD_LOGIC);
9 END para_ring_counter;
10 - Varijata sa reset signalom za inicijalizaciju ------------------
11 ARCHITECTURE rst_arch OF para_ring_counter IS
12   SIGNAL r_reg : STD_LOGIC_VECTOR(N-1 DOWNTO 0);
13   SIGNAL r_next : STD_LOGIC_VECTOR(N-1 DOWNTO 0);
14 BEGIN
15 -- registar ----------------------------------------------------
16   PROCESS(clk, rst)
17   BEGIN
18      IF(rst = '1') THEN
19        r_reg <= (0=>'1', OTHERS => '0');
20     ELSIF(clk'EVENT AND clk='1') THEN
21        r_reg <= r_next;
22     END IF;
23   END PROCESS;
24 -- logika sledeceg stanja  ---------------------------------------
25   r_next <= r_reg(0) & r_reg(N-1 DOWNTO 1);
26 -- logika izlaza ------------------------------------------------
27   q <= r_reg;
28 END rst_arch;
29 ------------------------------------------------------------------
```

Alternativan način za realizaciju kružnog brojača zasnovan je na ugradnji u registar tzv. logike za samo-korekciju (Sl. 8-2). U ovoj realizaciji, polazi se od zapažanja da u registar serijski treba upisati 1 samo ako su svi njegovi bitovi osim krajnjeg desnog jednaki nuli. Ili, drugim rečima, ako je na bilo kojoj poziciji registra izuzev krajnje desne prisutno 1, u registar treba serijski upisati 0. Dakle, u konfiguraciji sa Sl. 8-2, logika za samo-korekciju realizuje NILI funkciju. Treba, takođe, zapaziti da u ovoj realizaciji kružni registar ispravno radi čak i ako svoj rad započinje iz nekog neregularnog stanja (sve nule ili više od jedne jedinice). Na primer, ako je početno stanje brojača ″1011″, logika za samo-korekciju će serijskim upisivanjem nula postepeno istiskivati iz registar neregularnu sekvencu, sve dok se na prvih *N*-1 pozicija ne nađu sve nule. Nakon toga, registar ulazi u normalan režim

rada. Takođe, samokorigujući kružni brojač je u stanju da se oporavi od greške koja može nastati neželjenom promenom stanja nekog flip-flopa pod dejstvom električnih smetnji.



Sl. 8-2 Samokorigujući kružni brojač.

U VHDL opisu arhitekture samokorigujućeg kružnog brojača koji sledi uveden je alijas *r_high* koji predstavlja *N*-1 krajnjih levih bitova registra (linija 6-7). Funkcija logike za samo-korekciju opisana je naredbom *when* u kojoj se za ispitivanje da li je *r_high* jednak nuli koristi agregacija (*r_high'range => '0'*) – linije 19-20.

```
1  - Varijanta sa samo-korekcijom ---------------------------------
2  ARCHITECTURE self_corect_arch OF para_ring_counter IS
3   SIGNAL r_reg : STD_LOGIC_VECTOR(N-1 DOWNTO 0);
4   SIGNAL r_next : STD_LOGIC_VECTOR(N-1 DOWNTO 0);
5   SIGNAL s_in : STD_LOGIC;
6   ALIAS r_high : STD_LOGIC_VECTOR(N-2 DOWNTO 0) IS
7                    r_reg(N-1 DOWNTO 1);
8  BEGIN
9  -- registar ------------------------------------------------------
10  PROCESS(clk, rst)
11  BEGIN
12    IF(rst = '1') THEN
13       r_reg <= (OTHERS => '0');
14    ELSIF(clk'EVENT AND clk='1') THEN
15       r_reg <= r_next;
16    END IF;
17  END PROCESS;
18 -- logika sledeceg stanja  ---------------------------------------
19  s_in <= '1' WHEN r_high = (r_high'range => '0') ELSE
20          '0';
21   r_next <= s_in & r_high;
22 -- logika izlaza -------------------------------------------------
23  q <= r_reg;
24 END self_corect_arch;
```

8.4. GENERATE

Naredba *generate* je konkurentna naredba koja se koristi za kontrolisano ″generisanje″ konkurentnog kôda. Može se smatrati konkurentnim ekvivalentom sekvencijalnih naredbi *for loop* i *if*. Postoje dva oblika ove naredbe: *for generate* i *if generate*. Naredba *for generate* omogućava umnožavanje istog skupa konkurentnih naredbi (tj. odgovara sekvencijalnoj naredbi *for loop*), dok naredba *if generate* omogućava uslovno kreiranje obuhvaćenog konkurentnog kôda (tj. odgovara sekvencijalnoj naredbi *if*).

8.4.1. FOR GENERATE

Mnoga digitalna kola se mogu realizovati u vidu regularne strukture identičnih gradivnih blokova. Gradivni blokovi mogu biti povezani u obliku jednodimenzionog lanca, u strukturu oblika stabla ili u vidu dvodimenzionog polja. Tipični primeri ovih tzv. *iterativnih kola* su: sabirač sa rednim prenosom, pomerački registar, paralelni množač itd. Iterativna kola se lako mogu proširiti prostim povećanjem broja iteracija i stoga predstavljaju idealne kandidate za parametrizovano projektovanje. Naredba *for generate* nalazi glavnu primenu upravo za opisivanje ovakvih struktura.

Sintaksa naredbe *for generate* je sledećeg oblika:

```
labela: FOR indeks IN opseg GENERATE
 (konkurentne naredbe;)
END GENERATE;
```

Naredba *for generate* ponavlja obuhvaćene konkurentne naredbe zadati broj puta. Član *opseg* definiše opseg indeksa petlje, odnosno njegovu početnu i krajnju vrednost. Opseg mora biti statički, što znači da konkretne vrednosti granica opsega moraju biti poznate u momentu sinteze kôda. Za specifikaciju opsega po pravilu se koristi neki dimenzioni parametar sistema. Indeks igra ulogu brojača petlja; kreće od početne vrednosti zadatog opsega, a nakon svake iteracije uzima sledeću vrednosti iz opsega. Poslednja iteracija petlje je ona u kojoj je vrednost indeksa jednaka krajnjoj vrednosti opsega. Indeks po automatizmu preuzima tipa opsega i ne deklariše se van *for generate* naredbe. Labela je obavezna i jedinstvena u okviru arhitekture.

Pr. 8-10 Ilustracija *for generate* naredbe

Razmotrimo sledeći segment VHDL kôda:

```
SIGNAL x: BIT_VECTOR(7 DOWNTO 0);
SIGNAL y: BIT_VECTOR(15 DOWNTO 0);
SIGNAL z: BIT_VECTOR(7 DOWNTO 0);
. . .
G1: FOR i IN x'RANGE GENERATE
   z(i) <= x(i) AND y(i+8);
END GENERATE;
```

Naredba *for generate* iz prethodnog kôdu kreira po jednu instancu obuhvaćene konkurentne naredbe dodele za svaku vrednost indeksa *i* iz opsega 7 do 0. Možemo smatrati da ova naredba *for generate* predstavlja koncizan zapis sledećeg kôda:

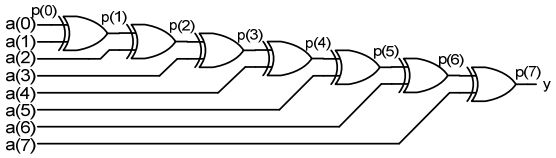
```
z(7) <= x(7) AND y(7+8);
z(6) <= x(6) AND y(6+8);
z(5) <= x(5) AND y(5+8);
z(4) <= x(4) AND y(4+8);
z(3) <= x(3) AND y(3+8);
z(2) <= x(2) AND y(2+8);
z(1) <= x(1) AND y(1+8);
z(0) <= x(0) AND y(0+8);
```

Konkurentne naredbe koje su obuhvaćene *for generate* naredbom, u suštini, opisuju jedan *stepen* nekog iterativnog kola. Pored toga što opisuju bazični gradivni blok, ove naredbe opisuju i način kako su susedni blokovi međusobno povezani u iterativnu strukturu, tj.,

drugim rečima, opisuju šablon za povezivanje gradivnih blokova. Za povezivanje stepena koriste interni signali, koji su najčešće deklarisani kao jednodimenzioni ili dvodimenzioni vektori. Šablon za povezivanje definiše pravilo po kome se u svakoj iteraciji formiraju indeksi za odabir internih signala koji će se koristiti za povezivanje.

Suština projektovanja iterativnih kola je u identifikaciji bazičnih gradivnih blokova i šablona za povezivanje susednih stepena. Ovo ne mora uvek biti jednostavan zadatak, te je stoga preporučljivo da se pre pisanja iterativnog kôda skicira dijagram željene strukture sa označenim vezama između stepena, kako bi se što lakše uočile pravilnosti u povezivanju, a onda i izvela opšta zakonitost.

Pr. 8-11 Generator bita parnosti



Sl. 8-3 Realizacija generatora bita parnosti pomoću XOR kola.

Na Sl. 8-3 je prikazana realizacija 8-bitnog generator bita parnosti u vidu kaskadne veze isključivo ILI (XOR) kola. Kao što se može zapaziti, mreža poseduje regularnu, iterativnu strukturu, koja se nadovezivanjem XOR kola može lako proširiti do proizvoljnog broja ulaza. Gradivni blokovi (stepeni) ove strukture su XOR kola. Stepene možemo numerisati s leva na desno, tako da krajnji levi stepen bude nulti, a krajnji desni sedmi, odnosno, u opštem slučaju, (*N*-1)-vi. Za povezivanje susednih stepena koristi se vektor *p*, tako što signal *p*(*i*) povezuje izlaz stepena *i*-1 i jedan od dva ulaz stepena *i*. Drugi ulaz *i*-tog stepena povezan je sa *i*-tim ulazom generator bita parnosti. Na osnovu dijagrama sa Sl. 8-3 lako se uočava zavisnost između indeksa izlaznog i dva ulazna signala svakog XOR kola. Konkretno, za *i*-ti stepen važi:

```
p(i) <= a(i) XOR p(i-1);
```

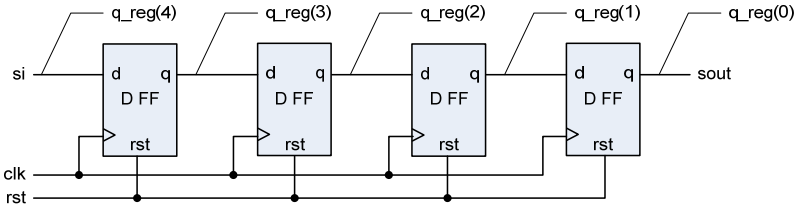
Ispod je dat parametrizovan opis arhitekture generatora parnosti u kome je iskorišćena *for generate* naredbe. Petlja *for generate* naredbe se ponavlja *N*-1 puta, kreirajući tako *N*-1 XOR kola.

```
1 ------------------------------------------------------------------
2  ARCHITECTURE generate_arch OF pargen IS
3    SIGNAL p : STD_LOGIC_VECTOR(N-1 DOWNTO 0);
4  BEGIN
5    p(0) <= a(0);
6    FOR I IN 1 TO N-1 GENERATE
7      p(I) <= a(I) XOR p(I-1);
8    END GENERATE;
9    y <= p(N-1);
10 END generate_arch;
11------------------------------------------------------------------
```

Sprega graničnih stepena iterativnih struktura sa spoljašnjim ulaznim, odnosno izlaznim signalima po pravilu se tretira kao specijalan slučaj u odnosu na opšte pravilo za povezivanje. U konkretnom primeru, ove specifične situacije su obrađene dvema naredbama (linije 5 i 9). Naredba dodele iz linije 5 ″preklapa″ nulti bit internog vektora *p* s ulazom *a*(0), dok naredba iz linije 9, na izlaz *y* prenosi vrednost MSB-a vektora *p*, koji ujedno predstavlja i izlaz iz poslednjeg XOR kola u nizu.

Pr. 8-12 Parametrizovan opis pomeračkog registra

U ovom primeru, naredba *for generate* je iskorišćena za kreiranje parametrizovanog opisa pomeračkog registra. Registar se sastoji od *n* redno povezanih D flip-flopova sa zajedničkim signalom takta i signalom za resetovanje (Sl. 8-4). Serijski ulaz registra, *si*, je ulaz u prvi, a serijski izlaz iz registra, *sout*, je izlaz iz poslednjeg flip-flopa u nizu. Svaki D flip-flop je jedan stepen u ovoj iterativnoj strukturi. Šema povezivanja je jednostavna, jer se izlaz iz svakog flip-flopa, prosto, povezuje sa ulazom u sledeći.



Sl. 8-4 Pomerački registar

U VHDL opisu koji sledi, naredba *for generate* se koristi za kreiranje *N* procesa, od kojih svaki opisuje jedan flip-flop sa asinhronim resetom. Povezivanje flip-flopova je ostvareno naredbom dodele iz linije 23, u kojoj se tekućem bitu vektora *q_reg* dodeljuje vrednost prethodnog bita. Uočimo da je signal *q_reg* proširen jednim dodatnim bitom, *q_reg*(*n*), koji se koristi za preklapanje sa ulaznim signalom *si*. Međutim, budući da se indeks petlje kreće od *n*-1 do 0, signalu *q_reg*(*n*) se ne dodeljuje vrednost ni u jednom procesu, pa je zbog toga broj sintetizovanih flip-flopova jednak *n*, a ne *n*+1.

```
1  ----------------------------------------------------------------------
2  LIBRARY IEEE;
3  USE IEEE.STD_LOGIC_1164.ALL;
4  ----------------------------------------------------------------------
5  ENTITY pomreg IS
6    GENERIC(N : NATURAL);
7    PORT(si : IN STD_LOGIC;
8         sout : OUT STD_LOGIC;
9         clk, rst : IN STD_LOGIC);
10 END pomreg;
11 ----------------------------------------------------------------------
12 ARCHITECTURE gen_proc_arch OF pomreg IS
13   SIGNAL q_reg : STD_LOGIC_VECTOR(N DOWNTO 0);
14 BEGIN
15   q_reg(N) <= si;
16   dff_gen: FOR I IN (N-1) DOWNTO 0 GENERATE
17   -- D FF
18      PROCESS(clk, rst)
19      BEGIN
20         IF(rst = '1') THEN
21            q_reg(I) <= '0';
22         ELSIF(clk'EVENT AND clk = '1') THEN
23            q_reg(I) <= q_reg(I-1);
24         END IF;
25      END PROCESS;
26   END GENERATE;
```

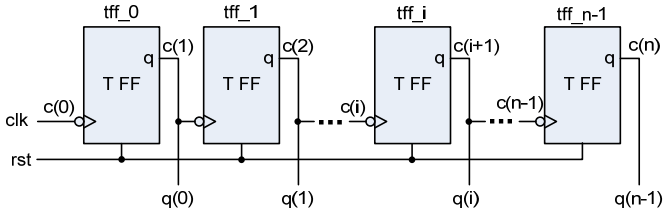
```
27   -- izlaz
28   sout <= q_reg(0);
29 END gen_proc_arch;
30 ---------------------------------------------------------------------
```

U nastavku je predstavljen još jedan parametrizovan opis arhitekture pomeračkog registra u kome se za kreiranje bazičnih blokova umesto procesa koristi komponenta DFF. U osnovi, ovo rešenje prestavlja strukturni opis blok dijagrama sa Sl. 8-4.

```
1  ---------------------------------------------------------------------
2  ARCHITECTURE gen_comp_arch OF pomreg IS
3    COMPONENT DFF
4      PORT(d : IN STD_LOGIC;
5           q : OUT STD_LOGIC;
6           clk, rst : IN STD_LOGIC);
7    END COMPONENT;
8    SIGNAL q_reg : STD_LOGIC_VECTOR(N DOWNTO 0);
9  BEGIN
10   q_reg(N) <= si;
11   dff_gen: FOR I IN (N-1) DOWNTO 0 GENERATE
12     dff_I: DFF
13     PORT MAP(d=>q_reg(I+1), q=>q_reg(I), clk=>clk, rst=>rst);
14   END GENERATE;
15   -- izlaz
16   sout <= q_reg(0);
17 END gen_proc_arch;
```

Pr. 8-13 Asinhroni brojač

Na Sl. 8-5 je prikazana struktura *n*-bitnog asinhronog binarnog brojača. Brojač se sastoji od *n* redno povezanih T flip-flopova. Ulazni taktni signal, *clk*, pobuđuje samo prvi flip-flop (*tff_0*), dok se takt za svaki sledeći flip-flop uzima sa izlaza prethodnog flip-flopa. Budući da se T flip-flopovi okidaju opadajućom ivicom takta (naznačeno kružićem na ulazu za takt), *i*-ti T flip-flop menja svoje stanje samo onda kada se stanja svih njemu prethodnih flip-flopova promene sa ´1´ na ´0´, što odgovara binarnoj sekvenci brojanja, sa izlazom q_{n-1} najveće i izlazom q_0 najmanje težine.



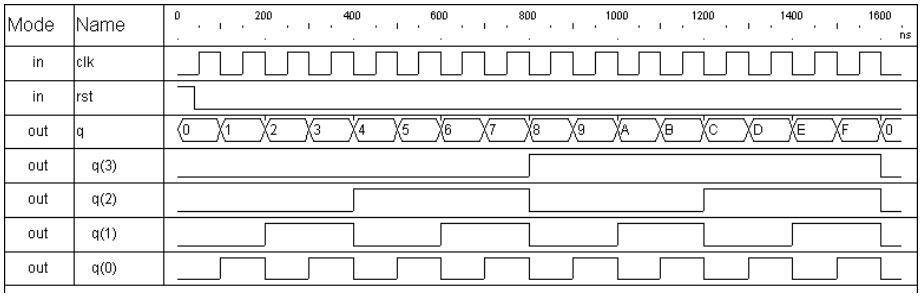
Sl. 8-5 *n*-bitni asinhroni brojač.

VHDL kôd koji sledi smešten je u dve datoteke: *tff.vhd*, s kôdom za T flip-flopa (TFF) i *asyn_counter.vhd*, s kôdom parametrizovanog opisa *n*-bitnog asinhronog brojača u kome se TFF koristi kao komponenta. U arhitekturi T flip-flopa, za čuvanje tekućeg stanja koristi se interni signal *ff*, koji se postavlja na ´0´ ako je reset signal aktivan (linija 16), odnosno komplementira (linija 18), ako je detektovana opadajuća ivica taktnog signala *clk* (linija 17). Stanje flip-flopa se prenosi na izlaz *q* u liniji 21.

U strukturnom VHDL opisu asinhronog brojača (kôd iz datoteke *asyn_counter.vhd*) naredba *for generate* se koristi za instanciranje i povezivanje *n* T flip-flopova. Za prenos takta između flip-flopova uveden je (*N*+1)-bitni vektor *c* koji se koristi na način kao na Sl. 8-5. Signal *c*(0) preklapa ulazni taktni signal (linija 18). *i*-ti T flip-flop se taktuje signalom *c*(*i*), a njegov izlaz *q* pobuđuje signal *c*(*i*+1) (linija 20). Bitovi 1, ..., *N* vektora *c* su ujedno i izlazi brojača (linija 22). Vremenski dijagram dobijen simulacijom datog VHDL kôda, za *N*=4, prikazan je na Sl. 8-6.

```
1 ------ tff.vhd ----------------------------------------------------
2 LIBRARY IEEE;
3 USE IEEE.STD_LOGIC_1164.ALL;
4 ---------------------------------------------------------------------
5 ENTITY tff IS
6    PORT (clk, rst: IN STD_LOGIC;
7          q: OUT STD_LOGIC);
8 END tff;
9 ---------------------------------------------------------------------
10 ARCHITECTURE behavior OF tff IS
11   SIGNAL ff : STD_LOGIC;
12 BEGIN
13   PROCESS (clk, rst)
14   BEGIN
15      IF(rst='1') THEN
16         ff <= '0';
17      ELSIF (clk'EVENT AND clk='0') THEN
18         ff <= not ff;
19      END IF;
20   END PROCESS;
21   q <= ff;
22 END behavior;
```

```
1 ----- asyn_counter.vhd ------------------------------------------
2 LIBRARY IEEE;
3 USE IEEE.STD_LOGIC_1164.ALL;
4 ---------------------------------------------------------------------
5 ENTITY asyn_count IS
6    GENERIC(N : INTEGER := 4);
7    PORT (clk, rst: IN STD_LOGIC;
8          q: OUT STD_LOGIC_VECTOR(N-1 DOWNTO 0));
9 END asyn_count;
10 ---------------------------------------------------------------------
11 ARCHITECTURE structural OF asyn_count IS
12   COMPONENT tff IS
13   PORT (clk, rst: IN STD_LOGIC;
14         q: OUT STD_LOGIC);
15   END COMPONENT;
16   SIGNAL c : STD_LOGIC_VECTOR(N DOWNTO 0);
17 BEGIN
18   c(0) <= clk;
19   G1:FOR i IN 0 TO N-1 GENERATE
20      tffx : tff PORT MAP (clk=>c(i),rst=>rst,q=>c(i+1));
21   END GENERATE;
22   q <= c(N DOWNTO 1);
23 END structural;
```



Sl. 8-6 Rezultati simulacije VHDL opisa asinhronog brojača.

Pr. 8-14 Generator bita parnosti – hijerarhijska struktura

Generator bita parnosti je razmatran u više navrata u prethodnim primerima. Međutim, uvek do sada ovo kola je predstavljano u vidu redne veze XOR kola (kao na Sl. 8-3). Problem s rednom ralizacijom je povećano propagaciono kašnjenje, jer kritična putanja prolazi kroz svih N-1 XOR kola. Efikasnije rešenje, u pogledu propagacionog kašnjenja, je ono u kojem su XOR kola umesto redno, raspoređena hijerahijski i povezana u strukturu oblika stabla (kao na Sl. 8-7). U poređenju sa rednom realizacijom, broj XOR kola je ostao isti, tj. N-1, ali je zato kritična putanja skraćena sa N-1 na $\lceil \log_2 N \rceil$ XOR kola.

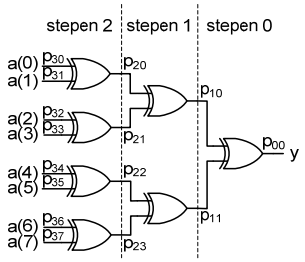
Osnovna razlika između redne i hijerarhijske realizacije generatora bita parnosti je u redosledu izračunavanja. U VHDL-u, redosled izračunavanja se može lako nametnuti korišćenjem zagrada. Na primer, jednačina 4-bitnog generatora parnosti koja odgovara rednom načinu izračunavanja sledećeg je oblika:

```
y <= a(3) XOR (a(2) XOR (a(1) XOR a(0))
```

dok za hijerarhijski slučaj postaje:

```
y <= (a(3) XOR a(2)) XOR (a(1) XOR a(0))
```

Međutim, ovaj pristup se ne može primeniti za kreiranje parametrizovanog opisa, zato što broj ulaza nije unapred poznat. U parametrizovanom opisu, neophodno je eksplicitno definisati šablon povezivanja. U tom cilju, strukturu sa Sl. 8-7, koja je u osnovi dvodimenziona, treba najpre podeliti na stepene, a zatim XOR kola numerisati dvodimenzionim indeksima, tako da prvi indeks ukazuje na stepen (tj. kolonu), a drugi na poziciju kola u stepenu (tj. vrstu). Tako se XOR kolo s indeksom (s, r) nalazi u r-toj vrsti s-tog stepenu. Uočimo da je na Sl. 8-7 izlazni signal XOR kola (s, r) označen sa $p_{s,r}$. Radi uniformnosti imenovanja signala, ulazni signali, $a(0)$, …, $a(7)$, i izlazni signal (y) generatora parnosti označeni su po istoj konvenciji.



Sl. 8-7 Hijerarhijski generator parnosti.

Ključ za kreiranje parametrizovanog opisa iterativne strukture je u pronalaženju zavisnosti koja postoji između indeksa signala susednih stepena. Razmotrimo XOR kolo (s, r). Dva ulaza ovog kola su u vezi sa $2r$-tim i $(2r+1)$-tim kolom iz stepena $s+1$. Faktor 2 u indeksu vrste posledica je činjenice da je broj XOR kola u svakom sledećem stepenu dvaput manji nego u prethodnom. Dakle, ulazno-izlazna zavisnost XOR kola (s, r) može se izraziti jednačinom:

$$p_{s,r} = p_{s+1,2r} \oplus p_{s+1,2r+1}$$

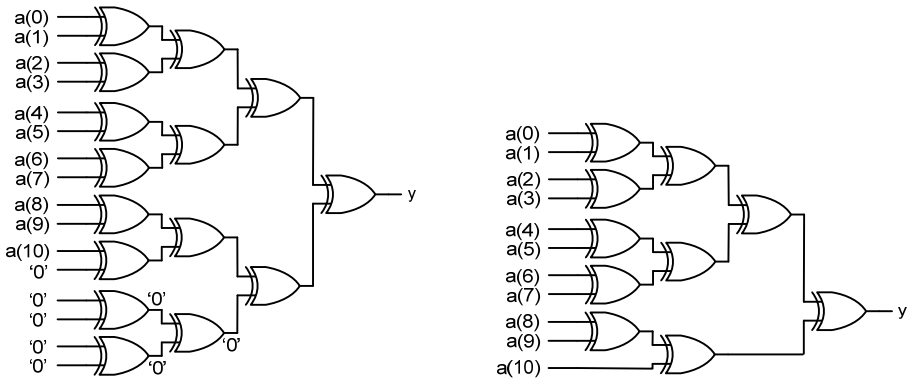
Izvedena jednakosti predstavljaće osnovu za kreiranje parametrizovanog VHDL opisa hijerarhijskog generatora parnosti. Dvodimenzionalnost strukture zahteva uvođenje dvodimenzionog tipa podataka za signal p i korišćenje dve ugnježdene *for generate* naredbe, gde će spoljna prolaziti kroz stepene, a unutrašnja kroz vrste. Napomenimo da broj stepena n-to ulaznog generatora parnosti iznosi $\lceil \log_2 n \rceil$, sa 2^s XOR kola u s-tom stepenu.

Prilikom pisanja VHDL kôda koji sledi pošlo se od pretpostavke da generički parametar n (broj ulaza) može uzeti samo vrednosti koje su jednake stepenu dvojke. U deklarativnom delu arhitekture sadržana je deklaracija funkcije *log2c* (linije 13-22 – v. Pr. 7-11), koja se u liniji 24 koristi za izračunavanje vrednosti konstante *stage* – broj stepena u strukturi sa n ulaza. Napomenimo da bi pravo mesto za ovu funkciju, zbog njene opštosti, bilo u paketu. U liniji 25 definisan je tip podataka *d2sig* koji predstavlja 2D polje dimenzija *stage* x n. Signal p tipa *d2sig* deklarisan je u liniji 26. Telo arhitekture sadrži tri *for generate* naredbe. Prva se koristi za preklapanje ulaznih signala, a_0, ..., a_7 s internim signalima $p_{\log2c(n)-1,0},\ldots,$ $p_{\log2c(n),n-1}$ (linije 29-31). Hijerarhijsko polje XOR kola generiše se pomoću dve ugnježdene *for generate* naredbe (linije 33-37). Konačno, u liniji 40, rezultat izračunavanja (vrednost signala $p_{0,0}$) prosleđuje se na izlazni port y.

```
1  ----------------------------------------------------------------
2  LIBRARY IEEE;
3  USE IEEE.STD_LOGIC_1164.ALL;
4  ----------------------------------------------------------------
5  ENTITY pargen IS
6    GENERIC(N: INTEGER);
7    PORT (a : IN STD_LOGIC_VECTOR(N-1 DOWNTO 0);
8          y : OUT STD_LOGIC);
9  END pargen;
10 ----------------------------------------------------------------
11 ARCHITECTURE tree_arch OF pargen IS
12 -- f-ja log2c(n) ------------------------------------------------
13   FUNCTION log2c(n : INTEGER) RETURN INTEGER IS
14     VARIABLE m, p : INTEGER;
15   BEGIN
16     m := 0; p := 1;
17     WHILE(p < n) LOOP
18       m := m + 1;
19       p := p * 2;
20     END LOOP;
21     RETURN m;
22   END log2c;
23 ----------------------------------------------------------------
24   CONSTANT STAGE : NATURAL := log2c(N);
25   TYPE d2sig IS ARRAY(STAGE DOWNTO 0,N-1 DOWNTO 0) OF STD_LOGIC;
```

```
26   SIGNAL p : d2signal;
27 BEGIN
28 -- preklapanje ulaznih signala ---------------------------------
29   in_gen: FOR I IN 0 TO (N-1) GENERATE
30      p(STAGE,I) <= a(I);
31   END GENERATE;
32 -- generisanje XOR polja ---------------------------------------
33   st_gen: FOR s IN (STAGE-1) DOWNTO 0 GENERATE
34      xor_gen: FOR r IN 0 TO (2**s - 1) GENERATE
35         p(s,r) <= p(s+1, 2*r) XOR p(s+1, 2*r+1);
36      END GENERATE;
37   END GENERATE;
38 - izlaz ------------------------------------------------------------
39   y <= p(0,0);
40 END tree_arch;
41-------------------------------------------------------------------
```

Kao što je već rečeno, prethodni VHDL opis je korektan samo ako je broj ulaza generatora parnosti jednak stepenu dvojke. Ako to nije slučaj, određeni broj ulaza ulaznog stepena ostaće nepovezan. Na primer, za $n = 11$ generisaće se *stage* = 4 stepena, a ulazni stepen imaće $2^{stage-1} = 8$ XOR kola sa ukupno 16 ulaza, od kojih će 11 biti povezani s ulaznim signalima (prva *for generate* petlja, linije 29-31), dok će preostalih 5 ostati nepovezani. Ovaj problem se može rešiti ako se neiskorišćeni ulazi postave na konstantnu vrednost ´0´. Ova modifikacija neće poremetiti funkciju kola budući da važi $x = x \oplus 0$. Takođe, prilikom sinteze, sva redundantna XOR kola biće automatski eliminisana. Ovaj princip je za slučaj $n = 11$ ilustrovan je na Sl. 8-8. Sl. 8-8(a) prikazuje hijerarhijsku XOR strukturu sa fiksiranim neiskorišćenim ulazima. Zapazimo da zbog konstantnih vrednosti na svojim ulazima, čak pet XOR kola iz donjeg dela ove strukture gube svoju svrhu i da se mogu odstraniti iz mreže. Strukturu sintetizovanog generatora parnosti prikazana je na Sl. 8-8(b).



Sl. 8-8. Generator bita parnosti za proizvoljan broj ulaza: (a) princip; (b) sintetizovano kolo.

U nastavku sledi modifikovan VHDL opis arhitekture hijerarhijskog generatora parnosti za proizvoljan broj ulaza (deklaracija funkcije *log2c* je izostavljena). Modifikacija se sastoji u uvođenju još jedne naredbe *for generate* koja nakon povezivanja ulaza kola na ulazni stepen, nastavlja sa postavljanjem nula na preostale ulaze ulaznog stepena (linije 13-15). Uočimo da za slučajeve kad je n jednako stepenu dvojke, uvedena *for generate* naredba neće imati nikakvo dejstvo, jer će početni indeks opsega petlje biti veći od krajnjeg.

```
1  ------------------------------------------------------------------
2  ARCHITECTURE tree_arch_1 OF pargen IS
3    CONSTANT STAGE : NATURAL := log2c(N);
4    TYPE d2signal IS ARRAY(STAGE DOWNTO 0, 2**STAGE-1 DOWNTO 0)
5      OF STD_LOGIC;
6    SIGNAL p : d2signal;
7  BEGIN
8  -- preklapanje ulaznih signala ---------------------------------
9    in_gen: FOR I IN 0 TO (N-1) GENERATE
10     p(STAGE,I) <= a(i);
11   END GENERATE;
12 -- postavljanje nula -------------------------------------------
13   zero_gen: FOR I IN N TO (2**STAGE-1) GENERATE
14     p(STAGE,I) <= '0';
15   END GENERATE;
16 -- generisanje XOR polja ----------------------------------------
17   st_gen: FOR s IN (STAGE-1) DOWNTO 0 GENERATE
18     xor_gen: FOR r IN 0 TO (2**s - 1) GENERATE
19       p(s,r) <= p(s+1, 2*r) XOR p(s+1, 2*r+1);
20     END GENERATE;
21   END GENERATE;
22 -- izlaz -------------------------------------------------------
23   y <= p(0,0);
24 END tree_arch_1;
```

8.4.2. IF GENERATE

U parametrizovanom projektovanju, naredba *if generate* se prevashodno koristi za realizaciju funkcionalnih parametara. Pomoću ove naredbe moguće je uključiti ili isključiti pojedine delove kola iz konačne implementacije. Sintakse naredbe *if generate* je sledećeg oblika:

```
labela: IF uslov GENERATE
 (konkurentne naredbe;)
END GENERATE;
```

Član *uslov* je logički izraz koji vraća vrednost tipa *boolean*. Izvršenje konkurentnih naredbe koje su obuhvaćene *if generate* naredbom je omogućeno ako je uslov tačan, što znači da će deo kola koji one opisuju biti uključen u implementaciju. U suprotnom slučaju, ako je uslov netačan, izvršenje obuhvaćenih konkurentne naredbe je onemogućeno, što praktično znači da će odgovarajući deo kola biti izostavljen iz implementacije. Naredba *if generate* ne sadrži *else* granu. To znači da za uključenje u implementaciju jednog od dva alternativna kola treba napisati dve *if generate* naredbe s komplementarnim uslovima. Kad se naredba *if generate* koristi u kôdu za sintezu, njen uslov mora biti statički, tj. u fazi sinteze mora biti poznato da li je uslov tačan ili netačan. Uslov *if generate* naredbe se najčešće izražava preko generičkih parametara.

Pr. 8-15 Generator parnosti

Naredba *if generate* se često koristi u opisima iterativnih kola, kada unutar *for generate* naredbe treba izdvojiti ″neregularne″ stepene iz inače regularne strukture. Razmotrimo još jedanput realizaciju generatora parnosti iz Pr. 8-11. U strukturi sa Sl. 8-3, prvi i poslednji stepen se razlikuju od unutrašnjih stepena po tome što se oni povezuju sa ulaznim i

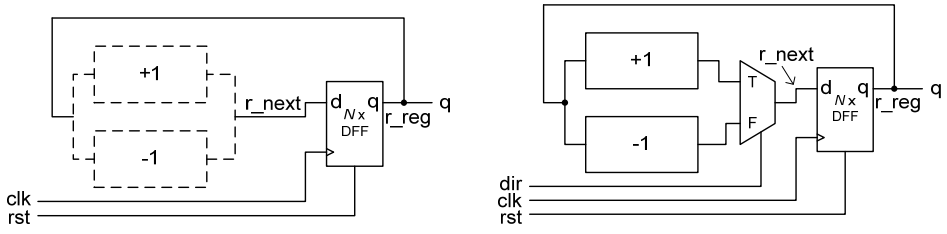
izlaznim portovima za koje se koristi drugačija konvencija imenovanja. U Pr. 8-11, ovaj problem je rešen pomoću dve naredbe dodele koje van *for generate* petlje preklapaju portove i unutrašnje signale za povezivanje:

```
p(0) <= a(0);
y <= p(N-1);
```

U VHDL opisu generatora parnosti koji sledi, kôd iz tela *for generate* petlje je naredbama *if generate* razdeljen na tri uzajamno isključive sekcije. Kao što se može zapaziti, u zavisnosti od vrednosti indeksa petlje, *i*, uvek će biti ispunjen uslov tačno jedne *if generate* naredbe. Za $i = 1$, kôd tela petlje se svodi na naredbu dodele iz linije 8 koja generiše krajnji levi stepen; za $i = n$-1, u telu petlje omogućena je samo naredba iz linije 16 koja generiše krajnji desni stepen, dok za sve ostale vrednosti brojača petlje, važi naredba dodele iz linije 12 koja generiše unutrašnje stepene generatora parnosti.

```
1  ------------------------------------------------------------------
2  ARCHITECTURE gen_if_arch OF pargen IS
3    SIGNAL p : STD_LOGIC_VECTOR(N-2 DOWNTO 1);
4  BEGIN
5    xor_gen: FOR I IN 1 TO (N-1) GENERATE
6       -- krajnji levi stepen
7       left_gen: IF I = 1 GENERATE
8          p(I) <= a(I) XOR a(0);
9       END GENERATE;
10      -- sredisnji stepeni
11      middle_gen: IF (I > 1) AND (I < (N-1) GENERATE
12         p(I) <= a(I) XOR p(I-1);
13      END GENERATE;
14      -- krajnji desni stepen
15      right_gen: IF I = (N-1) GENERATE
16         y <= a(I) XOR p(I-1);
17      END GENERATE;
18    END GENERATE;
19 END gen_if_arch;
```

Pr. 8-16 ″Up-or-Down″ brojač



Sl. 8-9 (a) ″Up-or-Down″ brojač; (b) ″Up-and-Down″ brojač.

Pod ″Up-or-Down″ brojačem podrazumevaćemo komponentu koja se može *instancirati* za rad u jednom od dva režima: brojanje naviše (*Up*) ili brojanje naniže (*Down*). Naglasimo da ovde nije reč o obostranom brojaču, tj. o brojaču s mogućnošću izbora smera brojanja (tzv. ″Up-and-Down″ brojač), već o parametrizovanom opisu brojača koji nakon sinteze radi na jedan ili drugi način. Konceptualni dijagrami ″Up-or-Down″ i ″Up-and-Down″ brojača

prikazani su na Sl. 8-9. Opcioni delovi ″Up-or-Down″ brojača prikazani su isprekidanim linijama (Sl. 8-9(a)). U zavisnosti od vrednosti funkcionalnog generičkog parametra, sintetizovaće se ili inkrementer (blok ″+1″) ili dekermenter (blok ″-1″). S druge strane, ″Up-and-Down″ brojač sadrži oba ova bloka, a poseban ulazni signal (*dir*) služi za izbor bloka čiji će izlaz biti prihvaćen kao novo stanje brojača, odnosno za izbor smera brojanja (Sl. 8-9(b)).

VHDL opis ″Up-or-Down″ brojača koji sledi sadrži dva generička parametra: jedan je dimenzioni, *n*, koji definiše dužinu brojača, dok je drugi funkcionalni, *up*. Za *up*=´1´, bira se brojač koji broji naviše, a za *up* ≠ ´1´ brojač koji broji naniže. Dve alternativne ″logike sledećeg stanja″ opisane su pomoću dve zasebne *if generate* naredbe. Pošto su uslovi ovih naredbi komplementarni (tj. uzajamno isključivi), konflikt na signalu *r_next* ne postoji.

```
1  ----------------------------------------------------------------
2  LIBRARY IEEE;
3  USE IEEE.STD_LOGIC_1164.ALL;
4  USE IEEE.NUMERIC_STD.ALL;
5  ----------------------------------------------------------------
6  ENTITY up_or_down_counter IS
7    GENERIC(N : NATURAL;
8            UP : NATURAL);
9    PORT (q : OUT STD_LOGIC_VECTOR(N-1 DOWNTO 0);
10         clk, rst : IN STD_LOGIC);
11 END up_or_down_counter;
12 ----------------------------------------------------------------
13 ARCHITECTURE arch OF up_or_down_counter IS
14   SIGNAL r_reg : UNSIGNED(N-1 DOWNTO 0);
15   SIGNAL r_next : UNSIGNED(N-1 DOWNTO 0);
16 BEGIN
17   -- registar ---------------------------------------------------
18   PROCESS(clk, rst)
19   BEGIN
20     IF(rst = '1') THEN
21        r_reg <= (OTHERS => '0');
22     ELSIF(clk'EVENT AND clk='1') THEN
23        r_reg <= r_next;
24     END IF;
25   END PROCESS;
26   -- logika sledeceg stanja - inkrementer ------------------------
27   inc_gen: IF UP = 1 GENERATE
28     r_next <= r_reg + 1;
29   END GENERATE;
30   -- logika sledeceg stanja - dekremeter ------------------------
31   dec_gen: IF UP /= 1 GENERATE
32     r_next <= r_reg - 1;
33   END GENERATE;
34   -- logika izlaza ----------------------------------------------
35   q <= STD_LOGIC_VECTOR(r_reg);
36 END arch;
```

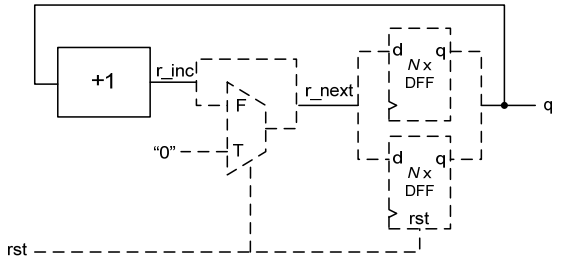
Poređenja radi, u nastavku je predstavljen VHDL opis ″Up-and-Down″ brojača.

```
1  ----------------------------------------------------------------
2  LIBRARY IEEE;
3  USE IEEE.STD_LOGIC_1164.ALL;
4  USE IEEE.NUMERIC_STD.ALL;
```

```
5  ----------------------------------------------------------------------
6  ENTITY up_and_down_counter IS
7   GENERIC(N : NATURAL)
8   PORT (q : OUT STD_LOGIC_VECTOR(N-1 DOWNTO 0);
9         clk, rst : IN STD_LOGIC);
10 END up_and_down_counter;
11 ----------------------------------------------------------------------
12 ARCHITECTURE arch OF up_and_down_counter IS
13   SIGNAL r_reg : UNSIGNED(N-1 DOWNTO 0);
14   SIGNAL r_next : UNSIGNED(N-1 DOWNTO 0);
15 BEGIN
16  -- registar ---------------------------------------------------
17  PROCESS(clk, rst)
18  BEGIN
19    IF(rst = '1') THEN
20       r_reg <= (OTHERS => '0');
21    ELSIF(clk'EVENT AND clk='1') THEN
22       r_reg <= r_next;
23    END IF;
24  END PROCESS;
25  -- logika sledeceg stanja - inkrementer/dekrementer -------------
26 r_next <= r_reg + 1 WHEN dir='1' ELSE
27           r_reg - 1;
28  -- logika izlaza ----------------------------------------------
29  q <= STD_LOGIC_VECTOR(r_reg);
30 END arch;
```

Pr. 8-17 Sinhroni ili asinhroni reset

Sekvencijalna kola po pravilu poseduju reset signal, koji se koristi za postavljanje kola u inicijalno stanje. Kao što znamo, reset signal može biti sinhronog ili asinhronog tipa. Izbor između ova dva tipa, između ostalog, zavisi i od specifičnih zahteva ciljne implementacione tehnologije. Na primer, kod FPGA tehnologije, najčešće se koristi reset signal sinhronog, a kod ASIC asinhronog tipa. Da bi se postigla što veća portabilnost VHDL opis, tj. da bi opis mogao lako da se implementira u različitim tehnologijama, korisno je u kôd uvrstiti generički parametar za izbor tipa reset signala.



Sl. 8-10 Binarni brojač sa sinhronim ili asinhronim resetom.

Realizacija asinhronog reseta je trivijalna (dovoljno je koristiti registre sa asinhronim resetom). S druge strane, realizacija sinhronog reseta zahteva modifikaciju logike sledećeg stanja, tako da ona pri neaktivnom signalu reseta generiše binarni kôd sledećeg, a pri aktivnom kôd inicijalnog stanja (najčešće, ″sve nule″). Ova modifikacija se može najlakše ostvariti uz pomoć multipleksera 2-u-1 koji će u zavisnosti od vrednosti reset signala da

propušta ka registru stanja bilo sledeće bilo inicijalno stanje. Na Sl. 8-10, ovaj princip je ilustrovan na primeru binarnog brojača: ili se koriste D flip-flopovi bez mogućnosti asinhronog reseta i multiplekser, ili D flip-flopovi sa asinhronim resetom.

Sledi VHDL opis binarnog brojača s parametarskim izborom tipa reset signala. Za *sync*=1, kôd se sintetiše u brojač sa sinhronim, a za *sync*≠1 u brojač sa asinhronim resetom. Selektivno generisanje kôda se javlja na dva mesta u ovom opisu. Prvi par *if generate* naredbi s komplementarnim uslovima generiše registar sa asinhronim resetom (linije 19-28), odnosno registar bez reset signala (linije 30-37). Drugi par *if generate* naredbi služi za povezivanje logike sledećeg stanja i registra stanja. U asinhronoj varijanti, ova veza je direktna (linije 41-43), dok se u sinhronom slučaju, između logike sledećeg stanja i registra stanja umeće multiplekser (linije 45-47).

```
1  ----------------------------------------------------------------
2  LIBRARY IEEE;
3  USE IEEE.STD_LOGIC_1164.ALL;
4  USE IEEE.NUMERIC_STD.ALL;
5  ----------------------------------------------------------------
6  ENTITY syn_or_asyn_counter IS
7    GENERIC(N : NATURAL;
8            SYNC : NATURAL);
9    PORT (q : OUT STD_LOGIC_VECTOR(N-1 DOWNTO 0);
10         clk, rst : IN STD_LOGIC);
11 END syn_or_asyn_counter;
12 ----------------------------------------------------------------
13 ARCHITECTURE arch OF syn_or_asyn_counter IS
14   SIGNAL r_reg : UNSIGNED(N-1 DOWNTO 0);
15   SIGNAL r_inc : UNSIGNED(N-1 DOWNTO 0);
16   SIGNAL r_next : UNSIGNED(N-1 DOWNTO 0);
17 BEGIN
18 -- registar sa asinhronim resetom ------------------------------
19   rst_reg_gen: IF(SYNC /= 1) GENERATE
20     PROCESS(clk, rst)
21     BEGIN
22       IF(rst = '1') THEN
23          r_reg <= (OTHERS => '0');
24       ELSIF(clk'EVENT AND clk='1') THEN
25          r_reg <= r_next;
26       END IF;
27     END PROCESS;
28   END GENERATE;
29 -- registar sa sinhronim resetom -------------------------------
30   reg_gen: IF(SYNC = 1) GENERATE
31     PROCESS(clk)
32     BEGIN
33       IF(clk'EVENT AND clk='1') THEN
34          r_reg <= r_next;
35       END IF;
36     END PROCESS;
37   END GENERATE;
38 -- logika sledeceg stanje - inkrementer ------------------------
39   r_inc <= r_reg + 1;
40 -- asinhroni reset ---------------------------------------------
41   asyn_rst: IF SYNC /= 1 GENERATE
42     r_next <= r_inc;
```

```
43   END GENERATE;
44 -- sinhroni reset ------------------------------------------------
45   syn_rst: IF SYNC = 1 GENERATE
46     r_next <= (OTHERS => '0') WHEN rst = '1' ELSE
47               r_inc;
48   END GENERATE;
49  -- logika izlaza -------------------------------------------------
50   q <= STD_LOGIC_VECTOR(r_reg);
51 END arch;
```

Razmotrimo još jedanput kolo ″Up-and-Down″ brojača iz Pr. 8-16. Logika sledećeg stanja ovog brojača se sastoji iz inkrementera, dekrementera i multipleksera, a spoljni upravljački signal *dir* svojom vrednošću određuje smer brojanja. Iako ″Up-and-Down″ brojač ne poseduje funkcionalni parametar *up*, efekat ovog generičkog parametra se može ″simulirati″ postavljanjem ulaznog signala *dir* na konstantnu vrednost.

Na primer, pretpostavimo da se u nekom projektu javila potreba za 8-bitnim brojačem naviše. Ovakav brojač možemo kreirati instanciranjem komponente ″Up-or-Down″ brojača, na sledeći način:

```
count8up: up_or_down_counter
   GENERIC MAP(N=>8, UP=>1)
   PORT MAP(clk=>clk, rst=>rst, q=>q);
```

Identičan brojač možemo kreirati i uz pomoć komponente ″Up-and-Down″ brojača:

```
count8up: up_and_down_counter
   GENERIC MAP(N=>8)
   PORT MAP(clk=>clk, rst=>rst, dir=>'1', q=>q);
```

Budući da je signal *dir* fiksiran na '1', brojač će uvek brojati naviše, baš kao brojač kreiran instanciranjem komponente *up_or_down_counter*.

Iako su dve prethodne instance funkcionalno identične, one ipak predstavljaju dva različita kola. Instanciranjem ″Up-or-Down″ brojača kreira se kolo koje sadrži samo neophodne funkcije, dok se instanciranjem brojača ″Up-and-Down″ kreira kolo obostranog brojača kompletne funkcionalnosti, s tim što je funkcija brojanja naniže onemogućena postavljanjem ulaza *dir* na ´1´.

Razlika između dve instance manifestovaće se i prilikom procesiranja VHDL kôda. Kao što znamo, procesiranje VHDL kôda uključuje tri koraka, analizu, elaboraciju i izvršenje (tj. sintezu). Naredba *if generate* se obrađuje u prvom koraku, tokom elaboracije, tako što se nepotrebni delovi kôda odstranjuju iz opisa. Tako pročišćen kôd se prosleđuje softveru za sintezu, koji se bavi samo preostalim kôdom. S druge strane, kad se koriste komponente pune funkcionalnosti, kao što je to ″Up-and-Down″ brojač, celokupan kôd se prosleđuje u fazu sinteze, bez obzira na to što neke funkcije eventualno isključene postavljanjem spoljašnjih ulaza na konstantne vrednosti. Na softveru za sintezu je da izvrši propagaciju konstantnih ulaznih vrednosti kroz kolo i eliminiše nepotrebnu logiku. U opštem slučaju, korišćenje naredbe za uslovno generisanje kôda (*if generate*) je bolji pristup, jer omogućava da se na jasniji način izdvoje opcioni delovi kola i da se nepotrebni delovi odstrane iz opisa pre njegove sinteze.

Selektivna sinteza hardvera se može takođe postići pomoću konfiguracije. Ovaj pristup se sastoji u tome da se za neko kolo kreira više zasebnih arhitektura specifičnih funkcionalnih karakteristika, a da se onda željena funkcionalnost bira povezivanjem entiteta sa odgovarajućom arhitekturom (v. 7.3). Na primer, parametrizovan opis ″Up-or-Down″

brojača se može razložiti na dve arhitekture, tako da jedna opisuje brojač naviše (*Up*), a druga brojač naniže (*Down*), a da se onda posredstvom konfiguracije bira po potrebi jedna od njih.

Takođe, više alternativnih arhitektura se može objediniti u jedinstvenu parametrizovanu arhitekturu u kojoj će izbor željenih funkcionalnih karakteristika biti omogućen uz pomoć funkcionalnih parametara i naredbi *if generate*.

Ne postoji opšte pravilo kad je bolje koristiti funkcionalne parametre, a kad konfiguraciju. Parametrizovan kôd je teže napisati, budući da jednim opisom treba obuhvatiti više različitih verzija jednog istog kola. S druge strane, ako se za svaku pojedinačnu kombinaciju funkcija kreira zasebna arhitektura, njihov broj može postati veliki. Primera radi, da bi se pokrile sve moguće kombinacije funkcija brojača koji bi imao mogućnost izbora smera brojanja (*Up/Down*) i tipa reset signala (sinhron/asinhron) bilo bi neophodno napisati četiri različite arhitekture.

8.5. FOR LOOP

U ovom poglavlju biće predstavljeno nekoliko primera parametrizovanih opisa u kojima se koristi sekvencijalna naredba *for loop*. Ova naredba je uvedena u poglavlju 5.4, a njena pojednostavljena sintaksa je sledećeg oblika:

```
FOR indeks IN opseg LOOP
   sekvencijalne naredbe;
END LOOP;
```

Naredbe *for generate* i *for loop* su slične po sintaksi i način rada, s tom razlikom što se prva koristi isključivo u konkurentnom, a druga u sekvencijalnom kôdu. Upravo iz tog razloga, naredba *for loop* je fleksibilnija i opštija od *for generate*. Međutim, kao i za *for generate*, opseg indeksa naredbe *for loop* mora biti statički ako se ona koristi u kôdu za sintezu.

Pr. 8-18 Binarni dekoder - parametrizovan opis

Binarni dekoder je razmatran u više navrata u prethodnim glavama (v. Pr. 4-7, Pr. 4-13, Pr. 5-7 i Pr. 5-10). U Pr. 5-10, za opis dekodera korišćena je sekvencijalna naredba *case*. Međutim, s obzirom na to što je broj grana u *case* naredbi jednak broju izlaza dekodera, ovakva realizacija nije pogodna kao osnova za kreiranje parametrizovanog opisa. To isto važi i za opis dekodera pomoću konkurentne naredbe *when* iz Pr. 4-7. Fleksibilniji opis dekodera, koji se može lako parametrizovati, zasnovan je na naredbi *for loop*. Odgovarajući VHDL kôd je dat ispod. Opis sadrži dimenzioni parametar *n* koji definiše broj ulaza dekodera. Dekoder sa *n* ulaza poseduje 2^n izlaza, od kojih je aktivan uvek samo jedan (postavljen na ´1´) i to onaj čiji je indeks, u vidu *n*-bitnog binarnog broja, postavljen na ulazu kola. U kôdu treba obratiti pažnju na uslov *if* naredbe iz linije 17, u kome se ulazna vrednosti tipa *std_logic_vector* konvertuje u tip *integer* radi poređenja s indeksom petlje.

```
1  ----------------------------------------------------------------
2  LIBRARY IEEE;
3  USE IEEE.STD_LOGIC_1164.ALL;
4  USE IEEE.NUMERIC_STD.ALL;
5  ----------------------------------------------------------------
6  ENTITY bin_decoder IS
7    GENERIC(N : NATURAL);
8    PORT(a : IN  STD_LOGIC_VECTOR (N-1 DOWNTO 0);
```

```
9          y : OUT  STD_LOGIC_VECTOR (2**N-1 DOWNTO 0));
10 END bin_decoder;
11 ----------------------------------------------------------------
12 ARCHITECTURE loop_arch OF bin_decoder IS
13 BEGIN
14   PROCESS(a)
15 BEGIN
16     FOR I IN 0 TO 2**N-1 LOOP
17       IF(I = TO_INTEGER(UNSIGNED(a))) THEN
18         y(I) <= '1';
19       ELSE
20         y(I) <= '0';
21       END IF;
22     END LOOP;
23   END PROCESS;
24 END loop_arch;
```

Pr. 8-19 Prioritetni koder - parametrizovan opis i konceptualna implementacija

VHDL opis prioritetnog kodera, zasnovan na naredbi *when*, predstavljen je u Pr. 4-9. Međutim, iz istih razloga kao kod binarnog dekodera, opis iz Pr. 4-9 nije pogodan za parametrizaciju. Zato se u parametrizovanom opisu prioritetnog kodera iz ovog primera koristi naredba *for loop*. Prioritetni koder ima n ulaza i $\log_2 n$ izlaza. Ulazima su pridruženi prioriteti, a vrednost prisutna na izlazu, interpretirana kao neoznačen binarni broj, predstavlja indeks (redni broj) aktivnog ulaza najvišeg prioriteta. Prioritetni koder takođe poseduje izlaz *z* koji je aktivan ako je barem jedan ulaz postavljen na ´1´. U datom kôdu, usvojeno je da najviši prioritet ima ulaz $a(n\text{-}1)$, a najniži ulaz $a(0)$. Kôd sadrži dva procesa. Prvi proces (linije 19-27) realizuje funkciju prioritetnog kodiranja, dok drugi (linije 29-35) opisuje funkciju n-to ulaznog ILI kola kojom se generiše vrednost izlaznog signala *z*. Iako oba procesa sadrže *for loop* naredbu, samo prvi zahteva detaljnije objašnjenje. Na početku ovog procesa, izlazni signal *b* se postavlja na ″sve nule″ (linija 21). Naredba *for loop* ″skenira″ ulaze, počev od ulaza najnižeg, pa do ulaza navišeg prioriteta. Uvek kada se naiđe na ativan ulaz, signal *b* se postavlja na kôd tog ulaza, odnosno dodeli mu se vrednost tekućeg indeksa petlje u binarnom obliku (linija 24). Ako postoji više aktivnih ulaza, signalu *b* će više puta biti dodeljena nova vrednost. Međutim, kao što znamo, višestruka dodela vrednosti istom signalu u procesu razrešava se tako što signal na kraju procesa dobija poslednje dodeljenu vrednost, a to će upravo biti kôd aktivnog ulaza navišeg prioriteta. Ako ni jedan ulaz nije aktivan, *b* će zadržati početnu vrednost, tj. nulu.

```
1  ----------------------------------------------------------------
2  LIBRARY IEEE;
3  USE IEEE.STD_LOGIC_1164.ALL;
4  USE IEEE.NUMERIC_STD.ALL;
5  USE work.nas_paket.ALL;
6  ----------------------------------------------------------------
7  ENTITY pencoder IS
8  GENERIC(N : NATURAL);
9    PORT( a : IN  STD_LOGIC_VECTOR (N-1 DOWNTO 0);
10         b : OUT  STD_LOGIC_VECTOR (log2c(N)-1 downto 0);
11         z : OUT STD_LOGIC);
12 END pencoder;
13 ----------------------------------------------------------------
14 ARCHITECTURE loop_arch OF pencoder IS
```

```
15   CONSTANT M : NATURAL := log2c(N);
16   SIGNAL p : STD_LOGIC_VECTOR(N-1 DOWNTO 0);
17 BEGIN
18 -- prioritetno kodiranje ----------------------------------------
19   PROCESS(a)
20   BEGIN
21     b <= (OTHERS => '0');
22     FOR I IN 0 TO N-1 LOOP
23       IF(a(i) = '1') THEN
24         b <= STD_LOGIC_VECTOR(TO_UNSIGNED(I,M));
25       END IF;
26     END LOOP;
27   END PROCESS;
28 -- N-to ulazno ILI kolo ---------------------------------------
29   PROCESS(a, p)
30   BEGIN
31     p(0) <= a(0);
32     FOR I IN 1 TO (N-1) LOOP
33       p(i) <= a(i) OR p(I-1);
34     END LOOP;
35   END PROCESS;
36   z <= p(N-1);
37 END loop_arch;
38 ----------------------------------------------------------------
```

Predstavljeni VHDL opis prioritetnog kodera je specifičan po tome što nije izveden na osnovu neke polazne konceptualne predstave o strukturi kola, kako je uobičajeno, već na osnovu apstraktnog opisa funkcije kola. Međutim, i pored toga, ovaj kôd se može sintetizovati. Da bi smo se uverili da je to zaista tako, razmotrićemo kako se petlja za prioritetno kodiranje (linije 22-26) prevodi u konceptualni dijagram. Razmotavanjem *for loop* petlje dobijamo sledeći kôd, uz pretpostavku da važi n=4:

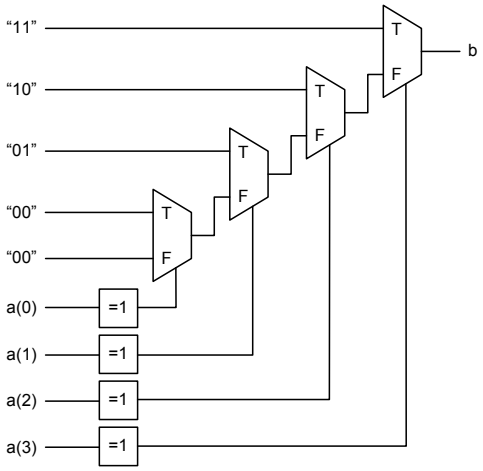
```
b <= "00";
IF (a(0) = '1') THEN
  b <= "00";
END IF;
IF (a(1) = '1') THEN
  b <= "01";
END IF;
IF (a(2) = '1') THEN
  b <= "10";
END IF;
IF (a(3) = '1') THEN
  b <= "11";
END IF;
```

U razmotanom kôdu obavlja se niz uzastopnih uslovnih dodela vrednosti istom signalu, *b*. Imajući u vidu da je konačna vrednost signala određena poslednjom dodelom, prethodni niz naredbi se može prevesti u jednu *if* naredbu s više *elsif* grana:

```
IF (a(3) = '1') THEN
  b <= "11";
ELSIF (a(2) = '1') THEN
  b <= "10";
ELSIF (a(1) = '1') THEN
  b <= "01";
ELSIF (a(0) = '1') THEN
```

```
  b <= "00";
ELSE
  b <= "00";
END IF;
```

Naredba *if-then-elsif-else* se direktno prevodi u konceptualni dijagram sa Sl. 8-11, koji u suštini predstavlja kaskadnu, prioritetnu mrežu multipleksera 2-u-1.



Sl. 8-11 Konceptualni dijagram prioritetnog kodera.

Pr. 8-20 Višeulazno I kolo - parametrizovan opis i konceptualna implementacija

Sledi parametrizovan opis logičke I funkcije za proizvoljan broj operanada. U kôdu se koristi varijabla *v* kao pomoćna promenljiva u petlji u kojoj se logička I operacija primenjuje redom na sve ulazne bitove. Za razliku od signala, dodela vrednosti varijabli u procesu je trenutna, što znači da se *v* ažurira u svakoj iteraciji petlje.

```
1  ----------------------------------------------------------------------
2  LIBRARY IEEE;
3  USE IEEE.STD_LOGIC_1164.ALL;
4  ----------------------------------------------------------------------
5  ENTITY andN IS
6  GENERIC(N : NATURAL);
7    PORT( a : IN  STD_LOGIC_VECTOR (N-1 DOWNTO 0);
8          y : OUT  STD_LOGIC);
9  END andN;
10 ----------------------------------------------------------------------
11 ARCHITECTURE loop_arch OF andN IS
12 BEGIN
13   PROCESS(a)
14     VARIABLE v : STD_LOGIC;
15   BEGIN
16     v := a(0);
17     FOR I IN 1 TO (N-1) LOOP
18       v := v AND a(i);
19     END LOOP;
20     y <= v;
```

```
21  END PROCESS;
22 END loop_arch;
23 ------------------------------------------------------------------
```

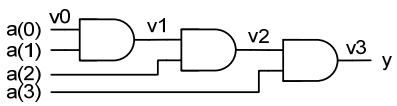
Prethodni VHDL kôd nalikuje programu pisanom u programskom jeziku. Međutim, što je nivo apstraktnosti i deskriptivnosti kôda viši, to izvođenje odgovarajuće konceptualne implementacije zahteva više napora. U svakom slučaju, ključ za konverziju kôda u hardver je u identifikovanju konstrukcija koje imaju direktnu hardversku interpretaciju. Prvi korak u tom procesu je razmotavanje petlje. Pod pretpostavkom da je n=4, petlja se razmotava u sledeći kôd:

```
v := a(0);
v := a(1) AND v;
v := a(2) AND v;
v := a(3) AND v;
y <= v;
```

Da bi se izbeglo samo-referenciranje, potrebno je izvršiti preimenovanje varijable, tako da se u svakoj sledećoj naredbi koristi novo ime za varijablu s desne strane znaka dodele (v. 5.1.4):

```
v0 := a(0);
v1 := a(1) AND v0;
v2 := a(2) AND v1;
v3 := a(3) AND v2;
y <= v3;
```

Sada postaje jasno da varijable predstavljaju veze između dvoulaznih I kola, na način kao na Sl. 8-12.



Sl. 8-12 Konceptualna implementacija višeulaznog I kola.

Pr. 8-21 Brojač jedinica - parametrizovan opis i konceptualna implementacija

Brojač jedinica je kombinaciono kolo koje prebrojava 1-ce u ulaznom višebitnom signalu. U parametrizovanom opisu brojača jedinica iz ovog primera koristi se naredba *for loop* i varijabla kao pomoćna promenljiva za sumiranje 1-ca:

```
1  ------------------------------------------------------------------
2  LIBRARY IEEE;
3  USE IEEE.STD_LOGIC_1164.ALL;
4  USE IEEE.NUMERIC_STD.ALL;
5  USE work.nas_paket.ALL;
6  ------------------------------------------------------------------
7  ENTITY once_counter IS
8    GENERIC(N : NATURAL);
9    PORT( a : IN  STD_LOGIC_VECTOR (N-1 DOWNTO 0);
10         b : OUT  STD_LOGIC_VECTOR (log2c(N)-1 DOWNTO 0));
11 END once_counter;
12 ------------------------------------------------------------------
13 ARCHITECTURE loop_arch OF once_counter IS
```

```
BEGIN
  PROCESS(a)
    VARIABLE sum : UNSIGNED(log2c(N)-1 DOWNTO 0);
  BEGIN
    sum := (OTHERS => '0');
    FOR I IN 0 TO N-1 LOOP
      IF(a(i) = '1') THEN
        sum := sum + 1;
      END IF;
    END LOOP;
   b <= STD_LOGIC_VECTOR(sum);
  END PROCESS;
END loop_arch;
-------------------------------------------------------------------
```

Kreiranje konceptualne implementacije datog kôda zahteva veće angažovanje. Pretpostavimo da je n=3. Petlja se razmotava u sledeći kôd:

```
sum := 0;
IF (a(0) = '1') THEN
  sum := sum + 1;
END IF;
IF (a(1) = '1') THEN
  sum := sum + 1;
END IF;
IF (a(2) = '1') THEN
  sum := sum + 1;
END IF;
b <= sum;
```

Prosto preimenovanje varijabli, kao u prethodnom primeru, nije korektno:

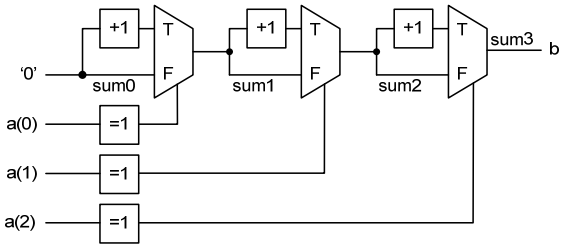
```
sum0 := 0;
IF (a(0) = '1') THEN
  sum1 := sum0 + 1;
END IF;
IF (a(1) = '1') THEN
  sum2 := sum1 + 1;
END IF;
IF (a(2) = '1') THEN
  sum3 := sum2 + 1;
END IF;
b <= sum3;
```

Očigledno, modifikovani kôd nije funkcionalno identičan polaznom, jer se tekuća vrednost sume ne prenosi korektno iz iteracije u iteraciju. Ako važi $a(i)$= ´0´, naredba za sumiranje se preskače i varijabla sum_{i+1} ostaje neinicijalizovana. Ispravna modifikacija zahteva da se svaka *if-then* naredba proširi *else* granom u kojoj će se u sledeću varijablu sum_{i+1} prepisati vrednost prethodne, sum_i, u slučaju da važi $a(i)$= ´0´:

```
sum0 := 0;
IF (a(0) = '1') THEN
  sum1 := sum0 + 1;
ELSE
  sum1 := sum0;
END IF;
IF (a(1) = '1') THEN
```

```
  sum2 := sum1 + 1;
ELSE
  sum2 := sum1;
END IF;
IF (a(2) = '1') THEN
  sum3 := sum2 + 1;
ELSE
  sum3 := sum2;
END IF;
b <= sum3;
```

Modifikovani kôd se preslikava u iterativnu hardversku strukturu sa Sl. 8-13. Svakoj *if-then-else* naredbi odgovara jedan bazični gradivni blok koji čine po jedan multiplekser i inkrementer. Ako je odgovarajući ulazni bit jednak ´0´, u sledeći stepen se prenosi suma iz prethodnog. U suprotnom, ako je ulazni bit jednak ´1´, u sledeći stepen se šalje suma iz prethodnog stepena uvećana za 1.



Sl. 8-13 Konceptualna implementacija brojača jedinica.

8.5.1. EXIT i NEXT

Naredbe *exit* i *next* su sekvencijalne naredbe koje se koriste isključivo unutar naredbe *for loop* radi promene toka izvršenja petlje. Naredba *exit* prekida izvršenje *for loop* naredbe - petlja se momentalno napušta, a preostale iteracije se zanemaruju. Naredba *next* prekida izvršenje tekuće iteracije – petlja momentalno prelazi u sledeću iteraciju, a preostale naredbe tekuće iteracije se zanemaruju. Naredbe *exit* i *next* su korisne u apstraktnom modeliranju, jer u nekim slučajevima mogu značajno da pojednostave kôd tela petlje, ali ih je teško sintetizovati, pa se zbog toga retko koriste u kôdu za sintezu. Šta više, pojedini softveri za sintezu ne podržavaju ove naredbe. U narednim primerima biće ilustrovana primena naredbi *exit* i *next* i pokazano kako se izvodi njihova konceptualna implementacija. Takođe, biće ukazano kako se ove dve naredbe mogu zameniti naredbom *if*.

Exit. Sintaksa *exit* naredbe je:

```
    EXIT WHEN uslov;
```

Član *uslov* je logički izraz koji definiše uslov izlaska iz petlje. Ako je uslov tačan, izvršenje petlje se prekida. Treba napomenuti da naredba *exit* može egzistirati i bez definisanog uslova, kao u sledećem kontekstu:

```
IF (uslov) THEN
. . .
  EXIT;
ELSE
. . .
```

Pr. 8-22 Višeulazno I kolo – realizacija pomoću naredbe *″for loop - exit – end loop″*

Parametrizovan opis višeulaznog I kola, zasnovan na naredbi *for loop*, dat je u Pr. 8-20. Naredba *for loop* iz Pr. 8-20 redom skenira i ″AND-uju″ ulazne bitove, od prvog do poslednjeg. U ovom primeru, iskoristićemo osobinu logičke I operacije da je $x \cdot 0 = 0$, odnosno da je rezultat I operacije jednak 0 ako u ulaznom niz postoji barem jedna nula. Dakle, umesto da se prolazi kroz ceo ulazni niz, biti po bit, skeniranje ulaza se može prekinuti čim se naiđe na prvu nulu. VHDL kôd koji sledi, zasnovan je na ovom opažanju. U kôdu se koristi naredba *for loop* za skeniranje ulaznih bitova i naredba *exit* koja prekida petlju pri nailasku na prvu nulu.

Gledano iz perspektive programskih jezika, kôd sa *exit* naredbom je efikasniji (brži) u odnosu na kôd iz Pr. 8-20, gde se petlja uvek izvršava do kraja. Budući da je efekat naredbe *exit* takav da se petlja završava nailaskom na prvu nulu, vreme izvršenja programa u proseku biće duplo kraće nego da se petlja uvek izvršava do kraja. Međutim, ako se dati kôd realizuje u hardveru, poboljšanja neće biti. Hardver ne može dinamički da se ″skupi″ ili ″proširi″ zavisno od ulaznog podatka, već mora biti fizički prisutan u obimu koji je neophodan da bi se obradila proizvoljna ulazna sekvenca. Prevremeni izlazak iz petlje ne samo da ne doprinosi ubrzanju rada već unosi dodatni hardver i usporava rad kola, kao što će to biti pokazano kroz konceptualnu implementaciju kôda iz sledećeg primera

```
1  ----------------------------------------------------------------
2  ARCHITECTURE loop_exit_arch OF andN IS
3  BEGIN
4   PROCESS(a)
5    VARIABLE v : STD_LOGIC;
6   BEGIN
7    v := ´1´; -- podrazumevana vrednost
8    FOR I IN 0 TO (N-1) LOOP
9     IF a(I) = ´0´ THEN
10      v := ´0´;
11      EXIT;
12    END IF;
13   END LOOP;
14   y <= v;
15  END PROCESS;
16 END loop_exit_arch;
```

Pr. 8-23 Brojač vodećih nula - Konceptualna implementacija *for loop* petlje sa *exit* naredbom

Brojač vodećih nula je kombinaciono kolo koje određuje broj nula koje se nalaze na početku višebitnog ulaznog signala. Na primer, broj vodećih nula u vektoru ″10011″ je 0, a u vektoru ″00110″ dve. U VHDL kôdu koji sledi, naredba *for loop* se koristi za prebrojavanje nula u ulaznom vektoru, a naredba *exit* za izlazak iz petlje kada se naiđe na prvu 1-cu.

```
1  ----------------------------------------------------------------
2  LIBRARY IEEE;
3  USE IEEE.STD_LOGIC_1164.ALL;
4  USE IEEE.NUMERIC_STD.ALL;
5  ----------------------------------------------------------------
6  ENTITY zero_counter IS
```

```
7    GENERIC(N : NATURAL);
8    PORT(a : IN STD_LOGIC_VECTOR(N-1 DOWNTO 0);
9         z : OUT STD_LOGIC_VECTOR(log2(N)-1 DOWNTO 0));
10 END zero_counter;
11 ----------------------------------------------------------------------
12 ARCHITECTURE loop_exit_arch OF zero_counter IS
13 BEGIN
14   PROCESS(a)
15     VARIABLE sum: UNSIGNED(log2(N)-1 DOWNTO 0);
16   BEGIN
17    sum := (OTHERS => '0');
18    FOR I IN N-1 DOWNTO 0 LOOP
19      IF a(I) = '1' THEN
20         EXIT;
21      ELSE
22         sum := sum + 1;
23      END IF;
24    END LOOP;
25    z <= STD_LOGIC_VECTOR(sum);
26   END PROCESS;
27 END loop_exit_arch;
28 ----------------------------------------------------------------------
```

Naredba *for loop* sa *exit* naredbom se ne može direktno sintetizovati. Problem je u tome što *exit* naredba onemogućava razmotavanje petlje, jer se unapred ne može znati kada će uslov za izlazak iz petlje biti ispunjen. Sinteza petlje sa *exit* naredbom zasnovana je na ″emulaciji″ naredbe *exit* dodatnom logikom koja će omogućiti ″preskakanje″ preostalih iteracija petlje onda kada uslov za napuštanje petlje postane ispunjen.

Modifikovana verzija kôda data je u nastavku. Kao što možemo videti, u kôdu sada postoje dve *for loop* petlje. Ideja je sledeća. Uvedena je još jedna višebitna varijabla, *bypass*, čiji svaki bit kontroliše rad jednog stepena u iterativnoj strukturi. Bitovi ove varijable se postavljaju u prvoj petlji (linije 12-18), a zatim se koriste u drugoj petlji (linije 20-26). Bit *bypass*(*i*) postavljen na ´1´ znači da operaciju inkrementiranja koja se obavlja u *i*-tom stepenu treba premostiti, odnosno da rezultat iz stepena *i*+1 treba, bez promene, preneti kroz *i*-ti stepen. U suprotnom, *bypass*(*i*) = ´0´ ukazuje da *i*-ti stepen treba da obavi svoju normalnu operaciju.

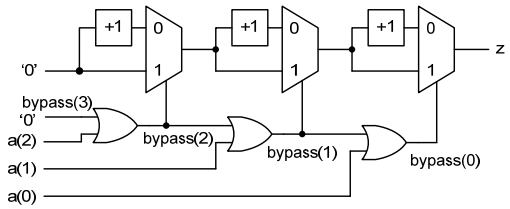
```
1  ----------------------------------------------------------------------
2  ARCHITECTURE bypass_arch OF zero_counter IS
3  BEGIN
4   PROCESS(a)
5     VARIABLE sum: UNSIGNED(log2(N)- 1 DOWNTO 0);
6     VARIABLE bypass: STD_LOGIC_VECTOR(N DOWNTO 0);
7   BEGIN
8  -- Inicijalizacija ------------------------------------------------
9     sum := (OTHERS => '0');
10    bypass(N) <= '0';
11 -- Postavljanje bypass indikatora --------------------------------
12    FOR I IN N-1 DOWNTO 0 LOOP
13      IF a(I) = '1' THEN
14         bypass(I) := '1';
15      ELSE
16         bypass(I) := bypass(I+1);
17      END IF;
18    END LOOP;
```

```
19 -- Brojanje vodecih nula ---------------------------------------
20    FOR I IN N-1 DOWNTO 0 LOOP
21      IF bypass(I) = '0' THEN
22        IF a(I) = '0' THEN
23          sum := sum + 1;
24        END IF;
25      END IF;
26    END LOOP;
27    z <= STD_LOGIC_VECTOR(sum);
28  END PROCESS;
29 END bypass_arch;
30 ----------------------------------------------------------------
```

Zapazimo da ako važi *bypass(i)=´0´*, tada mora da važi i *a(i)=´0´*. To znači da ispitivanje uslova *a(i)=´0´* iz druge *for loop* petlje nije neophodno i da se može izostaviti:

```
FOR I IN N-1 DOWNTO 0 LOOP
  IF bypass(I) = '0' THEN
   sum := sum + 1;
  END IF;
END LOOP;
```

Budući da u kôdu više ne postoji naredba *exit*, do konceptualne implementacije se može doći razmotavanjem petlji. Odgovarajući dijagram, za slučaj n=3, prikazan je na Sl. 8-14. Prva petlja se transformiše u lanac ILI kola, s obzirom na to što je efekat naredbe *if* (linije 13-17) identičan ILI operaciji nad bitovima $a(i)$ i $bypass(i+1)$. Druga *for loop* petlja se transformiše u strukturu koja je slična strukturi brojača jedinica iz Pr. 8-21. Uočimo da onda kada u nekom stepenu *bypass* postane jednako ´1´, ova 1-ca se prenosi kroz sve naredne stepene, što za posledicu ima preskakanje svih preostalih operacija inkrementiranja.



Sl. 8-14 Konceptualna implemenacija brojača vodećih nula.

Next. Sintaksa naredbe *next* je sledećeg oblika

```
NEXT WHEN uslov;
```

Član *uslov* je logički iskaz koji može biti tačan ili netačan. Ako je netačan, naredba *next* se preskače i nema nikakvog efekta na izvršenje kôda tela petlje; ako je tačan, efekat naredbe *next* je taj da se preskaču sve preostale naredbe iz tela petlje i momentalno se prelazi na izvršenje sledeće iteracije petlje. Slično naredbi *exit*, deo *"when uslov"* nije neophodan ako se naredba *next* nalazi unutar naredbe *if*.

Pr. 8-24 Brojač jedinica – realizacije pomoću naredbe *next*

Parametrizovan VHDL opis brojača jedinica predstavljen je u Pr. 8-21. U ovom primeru izložena je modifikovana arhitektura brojača jedinica u kojoj se naredba *next* koristi za preskakanje operacije inkrementiranja u slučajevima kada je ulazni bit jednak nuli.

```
1  ------------------------------------------------------------
2  ARCHITECTURE loop_next_arch OF zero_counter IS
3  BEGIN
4    PROCESS(a)
5      VARIABLE sum: UNSIGNED(log2(N)-1 DOWNTO 0);
6    BEGIN
7     sum := (OTHERS => '0');
8     FOR I IN 0 TO N-1 LOOP
9       NEXT WHEN a(I)='0';
10      sum := sum + 1;
11    END LOOP;
12    b <= STD_LOGIC_VECTOR(sum);
13   END PROCESS;
14 END loop_next_arch;
```

Naredba *next* je u izvesnom smislu ″suprotna″ naredbi *if* : za ispunjen uslov, *next* preskače, dok *if* izvršava obuhvaćene naredbe. Imajući to u vidu, *next* naredba se može izraziti preko *if* naredbe, kao u sledećem primeru:

```
FOR ... LOOP                          FOR ... LOOP
  sekvencijalna naredba 1;              sekvencijalna naredba 1;
  NEXT WHEN uslov;                      IF(NOT uslov) THEN
  sekvencijalna naredba 2;      →         sekvencijalna naredba 2;
END LOOP;                               END IF;
                                      END LOOP;
```

Treba naglasiti da se *if* naredba mnogo češće koristi od *next* naredbe budući da je modularnija i deskriptivnija. Prethodni primer takođe ukazuje na koji način bi se obavila sinteza *next* naredbe.

9. RTL PROJEKTOVANJE

Složeni postupci izračunavanja često se opisuju u vidu algoritama. Algoritam definiše niz koraka ili akcija koje treba sprovesti nad ulaznim podacima da bi se dobio željeni rezultat. Algoritmi se najčešće realizuju u vidu programa, koji se pišu u konvencionalnim programskim jezicima (tj. u softveru) i izvršavaju na računarima opšte namene. Međutim, radi postizanja boljih performansi i veće efikasnosti, nekada je bolje ili čak neophodno algoritam realizovati direktno u namenski projektovanom hardveru. RTL (engl. ***R****egister* ***T****ransfer* ***L****evel*) projektovanje (ili projektovanje na registarskom, odnosno mikroarhitekturnom nivou apstrakcije) je metodologija projektovanja koja pruža sistematski način za konverziju algoritama u hardver.

9.1. RTL metodologija projektovanja

Algoritam predstavlja detaljan opis toka izvršenja nekog zadatka ili postupka rešavanja nekog problema u vidu sekvence akcija ili koraka. Budući da je semantika programskih jezika zasnovana na sekvencijalnom modelu izračunavanja, algoritam se može lako opisati u vidu programa, koji se potom kompajlira i izvršava na računaru opšte namene. Razmotrimo jedan jednostavan zadatak koji sumira četiri elementa niza, deli sumu sa 8 i zaokružuje rezultat na najbliži ceo broj. Pseudo kôd odgovarajućeg algoritma je:

```
size = 4;
sum = 0;
for i in (0 to size-1) do {
   sum = sum + a(i);}
q = sum / 8;
r = sum rem 8;
if(r > 3) {
   q = q + 1;}
outp = q;
```

Nakon dodele početnih vrednosti promenljivama *size* i *sum*, algoritam u petlji sabira elemente niza *a*, jedan po jedan, i njihov zbir smešta u promenljivu *sum*. Posle toga slede operacije deljenja (/) i ostatka (*rem*) kojim se izračunava količnik (*q*) i ostatak (*r*) deljenja sume sa 8. Konačno, ako je ostatak veći od 3 na količnik se dodaje 1 radi zaokruživanja. Opisani primer ilustruje dve osnovne osobine svakog algoritma:

Goran Lj. Đorđević, *Arhitetkture mikrosistema*, 2009.

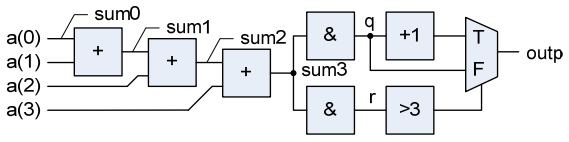
- *Upotreba promenljivih*. Promenljive u algoritmu mogu se razumeti kao memorijske lokacije za čuvanje međurezultata izračunavanja. Ime promenljive ukazuje na simboličku adresu memorijske lokacije. Na primer, u drugoj naredbi, 0 se smešta u memorijsku lokaciju sa simboličkim imenom *sum*. Ili, unutar *for* petlje *a*(*i*) se sabira sa tekućim sadržajem promenljive *sum* i dobijeni zbir se smešta ponovo u memorijsku lokaciju sa simboličkim imenom *sum*, itd.
- *Sekvencijalno izvršenje*. Naredbe algoritma se izvršavaju sekvencijalno, tj. jedna po jedna, po strogo definisanom redosledu. Na primer, sumiranje elemenata niza se mora obaviti pre deljenja sume, deljenje nakon sumiranja, a zaokruživanje nakon deljenja. Napomenimo da tok izvršenja algoritma može da zavisi i od uslova definisanih u naredbama kao što su *for* ili *if*.

Razmatrani algoritam, predstavljen u vidu pseudo kôda, može se relativno lako prevesti u ekvivalentan VHDL kôd, uz neophodna sintaksna prilagođenja i pretpostavku da su *sum*, *q* i *r* 8-bitni brojevi:

```
. . .
CONSTANT size : INTEGER := 4;
SIGNAL outp : STD_LOGIC_VECTOR(7 DOWNTO 0);
. . .
PROCESS(a)
  VARIABLE sum, q, r : STD_LOGIC_VECTOR(7 DOWNTO 0);
BEGIN
  sum := a(0);
  FOR i IN 1 TO size-1 LOOP
    sum := sum + a(i);
  END LOOP;
  q <= „000" & sum(7 DOWNTO 3);
  r <= „00000" & sum(2 DOWNTO 0);
  outp <= q + 1 WHEN (r > 3) ELSE
          q;
END PROCESS;
```

Nakon razmotavanja *for loop* petlje i preimenovanja varijable *sum* (slično kao u primeru iz odeljka 5.1.4), dolazimo do oblika kôda koji je pogodan za konceptualnu implementaciju (Sl. 9-1):

```
sum0 := a(0);
sum1 := sum0 + a(1);
sum2 := sum1 + a(2);
sum3 := sum2 + a(3);
q <= „000" & sum(7 DOWNTO 3);
r <= „00000" & sum(2 DOWNTO 0);
outp <= q + 1 WHEN (r > 3) ELSE
        q;
```



Sl. 9-1 Konceptualna implementacija

Rezultat sinteze je struktura kaskadno povezanih hardverskih blokova, gde svaki blok odgovara jednoj operaciji iz algoritma, kao što se može videti na Sl. 9-1. Promenljive više nisu direktno vidljive u ovoj strukturi (zato što su varijable i signali preslikani na interne

veze u kolu), a sekvencijalno izvršenje, karakteristično za pseudo kôd, samo je implicitno ugrađeno u način kako su povezani hardverski blokovi. Sintetizovano kolo se može razumeti kao kombinaciona funkcionalna jedinica koja obavlja jednu složenu aritmetičku operaciju. Međutim, način na koji se u hardveru obavlja zahtevana funkciju suštinski se razlikuje od semantike (načina rada) polaznog sekvencijalnog algoritma. U hardverskom rešenju sa Sl. 9-1, hardverski blokovi rade u paraleli, što omogućava preklapanje izvršenja pojedinih operacija, a time i ubrzanje procesa izračunavanja. Na primer, nakon izračunavanja zbira četiri ulazne vrednosti, operacije deljenja i određivanja ostatka deljenja (predstavljene blokovima sa upisanim operatorom konkatenacije, &) mogu se obaviti istovremeno. S druge strane, u softverskom rešenju, koje je namenjeno za izvršenje na računaru opšte namene, takva ″preklapanja″ izvršenja operacija nisu moguća, jer se naredbe algoritma uvek izvršavaju jedna za drugom, po definisanom redosledu.

Međutim, bez obzira na dobitak u performansama, digitalna kola koja se realizuju na opisan način nisu dovoljno univerzalna niti su, u najvećem broju slučajeva, najbolje hardversko rešenje koje postoji za dati problem. Na primer, pretpostavimo da se u razmatranom primeru vrednost konstante *size* promeni sa 4 na 20. Ova jednostavna modifikacije dovela bi do drastičnog povećanje složenosti sintetizovanog hardvera, jer bi umesto 4 biti potrebno čak 20 sabirača. Ili, pretpostavimo da *size* više nije konstanta, već ulazna promenljiva koja može imati različitu vrednost pri svakom novom pozivu algoritma. Ova modifikacija, iako gotovo neprimetna sa stanovišta sekvencijalnog izvršenja, stvorila bi nepremostive poteškoće prilikom realizacije hardvera – hardver se ne može dinamički ″širiti″ i ″skupljati″.

Prethodni primer ukazuje na ograničenja i nefleksibilnost direktnog preslikavanja sekvencijalnog algoritma na hardversku strukturu, odnosno sugeriše na drugačiji pristupa u projektovanju hardvera. Da bi se sekvencijalni algoritam na efikasan način realizovao u hardveru, neophodno je na neposredniji način realizovati koncept promenljivih i sekvencijalnog izvršenja. RTL metodologija služi upravo ovoj svrsi, a njene ključne postavke su sledeće:

- Registri se koriste kao zamena za promenljive.
- Hardver koji obavlja operacije sadržane u algoritmu (kao npr. +, -, /, ...) realizuju se u vidu kombinacionih blokova.
- Hardver koji reguliše redosled izvršenja operacija realizuje se u vidu upravljačke jedinice.

U RTL projektovanju, registri se koriste kao memorijski elementi opšte namene za čuvanje međurezultata izračunavanja, baš kao i promenljive u algoritmu. Primera radi, razmotrimo kako bi se jedna tipičnu naredbu iz pseudo kôda, $a = a + b$, realizovala u hardveru. Promenljivama *a* i *b*, u hardveru odgovaraju registri *a_reg* i *b_reg*. Naredba se izvršava tako što se sadržaji registara *a_reg* i *b_reg* sabiraju i zbir upisuje nazad u *a_reg* pod dejstvom prve sledeće rastuće ivice taktnog signala. Kad se algoritam realizuje u hardveru, sve manipulacije nad podacima se ostvaruju pomoću namenski realizovanog hardvera. Na primer, za realizaciju prethodne naredbe neophodan je sabirač.

Hardver za manipulaciju nad podacima, zajedno sa registrima i strukturom koja ih povezuje zajedničkim imenom se zove *staza podataka.* Imajući u vidu da algoritam definiše **sekvencu** operacija, neophodan je i hardver koji će regulisati kada i koju operaciju treba izvršiti u stazi podataka. Ova struktura se naziva *upravljačkom jedinicom* i realizuje se u vidu konačnog automata. Stanja konačnog automata nameću redosled algoritamskih koraka, dok višestruki prelazi između stanja ″imitiraju″ grananja i petlje iz algoritma.

9.1.1. Naredba registarskog prenosa

Naredba registarskog prenosa je osnovna operacija u RTL projektovanju, koja odgovara naredbi dodele iz algoritma. Za zapisivanje ove operaciju koristi se sledeća notacija:

$$r_{dest} \leftarrow f(r_{src1}, r_{src2}, ..., r_{srcn})$$

Registar s leve strane znaka ″←″, r_{dest}, je odredišni registar, dok su registri s desne strane, r_{src1}, r_{src2}, ..., r_{src}n, izvorni registri naredbe registarskog prenosa. (Tačnije, r_{src1}, r_{src2}, ..., r_{src}n označavaju izlaze, tj. sadržaje odgovarajućih izvornih registara). Funkcija *f* predstavlja operaciju koju treba izvršiti. Ova funkcija manipuliše sadržajima izvornih registara i eventualno podacima koji potiču sa spoljnih ulazima u sistem, a njen rezultat se upisuje u r_{dest} pod dejstvom rastuće ivice taktnog signala. Funkcija *f* može biti bilo koja funkcija koja se može realizovati u hardveru u vidu kombinacione mreže. Zapazimo da notacija ″←″ ne postoji u VHDL-u, ali će biti korišćena u ASM dijagramima za označavanje naredbi registarskog prenosa.

Pr. 9-1 Tipične naredbe registarskog prenosa

Slede primeri nekoliko tipičnih naredbi registarskog prenosa:

Naredba	Opis
r ← 1	U registar *r* se upisuje konstanta 1.
r ← r	U registar *r* se upisuje njegov tekući sadržaj. Sadržaj registra *r* ostaje nepromenjen.
r ← r << 2	Sadržaj registra *r* se pomera za 2 bitske pozicije ulevo, a zatim upisuje u isti registar
r0 ← r1	Sadržaj registra *r*1 se upisuje (prenosi) u registar *r*0.
n ← n + 1	Sadržaj registra *n* se uvećava za 1 i rezultat se upisuje u isti registar.
s ← $a^2 + b^2$	Zbir kvadrata sadržaja registra *a* i *b* se smešta u registar *s*.

Osnovna razlika između promenljivih iz algoritma i registra je u tome što se upis u registar inicira signalom takta, dok se upis u promenljivu obavlja istog momenta kada je njena nova vrednost izračunata. U hardveru, naredba registarskog prenosa $r_{dest} \leftarrow f(r_{src1}, r_{src2}, ..., r_{srcn})$ se izvršava na sledeći način:

1. U trenutku rastuće ivice taktnog signala, na izlazima izvornih registara, r_{src1}, r_{src2}, ..., r_{src}n, dostupne su nove vrednosti.
2. Kombinaciona mreža koja realizuje funkciju *f* izračunava novi rezultat, koji se s njenog izlaza prenosi do ulaza u odredišni registar r_{dest}. (Pretpostavka je da propagaciono kašnjenje kombinacione mreže ne traje duže od taktnog perioda.)
3. U trenutku prve sledeće rastuće ivice taktnog signala, rezultat funkcije *f* se upisuje u registar r_{dest}.

Hardverska realizacija pojedinačne naredbe registarskog prenosa je gotovo trivijalna. Potrebno je konstruisati funkciju *f* korišćenjem kombinacionih komponenti, a zatim povezati njene ulaze sa izlazima izvornih registara i njen izlaz sa ulazom odredišnog registar.

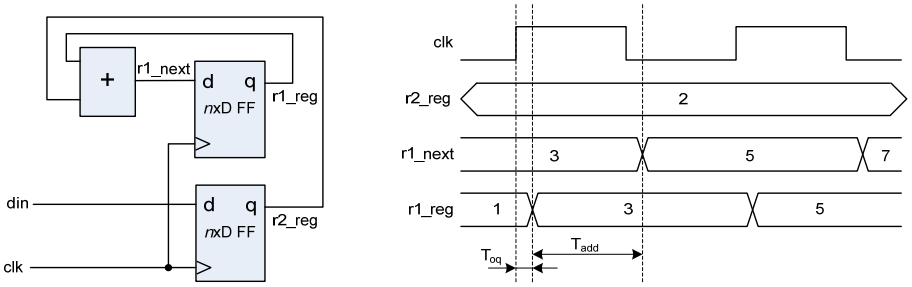
Pr. 9-2 Realizacija naredbe registarskog prenosa

Na Sl. 9-2(a) je prikazan blok dijagram hardverske realizacije naredbe registarskog prenosa *r*1 ← *r*1 + *r*2, a na Sl. 9-2(b) vremenski dijagrami koji ilustruju rada ovog kola. Promenljivama *r*1 i *r*2 odgovaraju registri *r1_reg* i *r2_reg*. Prva rastuća ivica takta upisuje novu vrednost u registar *r2_reg*, koja se neposredno posle toga pojavljuje na izlazu ovog registra. U tom trenutku započinje izračunavanje *r*1 + *r*2, a rezultat je dostupan na izlazu

sabirača (signal *r1_next*) nakon kašnjenja T_{add}. Iako je rezultat izračunat i prisutan na ulazu registra *r1_reg*, sadržaj ovog registra ostaje nepromenjen sve do sledeće rastuće ivice taktnog signala. Takođe, treba zapaziti da kolo sa Sl. 9-2(a) u svakom taktnom ciklusu obalja ne samo jednu, već dve naredbe registarskog prenosa:

$r1 \leftarrow r1 + r2$, i

$r2 \leftarrow din$



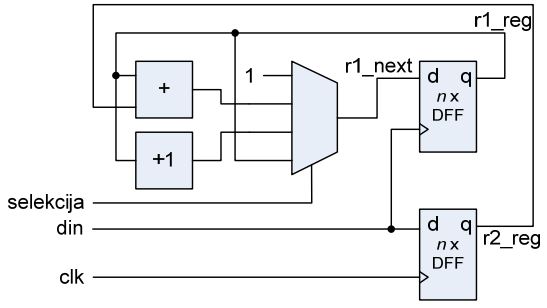
Sl. 9-2 Realizacija naredbe registarskog prenosa $r1 \leftarrow r1 + r2$: (a) blok dijagram; (b) vremenski dijagram. Napomena: T_{add} – propagaciono kašnjenje sabirača; T_{oq} – kašnjenje flip-flopa.

9.1.2. Staza podataka

Svaki algoritam se sastoji iz više koraka, a u neku promenljivu, odnosno odredišni registar, ne mora uvek da se upisuje isti podatak u svakom koraku. Na primer, u prvom koraku nekog algoritma, u registar *r*1 se može upisati vrednost 1; u sledeća dva koraka *r*1 se može najpre sabrati sa *r*2, a zatim uvećati za 1 i konačno u četvrtom koraku, sadržaj registra *r*1 može ostati neizmenjen:

1. $r1 \leftarrow 1$
2. $r1 \leftarrow r1 + r2$
3. $r1 \leftarrow r1 + 1$
4. $r1 \leftarrow r1$

Uočavamo da se *r*1 koristi kao odredišni registar u sve četiri naredbe registarskog prenosa. To znači da će prilikom hardverske realizacije biti neophodan multiplekser za izbor jednog od četiri podatka koji će u datom koraku biti upisan u *r*1. Na Sl. 9-3 je prikazan odgovarajući blok dijagram. Uočimo da se izbor jedne od četiri operacije vrši posredstvom selekcionog signala multipleksera.



Sl. 9-3 Realizacija četiri naredbe registarskog prenosa sa istim odredišnim registrom.

Kao što u tipičnom algoritmu ne postoji samo jedna već više primenljivih, tako i u tipičnom RTL sistemu ne postoji samo jedan, već više odredišnih registara, Ukoliko opisanu proceduru ponovimo za svaki odredišni registar, kreiraćemo strukturu koja će biti u mogućnosti da obavlja sve operacije (naredbe registarskog prenosa) sadržane u polaznom algoritmu. Takva jedna hardverska struktura se naziva *stazom podataka*.

Pr. 9-3 Paralelno izvršenje operacija

Za razliku od softvera, gde se operacije izvršavaju jedna po jedna, u hardveru može biti izvršena jedna ili **više** operacija u svakom taktnom ciklusu. Na primer, razmotrimo algoritam koji koristi dve promenljive, *x* i *y*, i sastoji se iz 5 sukcesivnih koraka u kojima se redom obavljaju sledeće operacije:

```
x = A;
y = B;
y = x + y;
y = y + 1;
x = x + 1;
```

Najpre se u promenljive *x* i *y* upisuju ulazni podaci, *A* i *B*, zatim se *y* prvo sabira sa *x*, a onda i uvećava za 1. Konačno, u poslednjem koraku, *x* se uvećava za 1. Za hardversku realizaciju opisanog postupka izračunavanja potrebna su dva registra: *rx*, za čuvanje vrednosti promenljive *x*, i *ry*, za promenljivu *y*. Svaka operacija iz pseudo kôda postaje naredba registarskog prenosa, koja se izvršava tokom jednog taktnog ciklusa:

```
ciklus 1: rx ← A;
ciklus 2: ry ← B;
ciklus 3: ry ← rx + ry;
ciklus 4: ry ← ry + 1;
ciklus 5: rx ← rx + 1;
```

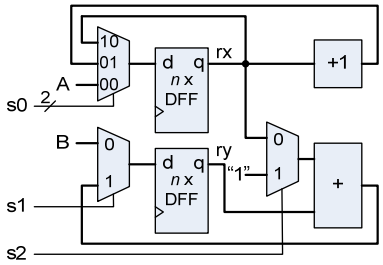
Primetimo da su naredbe koje su predviđene za izvršenje u prva dva taktna ciklusa međusobno nezavisne i da ne postoji razlog (osim mehaničkog prepisivanja pseudo kôda) zašto ne bi mogle biti izvršene u istom taktnom ciklusu. To isto važi i za naredbe iz 4. i 5. ciklusa. Ove dve naredbe su nezavisne zato što ni jedna od njih ne koristi rezultat one druge kao svoj ulazni argument. Nakon jedne ovakve analiza dolazimo do sledeće preraspodele operacija po taktnim ciklusima:

```
ciklus 1: rx ← A; ry ← B;
ciklus 2: ry ← rx + ry;
ciklus 3: ry ← ry + 1; rx ← rx + 1;
```

Dakle, u prvom taktnom ciklusu, u registre *rx* i *ry* se **istovremeno** upisuju ulazne vrednosti. Drugi taktni ciklus je posvećen sabiranju sadržaja dva registra, dok se u trećem, sadržaji registra *rx* i *ry* inkrementiranju (ponovo, **istovremeno**). Zahvaljujući uvedenom paralelizmu, broj taktnih ciklusa je smanjen sa 5 na 3.

Za obavljanje operacija sadržanih u razmatranom algoritmu potrebni su: jedan sabirač (za operaciju ″+″) i jedan inkrementer (za operaciju ″+1″). Primetimo da je dovoljan samo jedan inkrementer bez obzira na dve operacije inkrementiranja u 3. ciklusu. Sabirač koji se u 2. ciklusu koristi za sabiranje *rx* i *ry*, u 3. ciklusu je slobodan i može se koristiti za inkrementiranje bilo *rx* bilo *ry* (usvojićemo da se u 3. ciklusu sabirač koristi za *ry*+1). Osim registara i kombinacionih jedinica, za konstrukciju staze podataka takođe su neophodni multiplekseri, pomoću kojih će komponente staze podataka biti međusobno povezane. Na

primer, u 1. ciklusu, u registar *rx* se u upisuje *A*, u 2. ciklusu njegova sopstvena vrednost, a u 3. ciklusu vrednost sa izlaza inkrementera. Slično, u registar *ry* se u tri ciklusa upisuju vrednosti iz dva različita izvora: u 1. ciklusu to je konstanta 0, a u 2. i 3. ciklusu vrednost sa izlaza sabirača. Takođe, biće potreban i multiplekser za izbor jednog od operanda sabirača budući da se on koristi za dve različite operacije, *rx*+*ry* i *ry*+1. Imajući u vidu jednu ovakvu analizu, dolazimo do strukture staze podataka sa Sl. 9-4(a). Operacije koje će biti izvršene u stazi podataka određene su vrednostima selekcionih signala *s*0, *s*1 i *s*2, na način kako je to navedeno u tabeli sa Sl. 9-4(b).



(a)

Ciklus	Operacije	s0	s1	s2
1.	rx ← A ry ← B	″00″	0	x
2.	ry ← rx + ry	″10″	1	0
3.	ry ← ry + 1 rx ← rx + 1	″01″	1	1

(b)

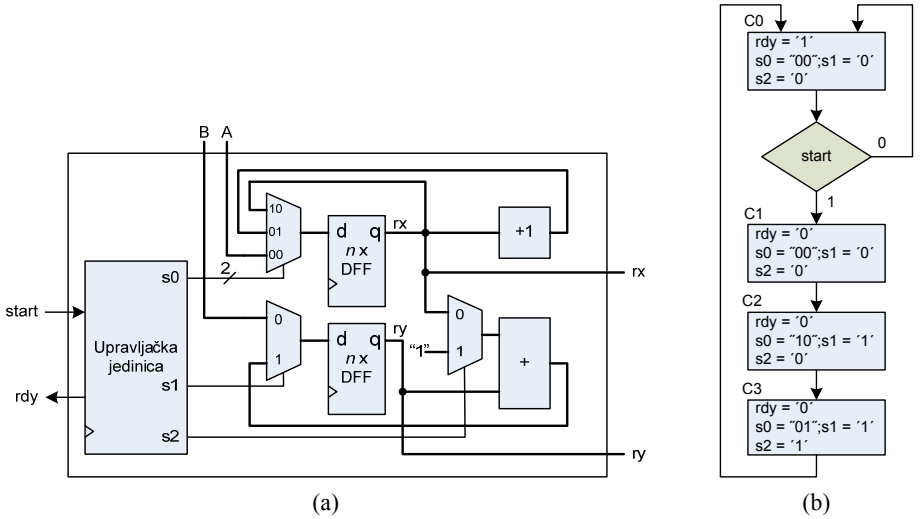
Sl. 9-4 Staza podataka iz Pr. 9-3: (a) struktura staze podataka; (b) tabela operacija.

9.1.3. Upravljačka jedinica

Da bi RTL sistem bio kompletan, osim staze podataka koja realizuje sve operacije sadržane u algoritmu, neophodan je i mehanizam koji će upravljati radom stazom podataka tako što će birati kada i koju operaciju treba izvršiti. Takav jedan mehanizam se naziva *upravljačkom jedinicom*, koja shodno svom internom statusu selektivno inicira izvršenje operacija u stazi podataka. Upravljačka jedinica se realizuje u vidu namenski projektovanog konačnog automata. Svakom stanju konačnog automata se pridružuju operacije koje u datom taktnom ciklusu treba izvršiti u stazi podataka. Na taj način, prolaskom kroz stanja, konačni automat može da nametne željenu sekvencu akcija. Nakon ispitivanja ulaza, konačni automat može nastaviti dalje različitim putanjama i tako izmeniti sekvencu akcija, što se koristi za realizaciju grananja i petlji iz algoritma.

Pr. 9-4 Upravljačka jedinica za stazu podataka iz Pr. 9-3

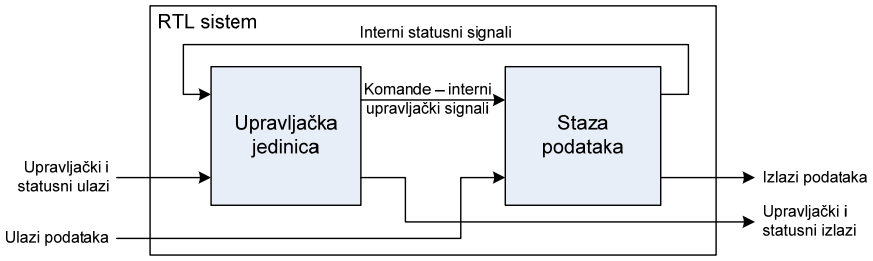
Staza podataka sa Sl. 9-4(a) iako u mogućnosti da obavi sve operacije sadržane u algoritmu iz Pr. 9-3, ipak nije u sanju da samostalno sprovede celokupno izračunavanje. Da bi ovaj RTL sistem bio kompletan, stazu podataka je neophodno proširiti upravljačkom jedinicom, kao na Sl. 9-5(a). Zadatak upravljačke jedinice je da u tri uzastopna taktna ciklusa postavlja selekcione signale staze podataka shodno tabeli operacija sa Sl. 9-4(b). Dodatno, upravljačka jedinica treba da obezbedi i način za interakciju korisnika i sistema. Iz tog razloga, uvedena su dva nova signala, ulazni upravljački signal *start* i izlazni statusni signal *rdy*. Od korisnika se očekuje da nakon što postavi vrednosti koje želi da procesira na ulaze *A* i *B*, aktivira signal *start* i na taj način naloži sistemu da počne s radom. Na izlazu *rdy* je prisutna ´0´ sve dok traje izračunavanje. Onda kad sistem završi s radom izlaz *rdy* ponovo dobija vrednost ´1´, što predstavlja indikaciju korisniku da sa izlaza podataka, *rx* i *ry*, može da preuzme rezultat. ASM dijagram koji opisuje rad upravljačke jedinice prikazan je na Sl. 9-5(b). U stanju C0 sistem čeka da bude aktiviran, dok stanja C1, C2 i C3 odgovaraju trima algoritamskim koracima.



Sl. 9-5 RTL sistem koji realizuje algoritam iz Pr. 9-3: (a) blok dijagram sistema; (b) ASM dijagram upravljačke jedinice.

9.1.4. Blok dijagram RTL sistema

Na Sl. 9-6 prikazan je uopšteni blok dijagram RTL sistema. Kao što se može videti, sistem je podeljen na dva podsistema, upravljačku jedinicu i stazu podataka. Staza podataka sadrži: (a) registre neophodne za čuvanje vrednosti promenljivih koje se javljaju u naredbama registarskog prenosa; i (b) kombinacionu logiku neophodnu za obavljanje izračunavanja specificiranih naredbama registarskog prenosa. Iako sadrži sve što je neophodno za izračunavanje i memorisanje vrednosti promenljivih, staza podataka nije u stanju da samostalno sprovede sekvencu operacija na način kako je to predviđeno algoritmom. To je razlog postojanja upravljačke jedinice, čiji je zadatak da u svakom taktnom ciklusu ¨naloži¨ stazi podataka šta da uradi. Drugim rečima, u upravljačkoj jedinici je ugrađeno ponašanje sistema, dok se resursi neophodni za obavljanje svih potrebnih izračunavanja i memorisanje podataka nalaze u stazi podataka. Upravljačka jedinica izdaje komande stazi podataka tako što postavlja interne upravljačke signale. Staza podataka obavlja aktivnosti koje su naložene komandom i kao odgovor generiše interne statusne signale. Upravljačka jedinica ispituje statusne signale koje dobija od staze podataka i na bazi njihove vrednosti i vrednosti upravljačkih i statusnih ulaza, donosi odluke koje određuju dalji tok rada sistema.

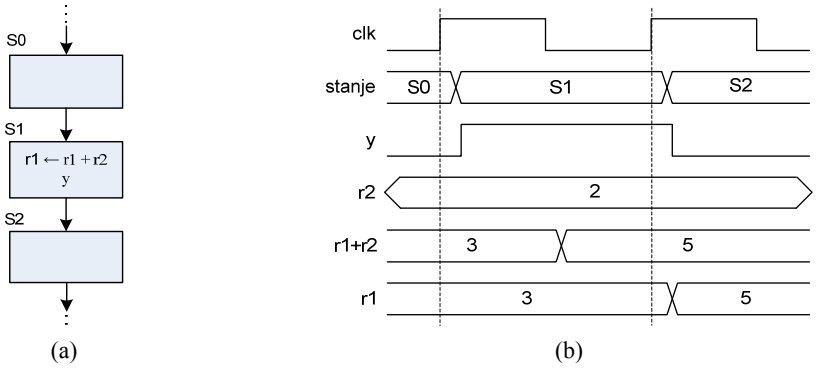


Sl. 9-6 Struktura RTL sistema.

9.1.5. ASMD dijagram

Konačni automat upravljačke jedinice se može lako proširiti na način da opisuje ne samo rad upravljačke jedinice već rad celokupnog sistema (upravljačka jedinica + staza podataka). Dovoljno je da se u svakom stanju automata naznače naredbe registarskog prenosa koje se u stazi podataka izvršavaju u tom stanju. Radi preglednijeg prikaza, za ovu namenu se koristi ASM dijagram (a ne dijagram stanja), a jedna takva predstava se naziva ASMD dijagramom (engl. ***ASM*** *with* ***D****atapath*).

Na Sl. 9-7(a) je prikazan jedan segment ASMD dijagrama. U bloku stanja S1 navedene su dve linije. Prva, $r1 \leftarrow r1 + r2$, ukazuje na naredbu registarskog prenosa koja se tokom stanja S1 izvršava u stazi podataka, dok druga, *y*, ukazuje da je u ovom stanju aktivan istoimeni izlazni signal upravljačke jedinice. Iako naveden kao odredišni registar u naredbi registarskog prenosa, *r*1 se ne menja u stanju S1, već zadržava vrednost koju je imao u momentu ulaska u ovo stanje. Rezultat izračunavanje izraza $r1 + r2$ biće upisan u *r*1 u trenutku prelaska iz stanja S1 u stanje S2, a nova vrednost registra *r*1 postaće dostupna tek kad ASMD pređe u stanje S2!



(a) (b)

Sl. 9-7 Naredba registarskog prenosa u funkcionalnom ASM dijagramu: (a) segment ASM dijagrama; (b) vremenski dijagram.

Na Sl. 9-7(b) je prikazan odgovarajući vremenski dijagram. Zapazimo da izračunavanje $r1 + r2$ počinje odmah nakon što ASMD uđe u stanje S1, ali da izlaz odredišnog **registra** *r*1 ostaje nepromenjen sve do kraja ovog stanja. S druge strane, izlazni **signal** *y* je aktivan za sve vreme dok je sistem u stanju S1, baš kao da se radi o regularnom ASM dijagramu. Sa sledećom rastućom ivicom taktnog signala, sistem prelazi u stanje S2, a u registar *r*1 se upisuje nova vrednost. Uočimo da je izlaz *y* neaktivan u stanju S2.

RTL sistemi su sinhroni sistem, odnosno spadaju u kategoriju digitalnih sistema kod kojih se svi registri taktuju istim, sistemskim taktom. U sinhronom sistemu, u svakom taktu obavlja se upis u sve registre budući da se rad registra ne može zabraniti ili onemogućiti. U konkretnom primeru to znači da se *r*1 ažurira ne samo pri prelasku iz stanja S1 u stanje S2, već i prilikom svakog prelaska iz jednog u neko drugo stanje, tj. čak i onda kad to nije potrebno. Onda kada sadržaj registara treba da ostane nepromenjen, u registar se zapravo upisuje njegova sopstvena vrednost, npr. $r1 \leftarrow r1$. Međutim, radi preglednosti, ove operacije se ne navode u ASMD dijagramu. Drugim rečima, ako se registar *r* ne pojavljuje kao odredišni registar ni jedne naredbe registarskog prenosa iz nekog stanju, podrazumevaćemo da se u tom stanju obavlja operacija $r \leftarrow r$.

Premda ASMD dijagram nalikuje softverskom dijagramu toka, oblasti primene i interpretacija ova dva tipa dijagrama se razlikuju. Dok softverske dijagrame toka koriste programeri kao početni korak u procesu razvoja programa, čija je namena da omogući izvršenje datog algoritam na računaru, dotle ASMD dijagrame koriste projektanti digitalnog hardvera, kao početni korak u procesu realizacije datog algoritma u hardveru. U pogledu interpretacije, razlika između softverskog dijagrama toka i ASMD dijagrama ogleda se prvenstveno u tome kako protok vremena utiče na rad algoritma. Računar izvršava program instrukciju-po-instrukciju; svaka instrukcija se obavlja u toku jednog instrukcijskog ciklusa, pri čemu instrukcijski ciklusi mogu biti različitog trajanja. Iz tog razloga, softverski dijagram toka ne sadrži informaciju o tačnom iznos vremena koje je potrebno da bi se izvršio jedan algoritamski korak, već samo jednoznačno određuje tok (sekvencu) algoritamskih koraka. Programer je svestan činjenice da izvršenje programa na računaru zahteva neko konačno vreme, koje zavisi od broja i složenosti operacija koje treba izvršiti, ali da sam tok algoritma i konačan rezultat ne zavise od vremena izvršenja pojedinačnih operacija. S druge strane, rad RTL sistema je iniciran taktnim signalom, a u toku jednog taktnog perioda sistem može da obavi jednu ili **više** operacija. Naime, u radu RTL sistema ispoljava se *paralelizam*. Stoga, osnovna pretpostavka kod ASMD dijagrama jeste u sledećem: svaki algoritamski korak (tj. blok stanja) traje isto, fiksno vreme (jednako taktnom periodu), a sve operacije obuhvaćene jednim stanjem izvršavaju se istovremeno (u paraleli). Druga napomena se odnosi na interpretaciju naredbi registarskog prenosa. Uvek treba imati na umu da naredba registarskog prenosa ne definiše vrednost koju registar, naveden s leve strane znak ←, dobija u datom stanju, već vrednost koja će biti upisana u taj registar sledećom aktivnom ivicom taktnog signala, tj. u trenutku napuštanja datog stanja.

9.2. Postupak projektovanja

Uopšteno govoreći, postoje dva osnovna pristupa rešavanju problema iz oblasti projektovanja digitalnih sistema: ¨odozdo-naviše¨ (engl. *bottom-up*) i ¨odozgo-naniže¨ (engl. *top-down*). Projektant koji primenjuje pristup ¨odozdo-naviše¨ započinje rad rešavanjem nekog izdvojenog detalja celokupnog problema. Nakon toga, projektant prelazi na neki drugi detalj koji ne mora obavezno biti u vezi s prvim. Konačno, projektant dolazi u situaciju da nezavisno rešene delove problema spaja u konačno rešenje. Međutim, nezavisno rešavani delovi obično se ne uklapaju idealno jedni s drugim, naročito ako se radi o problemima većeg obima. To je posledica samog pristupa koji fokusiranjem na izdvojene detalje odvlači pažnju projektanta sa slike o celovitosti rešenja. Konačan rezultat je po pravilu takav da se najveći deo projektantskog vremena troši na uklapanje nezavisnih detalja, što često zahteva i prepravku već rešenih delova opšteg problema.

Projektant koji se pridržava pristupa ¨odozgo-naniže¨ započinje rad na problemu razradom globalnog plana, što podrazumeva sveobuhvatno sagledavanje problema, razradu strategije rešavanja problema, razlaganje problema na potprobleme manjeg obima i definisanje odnosa između pojedinih potproblema. Drugim rečima, projektant se trudi da polazni problem, koji je obično isuviše obiman da bi se rešio ¨u jednom dahu¨, razloži na veći broj jasno definisanih problema manjeg obima. Pri tom projektant ne rešava odmah uočene potprobleme, već ih tretira kao apstraktne sklopove (crne kutije) koje međusobno uklapa i povezuje. U drugoj fazi projektovanja, projektant nastavlja s razradom i rešavanjem uočenih potproblema. Pri tom projektant može slobodno da se usredsredi na svaki pojedinačni, prethodno definisani potproblem, bez brige oko njihovog uklapanja, zato što je

detalje koji se tiču spajanja podeljenih delova razradio još u prvoj fazi projekta i time ih ugradio u definiciju svakog pojedinačnog potproblema.

Projektanti-početnici obično ne doživljavaju projektovanje ¨odozgo-naniže¨ kao prirodan pristup. Prirodno je da neko ko nema iskustva u projektovanju hardvera bude zabrinut detaljima koji se odnose na realizaciju. Pristup ¨odozgo-naniže¨ zasnovan je na iskustvu i samopouzdanju projektanta, jer samo neko ko je ranije rešavao detalje slične onim koji se javljaju u tekućem problemu, može u toku razrade globalnog plana ignorisati te detalje siguran u to da će moći da ih reši onda kad pređe na realizaciju svog plana.

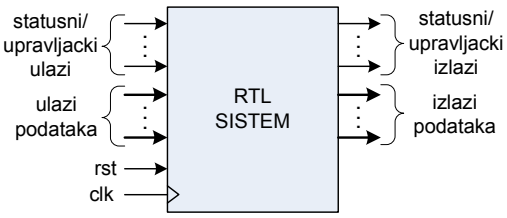
U kontekstu RTL projektovanja, pristup ″odozgo-naniže″ podrazumeva sledeće tri faze:

- Opis ponašanja
- Razrada
- Realizacija

9.2.1. Opis ponašanja

Opis ponašanja se smatra najvažnijom fazom celokupnog procesa projektovanja RTL sistema. U ovoj fazi, sistem se opisuje na algoritamskom nivou, npr. u vidu ASMD dijagrama. Blok dijagram koji specificira ulazne i izlazne portove sistema, tj. *interfejs* sistema, jedina je strukturna forma koja postoji u ovoj fazi projektovanja. Drugim rečima, u fazi opisa ponašanja, sistem se tretira kao *crna kutija* s definisanim ulazima i izlazima koja se ponaša na način kako je opisano pratećim algoritmom (odnosno ASMD dijagramom). Tipično, projektant polazi od algoritma koji je predstavljen u obliku pseudo kôda, a koji potom prevodi u ekvivalentan ASMD dijagram. Tokom ove konverzije, projektant, po pravilu, ne kreira samo jedan ASMD dijagram, već istražuje različite opcije i varijante mogućih rešenja, vrednujući ih u skladu s unapred postavljenim kriterijumima (brzina rada, hardverska složenost i sl.). Napomenimo da ASMD dijagram, iako opis visokog nivoa apstrakcije, pruža dovoljno informacija na osnovu kojih je moguće odrediti broj taktnih ciklusa potrebnih za obavljanje pojedinih aktivnosti i grubo proceniti složenost hardverske realizacije, npr., na osnovu broja registara i broja i tipova aritmetičkih i logičkih operacija.

Interfejs. Osim algoritma koji definiše funkciju sistema, opis ponašanja sadrži i specifikaciju interfejsa. Kao pojam, interfejs označava mesto na kome se dva nezavisna sistema susreću ili spajaju (ili sredstvo koje koriste) radi zajedničkog dejstva ili međusobne komunikacije i koordinacije. U toku rad, digitalni sistem interaguje sa okruženjem koje mogu da čine drugi digitalni sistemi, nedigitalni uređaji, kao što su prekidači ili senzori, pa čak i čovek koji posredstvom odgovarajućih ulazno-izlaznih uređaja komunicira sa sistemom. Rećićemo da se okruženje RTL sistema sastoji od nezavisnih *aktora*, koji interaguju kako sa digitalnim sistemom, tako i između sebe. S tačke gledišta projektanta RTL sistema, detalji koji se tiču rada i implementacije aktora nisu od značaja. Za projektanta je bitno da poznaje informacije koje se razmenjuju između sistema i aktora kao i način na koji aktori komuniciraju sa sistemom, kako bi mogao da ih ugradi u specifikaciju interfejsa. Za razliku od projektanta RTL sistema, programeri imaju na raspolaganju standardne, sofisticirane korisničke interfejse (tastatura, miš, monitor), koji olakšavaju korisnicima da interaguju sa programom. Korisnički interfejsi ovog tipa rešavaju dva osnovna problema koji se javljaju uvek kada dva aktora pokušavaju da komuniciraju: (a) koji podaci se razmenjuju u toj komunikaciji i (b) kada i kako se obavlja prenos informacije. Prilikom projektovanja hardvera javljaju se isti ovi problemi, ali sada je odgovornost na projektantu digitalnog sistema da ih reši.



Sl. 9-8 Interfejs RTL sistema.

Specifikacija interfejsa RTL sistema uključuje specifikaciju ulaza i izlaza (tj. portova) proširenu pravilima za korišćenje sistema od strane korisnika. U opštem slučaju, RTL sistem razmenjuje s aktorima iz okruženja dva tipa ulaznih i izlaznih informacija: (a) upravljački i statusni signali i (b) podaci (Sl. 9-8). Upravljački i statusni signali su najčešće jednobitni, dok su podaci višebitni signali. Aktor inicira neku akciju sistema posredstvom upravljačkih ulaza, dok preko ulaznih statusnih signala obaveštava sistem o svom trenutnom stanju. Putem statusnih ulaza sistem dobija odgovor na pitanja tipa ¨da-ne¨, kao što je npr.: ¨da li je prekidač zatvoren?¨ U ASMD dijagramu koji opisuje ponašanje sistema, imena statusnih i upravljačkih ulaza se javljaju samo unutar blokova grananja, tj. koriste se za donošenje direktnih odluka, ali ne i za neka druga izračunavanja. Slično, sistem koristi upravljačke izlaze da bi inicirao neku akciju aktora, a statusne da bi obavestio okruženje o svom trenutnom stanju (npr. ″izračunavanje je završeno″). Klasifikacija na upravljačke i statusne signale nije stroga, a projektant ima slobodu da klasifikaciju obavi na način koji smatra najprikladnijim u kontekstu konkretnog problema. Ulazi podataka igraju istu ulogu kao ulazne promenljive kod konvencionalnih programskih jezika, tj. koriste se za prosleđivanje sistemu informacije koju on treba da obradi. U ASMD dijagramu ulazi podataka se javljaju samo s desne strane naredbi registarskog prenosa i/ili uslova u blokovima grananja. Sistem koristi izlaze podataka da bi predao okruženju obrađenu informaciju (analogno izlaznim promenljivama iz programskih jezika). Vrednost koja se postavlja na izlaz podataka uvek potiče iz nekog internog registra. Dodatno, svaki sinhroni digitalni sistem poseduje još dva obavezna signala: signal takta, *clk*, i signal resetovanja, *rst*, kojim se sistem postavlja u početno stanje. Ova dva signala se obično ne crtaju, jer se smatraju podrazumevanim.

Pr. 9-5 Interfejs sekvencijalnog delitelja

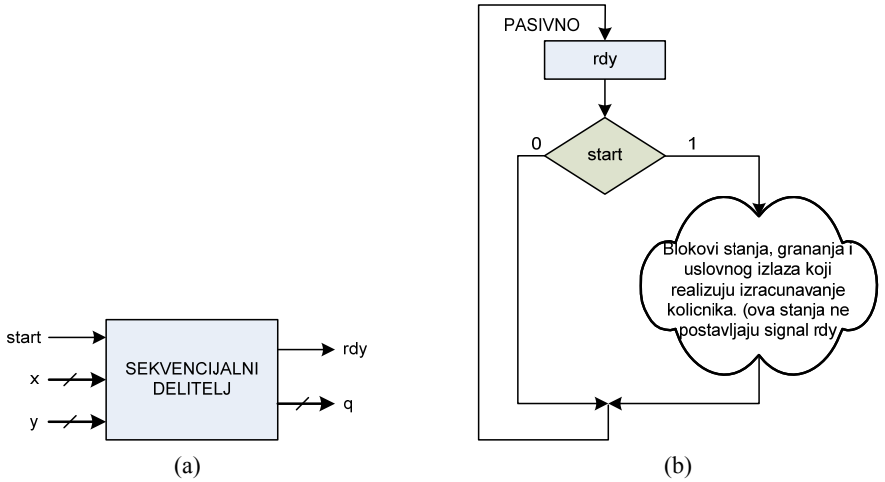
Na Sl. 9-9(a) je prikazan blok dijagram sekvencijalnog delitelja. Ovo kolo obavlja operaciju deljenja $q=x/y$, gde su x i y neoznačeni celi brojevi. Rad delitelja se startuje signalom *start*. Po završenom deljenju, sistem postavlja rezultat na izlaz podataka q i aktivira statusni izlaz *rdy*.

Osim ulaza i izlaza, opis interfejsa može da sadrži i neka dodatna ograničenja koja se odnose na interakciju sistema i korisnika. Na primer, mogu se postaviti sledeći zahtevi:

1. Pre nego što aktivira upravljački ulaza *start* (*start*=´1´), korisnik treba da postavi deljenik i delilac na ulazne portove podatak x i y. Korisnik ne sme da menja x i y za vreme dok traje izračunavanje (tj. dok je *rdy*=´0´).
2. Trajanje signala *start* mora biti tačno jedan taktni period.
3. Korisnik je u obavezi da pre iniciranja sledećeg izračunavanja čeka barem dva taktna ciklusa nakon što je sistem završio prethodno izračunavanje.

Globalni oblik ASMD dijagrama koji opisuje ponašanje korisničkog interfejsa

sekvencijalnog delitelja prikazan je na Sl. 9-9(b). Za sada, ASMD dijagram ima samo jedno stanje, PASIVNO, u kome čeka na aktiviranje signala *start* i pri tom drži na izlazu *rdy* vrednost ´1´. *Rdy*=´1´ predstavlja indikaciju okruženju da je sistem spreman za rad. U nastavku projektovanja, ¨oblak¨ iz ovog ASMD dijagramu biće zamenjen blokovima stanja, grananja i uslovnih izlaza, uz pomoć kojih će biti opisan tok algoritma za deljenje. Međutim, signal *rdy* neće figurisati ni u jednom od tih stanja. To znači da u prvom narednom taktnom periodu nakon što korisnik aktivira signal *start*, signal *rdy* dobija vrednost ´0´, a onda zadržava ovu vrednost za sve vreme izračunavanja količnika. Nakon što količnik bude izračunat i ASMD se vrati u stanje PASIVNO, signal *rdy* će ponovo dobiti vrednost ´1´.

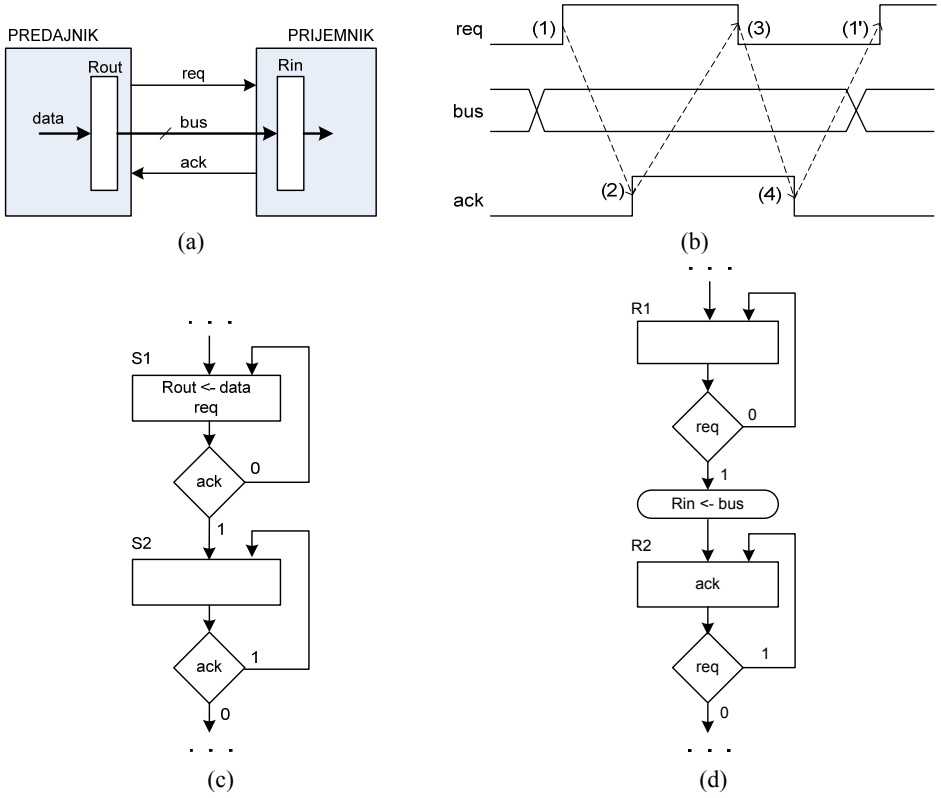


Sl. 9-9 Sekvencijalni delitelj: (a) blok dijagram; (b) ASM opis ponašanja interfejsa.

Kao što se može zaključiti iz prethodnog primera, specifikacija interfejsa uključuje specifikaciju ulaza i izlaza, proširenu pravilima za korišćenje sistema od strane korisnika. Može se reći da interfejs predstavlja neku vrstu *ugovora* između dva aktora kojeg se moraju pridržavati obe strane kako bi krajnji rezultat njihove interakcije bio korektan. ¨Prijateljski¨ korisnik je onaj koji se u radu sa sistemom pridržava svih navedenih ograničenja.

Pr. 9-6 *Handshake* interfejs

Handshake interfejs, kao pojam, ukazuje na princip koji se često koristi za prenos podataka između digitalnih sistema. *Handshake* (u prevodu, rukovanje) definiše način koordinacije dva sistema koji se ostvaruje posredstvom dva signala, *req* (od *request* - zahtev) i *ack* (od *acknowladge* - potvrda) (Sl. 9-10(a)). Jedan sistem igra ulogu predajnika, a drugi prijemnika. Aktivnosti predajnika i prijemnika tokom jednog ciklusa prenosa podatka prikazani su vremenskim dijagramom sa Sl. 9-10(b). Predajnik postavlja podatak koji želi da preda prijemniku na magistralu *bus* i aktivira signal *req* (1). S druge strane, prijemnik čeka na ovaj zahtev i po njegovom prijemu (*req*=´1´), prihvata podatak sa magistrale *bus* i posredstvom signala *ack* potvrđuje prijem (*ack*=´1´, (2)). Nakon što je primio potvrdu, predajnik deaktivira signal *req* (3), kako bi se pripremio za sledeći prenos (1'). Uslov *req*=´0´ je znak prijemniku da deaktivira signal potvrde, (*ack*=´0´, (4)) i nastavi da čeka na novi zahtev. Na Sl. 9-10(c) i (d) su prikazani delimični ASMD dijagrami predajnika i prijemnika, koji zajedno opisuju rad *handshake* interfejsa.



Sl. 9-10 *Handshake* interfejs: (a) blok dijagram; (b) vremenski dijagram; (c) ASMD dijagram predajnika; (d) ASMD dijagram prijemnika.

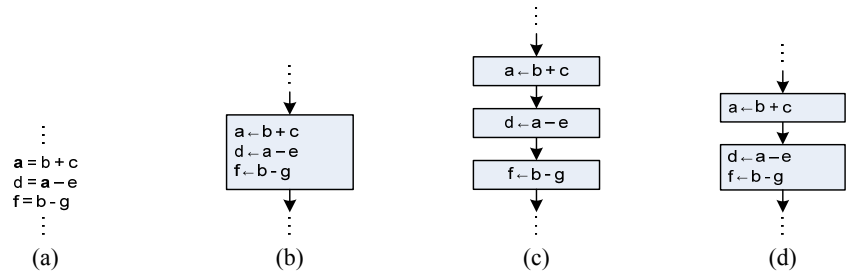
Konverzija algoritma u ASMD dijagram. Proces projektovanja RTL sistema obično počinje kreiranjem algoritma visokog nivoa. Algoritam se piše u obliku pseudo kôda ili dijagrama toka i kao takav zapravo predstavlja softversko rešenje datog problema. Osnovna pretpostavka kod softverskog algoritma je da se algoritamski koraci izvršavaju jedan za drugim, strogo sekvencijalno. Može se desiti da su dve ili više uzastopnih operacija međusobno nezavisne (u smislu da ni jedna od njih kao operand ne koristi rezultat bilo koje druge) i da bi usled toga mogle biti izvršavane istovremeno (paralelno). Međutim, pomoću softverskog algoritma nije moguće izraziti paralelizam. Zbog toga, softverski algoritam nije pogodno sredstvo za opis ponašanja RTL sistema.

RTL sistem se pobuđuje taktnim signalom i u mogućnosti je da izvrši jednu ili **više** elementarnih operacija u svakom taktnom periodu. Ovakav model izračunavanja, koji dozvoljava paralelizam i nameće diskretizaciju vremena, efikasno se predstavlja ASMD dijagramom. Kao što je već rečeno, ASMD dijagram definiše stanja RTL sistema i operacije koje se izvršavaju u svakom stanju. Svako stanje traje jedan taktni period, a sve operacije obuhvaćene jednim stanjem izvršavaju se u paraleli. Međutim, algoritam je obično teže predstaviti u obliku ASMD dijagrama nego u obliku softverskog dijagrama toka ili pseudo kôda. Iz tog razloga, opis ponašanja RTL sistema po pravilu počinje kreiranjem softverskog algoritma koji se potom prevodi u ekvivalentan ASMD dijagram.

Nažalost, direktno prevođenje softverskog algoritma u ASMD dijagram, tako što bi se sekvencijalni blokovi iz softverskog algoritma prosto zamenili blokovima stanja u ASMD dijagramu, nije korektno. Na taj način bi se u potpunost poništila sekvencijalnost u izvršenju naredbi, pa bi rezultat rada algoritma bio pogrešan u slučajevima kada u sekvencijalnom bloku postoje međusobno zavisne naredbe. Prilikom prevođenja, s jedne strane je **neophodno** rasporediti zavisne operacije u različita stanja, da bi konverzija bila korektna, dok je s druge strane **poželjno** spojiti nezavisne operacije u ista stanja, kako bi se kroz paralelizam postigle što bolje performanse. To nije uvek jednostavan zadatak, kao što će biti ilustrovano u primerima koji slede.

Pr. 9-7 Paralelizacija sekvencijalnog kôda

Na Sl. 9-11(a) je prikazan sekvencijalni blok iz nekog softverskog dijagrama toka. Blok sadrži tri sekvencijalne naredbe. Zapazimo da zavisnost postoji samo između prve i druge naredbe, dok je treća nezavisna od obe prethodne. Zbog postojanja zavisnosti između naredbi, zamena ovog sekvencijalnog bloka jednim blokom stanja nije korektna (Sl. 9-11(b)). Budući da se sve naredbe iz istog bloka stanja izvršavaju istovremeno, u drugoj naredbi se ne bi koristila vrednost promenljive *a* koja je izračunata u prvoj naredbi bloka, već vrednost koju je ova promenljiva imala u momentu ulaska u dato stanje. Rešenje gde se svaka naredba iz sekvencijalnog bloka raspoređuje u zasebno stanje ASMD dijagrama svakako je korektno (Sl. 9-11(c)), jer se naredbe izvršavaju jedna za drugom, po redosledu kao u sekvencijalnom bloku. Međutim, rešenje sa Sl. 9-11(c) je previše restriktivno, jer nepotrebno uvodi posebno stanje za izvršenje treće naredbe. Budući da treća naredba slobodno može biti izvršavana u isto vreme kad i druga, u mogućnosti smo da ″uštedimo″ na jednom stanju i da na taj način skratimo vreme rada sistema za jedan taktni ciklus (kao u rešenju sa Sl. 9-11(d)).



Sl. 9-11 Konverzija softverskog dijagrama toka u ASMD dijagram: (a) sekvencijalni blok naredbi; (b) ASMD – maksimalni paralelizam, neispravno rešenje; (c) sekvencijalni ASMD dijagram – minimalni paralelizam, ispravno rešenje; (d) ASMD – optimalno rešenje.

Početni korak u procesu konverzije algoritma u ASMD dijagram zasnovan je na sledeća dva jednostavna pravila:

1. Svaka naredba dodele iz softverskog algoritma prevodi se u naredbu registarskog prenosa koja se smešta u zaseban blok stanja ASMD dijagrama.
2. Svaka naredba grananja (npr. *if*, *while*) iz softverskog algoritma se prevodi u prazan blok stanja sa pridruženim blokom grananja u koji je upisana relacija iz softverske naredbe grananja.

Promenljive iz softverskog algoritma tretiraju se kao registri, a naredbe dodele kao naredbe registarskog prenosa. Pravila 1 i 2 su neophodna da bi se: (a) očuvao sekvencijalni tok izvršenja algoritma (u svakom taktnom ciklusu izvršava se samo jedna naredbe.) (b)

obezbedilo da se u bloku grananja neće koristiti promenljiva čija se vrednost modifikuje u roditeljskom bloku stanja. (Podsetimo da se blok stanja zajedno sa pridruženim grananjima izvršava u jednom taktnom periodu, a da se upis novih vrednosti u registre obavlja tek na kraju taktnog perioda).

Pr. 9-8 ASMD dijagram sekvencijalnog delitelja

U cilju ilustracije postupka konverzije softverskog algoritma u ASMD dijagram, razmotrićemo hardversku realizaciju sekvencijalnog delitelja neoznačenih celih brojeva. Blok dijagram i interfejs delitelja su specificirani u Pr. 9-5. Premda za ovu namenu postoji veći broj efikasnih i brzih algoritama, u ovom primeru biće korišćen trivijalan algoritam deljenja, koji se zasniva na uzastopnom oduzimanju delioca od deljenika. Rad ovog algoritma može se predstaviti u vidu sledećeg pseudo kôda:

```
r1 = x;
r2 = 0;
while (r1 ≥ y)
  {r1 = r1 – y;
   r2 = r2 + 1;}
```

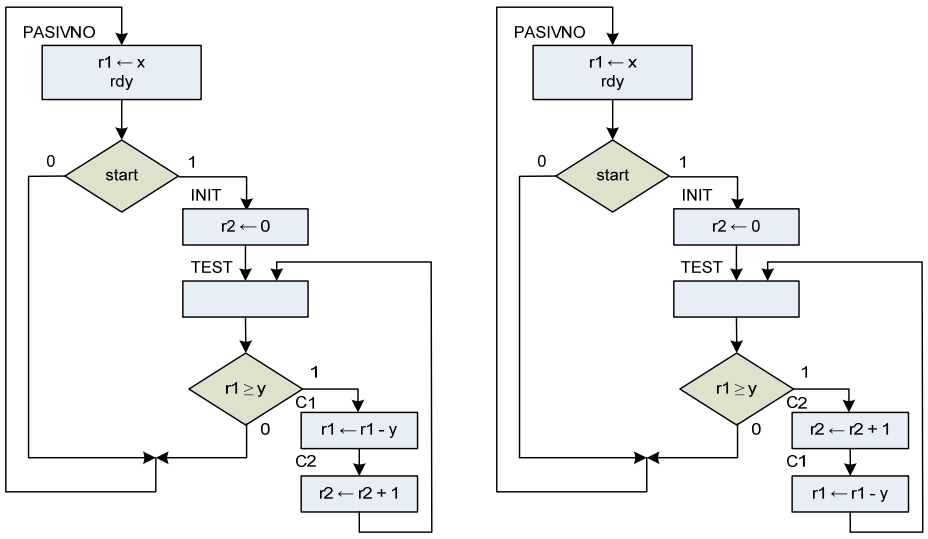
Lako se uočava da nakon izlaska iz *while* petlje, promenljiva *r*2 ima vrednost *x*/*y*. Očigledno, radi se o sporom algoritmu, s obzirom na to što se petlja mora izvršiti *x*/*y* puta. Uočimo da bi algoritam s promenjenim redosledom naredbi unutar *while* petlje takođe bio korektan:

```
r1 = x;
r2 = 0;
while (r1 ≥ y)
  {r2 = r2 + 1;
   r1 = r1 – y;}
```

Primenom pravila konverzije, dve prethodne varijante softverskog algoritma za deljenje lako se prevode u ASMD dijagrame koji su prikazani na Sl. 9-12. Jedina razlika između ova dva ASMD dijagrama je u redosledu stanja C1 i C2 u petlji koja oduzima delilac od deljenika i računa količnik. Sa stanovišta algoritma deljenja redosled ove dve operacije je nebitan, što znači da su oba ASMD dijagrama korektna.

Pregledom dva ASMD dijagrama može se zapaziti da posle stanja PASIVNO sledi blok grananja koji ispituje upravljački ulaz *start*, što na prvi pogled nije u skladu s drugim pravilom prevođenja softverskog algoritma u ASMD dijagram. Međutim, testiranje signal *start* u stanju PASIVNO ne potiče iz softverskog algoritma, već je deo specifikacije interfejsa, pa time ne narušava softverske zavisnosti između naredbi registarskog prenosa.

Razmotrimo sada s više detalja rad ASMD dijagrama sa Sl. 9-12(a). U stanju PASIVNO, u registar *r*1 se upisuje vrednost deljenik *x*. Ovaj upis se ne dešava samo jedanput, već se ponavlja u svakom taktnom ciklusu za sve vreme dok je sistem u stanju PASIVNO. Onda kad korisnik postavi *start*=´1´, ASMD iz stanja PASIVNO prelazi u stanje INIT, gde će u registar *r*2 biti upisana inicijalna vrednost količnika, tj. 0. Kako se u stanju PASIVNO vrednost registra *r*2 ne menja, a pri tom *r*2 sadrži rezultat poslednjeg deljenja, ovako koncipiran interfejs pruža korisniku onoliko vremena koliko mu je potrebno da preuzme rezultat. Zapazimo da interfejs, a ne algoritam deljenja, zahteva da se *r*2 inicijalizuje u posebnom stanju, INIT. Naime, sa stanovišta algoritma deljenja, inicijalizacija registra *r*2 u stanju PASIVNO bila bi korektna, a pri tom bi i ukupan broj stanja u ASMD dijagramu bio za jedan manji. Međutim, kod takvog rešenja, rezultat deljenja bi bio dostupan za preuzimanje s izlaza registra *r*2 samo tokom jednog taktnog ciklusa.



Sl. 9-12 ASMD dijagrami sekvencijalnog delitelja: (a) varijanta 1, (b) varijanta 2.

Deo ASMD dijagrama koji izračunava količnik je petlja koju čine stanja TEST, C1 i C2. U stanju TEST (tj. u bloku grananja koji je pridružen ovom stanju) ispituje se da li je $r1$ jednako ili veće od y, a to je isto ono ispitivanje koje postoji u *while* naredbi softverskog algoritma. Stanja C1 i C2, u uzastopnim taktnim ciklusima, realizuju, u vidu naredbi registarskog prenosa, naredbe dodele koje čine telo *while* petlje. Obe varijante ASMD dijagrama, rade ispravno i onda kada je $x < y$. Za $x < y$, uslov $r1 \geq y$ iz bloka grananja u stanju TEST je netačan i ASMD se vraća u stanje PASIVNO sa nulom u registru $r2$ (tj. $q=0$).

Pažljivom analizom ASMD dijagrama sa Sl. 9-12(b), može se uočiti razlog zašto je stanje TEST neophodno. Ovo stanje, osim ispitivanja uslova $r1 \geq y$, koji mu je pridružen, ne sadrži nikakvu dodatnu aktivnost. U ovoj varijanti ASMD dijagrama, u stanju C1, koje prethodi stanju TEST, izvršava se naredba registarskog prenosa $r1 \leftarrow r1 - y$. Tačnije, tokom stanja C1 obavlja se operacija oduzimanja, $r1 - y$, dok se upis razlike u $r1$ dešava tek u momentu prelasku u stanje TEST. Izostavljanjem stanja TEST, ispitivanje $r1 \geq y$ bi postalo deo stanja C1, pa bi se za $r1$ koristila stara, a ne nova vrednost, što bi svakako dovelo do poremećaja u radu algoritma.

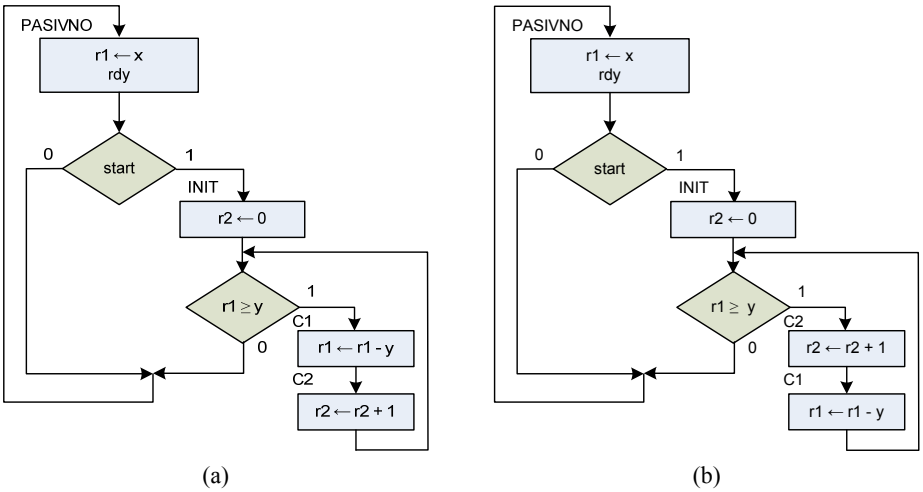
ASMD dijagrami sa Sl. 9-12 izračunavaju količnik za $4+3q$ taktnih ciklusa, gde je $q=x/y$. Konstantni član 4 predstavlja zbir: dva taktna ciklusa koja je korisnik u obavezi da provede u stanju PASIVNO pre nego što ponovo aktivira signal *start* (zahtev koji potiče iz specifikacije korisničkog interfejsa – v. Pr. 9-5), jednog taktnog ciklusa za stanje INIT i još jednog taktnog ciklusa za stanje TEST u poslednjoj iteraciji petlje. Faktor 3 potiče od tri stanja po jednoj iteraciji petlje.

Optimizacija ASMD dijagrama. Prethodni primer pokazuje da se softverski algoritam uvek može prevesti u korektan ASMD dijagram primenom jednostavnih pravila. Međutim, za isti polazni algoritam često postoji više varijanti ASMD dijagrama, pri čemu ona koja se dobija direktnim prevođenjem ne mora uvek biti i najbolja. Naime, ASMD dijagrami dobijeni direktnim prevođenjem, premda garantovano ispravni, ne moraju biti i

najefikasniji mogući u pogledu broja stanja i broja registara. Glavni uzrok neefikasnosti krije se u činjenici da su u ovako dobijene ASMD dijagrame prenesene sve zavisnosti između naredbi softverskog programa koje, forsirajući strogo sekvencijalan redosled izvršenja naredbi registarskog prenosa, onemogućavaju ispoljavanje paralelizma u radu različitih delova digitalnog sistema. Zato, ovako dobijene ASMD dijagrame treba koristiti samo kao polaznu osnovu za detaljniju analizu koja bi imala za cilj pronalaženje optimalne varijante ASMD dijagrama, što će biti tema nekoliko narednih primera.

Pr. 9-9 Optimizacija sekvencijalnog delitelja - eliminacija stanja TEST

Stanje TEST je uvedeno u ASMD dijagram sekvencijalnog delitelja samo iz razloga mehaničkog (direktnog) prevođenja softverskog algoritma u ASMD. U mnogim slučajevima, ispitivanje kao što je ono iz stanja TEST može se pridružiti nekom drugim stanju, a da to ne nariši korektnost algoritma. Na taj način, eliminiše se jedno stanje, što ubrzava rad sistema. Podsetimo se da blok grananja koji sledi neposredno posle bloka stanja koji nije prazan, znači da se ispitivanje u bloku grananja i izračunavanja u bloku stanja izvršavaju u paraleli. Iz tog razloga, nije korektno spojiti grananje sa stanjem u kome se obavlja izračunavanje čiji se rezultat koristi u uslovu grananja. Razmotrimo modifikovane verzije ASMD dijagrama sa Sl. 9-12 u kojima je stanje TEST, prosto, obrisano.



Sl. 9-13 ASMD dijagrami sekvencijalnog delitelja sa izbačenim stanjem TEST: (a) ispravno rešenje; (b) neispravno rešenje.

Bez obzira na to što ASMD dijagram sa Sl. 9-13(b) ispravno radi za $x<y$ (jer ne ulazi u petlju), greška u izračunavanju količnika se javlja pri $x\geq y$. Kao ilustraciju pogrešnog rada, razmotrimo rad ovog ASMD dijagrama za x=14 i y=7, pod pretpostavkom da je registar $r1$ 12-bitni (T. 9-1).

U ASMD dijagramu sa Sl. 9-13(b), uslov $r1\geq y$ se ispituje dva stanja: INIT i C1. Izračunavanje obuhvaćeno stanjem INIT modifikuje jedino registar $r2$, što nema uticaja na ispitivanje $r1\geq y$. Međutim, problem postoji u stanju C1, jer izračunavanje obuhvaćeno ovim stanjem modifikuje registar $r1$, a odlučivanje koje sledi zasnovano je upravo na vrednosti registra $r1$. Prilikom drugog prolaska kroz stanje C1, koji bi trebalo da je poslednji budući da je očekivana vrednost količnika 2, $r1$ još uvek sadrži vrednost 7 (vrsta

u tabeli T. 9-1 prikazana masnim slovima). Naredba registarskog prenosa $r1 \leftarrow r1 - y$ nalaže promenu vrednosti u $r1$ na 0, ali to će se desiti tek prilikom prelaska u sledeće stanje. S obzirom na to što ispitivanje $r1 \geq y$ koristi tekuću vrednost registra $r1$ (tj. 7), petlja će se ponoviti još jedanput, što daje pogrešan rezultat ($r2=3$). Vrednost 4089, koju dobija registar $r1$ prilikom trećeg prolaska kroz petlju, posledica je potkoračenja prilikom izvođenja operacije $r1 - y$ ($4089+7 = 2^{12}$).

T. 9-1 Ilustracija rada ASMD dijagrama sa Sl. 9-13(b).

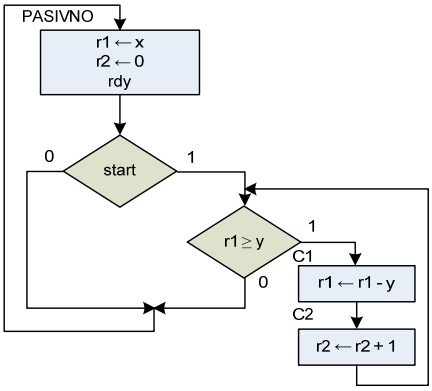
Tekuće stanje				
PASIVNO	r1= ?	r2= ?	start=0	rdy=1
PASIVNO	r1= 14	r2= ?	start=1	rdy=1
INIT	r1= 14	r2= ?	start=0	rdy=0
C2	r1= 14	r2= 0	start=0	rdy=0
C1	r1= 14	r2= 1	start=0	rdy=0
C2	r1= 7	r2= 1	start=0	rdy=0
C1	**r1= 7**	**r2= 2**	**start=0**	**rdy=0**
C2	r1= 0	r2= 2	start=0	rdy=0
C1	r1= 0	r2= 3	start=0	rdy=0
PASIVNO	r1= 4089	r2= 3	start=0	rdy=1
PASIVNO	r1= ?	r2= 3	start=0	rdy=1

Premda je eliminacijom stanja TEST iz ASMD dijagrama sa Sl. 9-12(b) dobijeno pogrešno rešenje, postavlja se pitanje šta će se desiti ako isto ovo stanje odstranimo iz ASMD dijagrama sa Sl. 9-12(a), gde se stanje C1 nalazi na **početku** petlje (Sl. 9-13(a)). U ASMD dijagramu sa Sl. 9-13(a), ispitivanje $r1 \geq y$ se javlja takođe u dva različita stanja, INIT i C2. Razlika u odnosu na ASMD sa Sl. 9-13(b) je u tome što sada izračunavanje iz stanja C2, $r2 \leftarrow r2 + 1$, ne utiče na ishod ispitivanja $r1 \geq y$. Iz tog razloga, ASMD dijagram sa Sl. 9-13(a) je ispravan. Pošto petlja sada obuhvata dva, a ne više tri stanja (kao u ASMD dijagramu iz Pr. 9-8), ukupno vreme izračunavanja količnika je smanjeno i iznosi: $3+2q$ taktnih ciklusa, gde je $q=x/y$. Naglasimo da je ostvareno ubrzanje posledica preklapanja izvršenja dve nezavisne operacije, naredbe registarskog prenosa $r2 \leftarrow r2 + 1$ i ispitivanja uslova $r1 \geq y$.

Pr. 9-10 Optimizacija sekvencijalnog delitelja - eliminacija stanja INIT

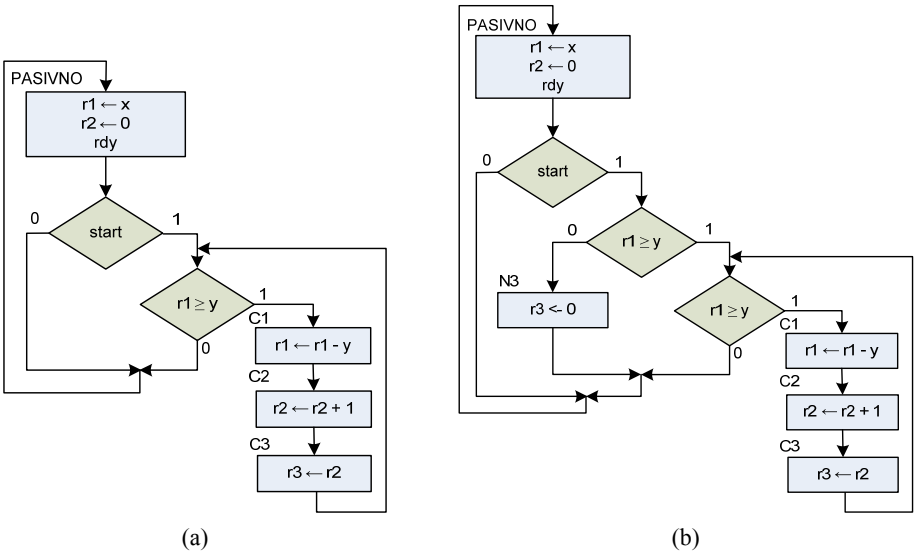
Stanje INIT se koristi za inicijalni upis vrednosti 0 u registar $r2$. Razmotrićemo mogućnost eliminacije ovog stanja kroz pripajanje operacije $r2 \leftarrow 0$ stanju PASIVNO. Prebacivanjem ove naredbe registarskog prenosa iz stanja INIT u stanje PASIVNO dobijamo ASMD dijagram sa Sl. 9-14.

Broj naredbi registarskog prenosa koje se, obuhvaćene istim stanjem, izvršavaju u paraleli može biti proizvoljan, sve dok su odredišni registri ovih naredbi jedinstveni u okviru datog stanja. U ASMD dijagramu sa Sl. 9-14, stanje PASIVNO sadrži dve naredbe registarskog prenosa, koje iniciraju upise u registre $r1$ i $r2$ (naglasimo da će sam upis biti obavljen na početku sledećeg taktnog perioda). Zapazimo da je nakon učinjene intervencije ispitivanje uslova $r1 \geq y$ takođe pripojeno stanju PASIVNO, što može da dovede do problema u radu, jer se u istom stanju $r1$ koristi i kao odredišni registar u naredbi registarskog prenosa i kao operand u relacionom izrazu. Međutim, zahtev da korisnik treba da čeka bar još dva taktna ciklusa nakon što postavi nove vrednosti deljenika i delioca (vidi specifikaciju interfejsa iz Pr. 9-5), garantuje da će u momentu kada vrednost signala *start* postane ´1´, registar $r1$ već sadržati novu vrednost deljenika.



Sl. 9-14 ASMD dijagram sekvencijalnog delitelja sa izbačenim stanjem INIT (pogrešno rešenje).

Međutim, u ovoj verziji ASMD dijagrama javlja se novi problem, koji nije postojao u prethodnim varijantama - količnik se zadržava u registru *r*2 samo tokom jednog taktnog perioda. Razlog za ovakvo ponašanje je taj što se ASMD vraća u stanje PASIVNO odmah nakon završenog izračunavanja, gde se u *r*2 upisuje 0 već u sledećem taktnom ciklusu. S matematičke tačke gledišta, ASMD je ispravan, ali s tačke gledišta korisničkog interfejsa, ovakvo ponašanje je neprihvatljivo.



(a) (b)

Sl. 9-15 ASMD dijagram sekvencijalnog delitelja s dodatnim registrom za pamćenje količnika: (a) pogrešno rešenje; (b) ispravno rešenje.

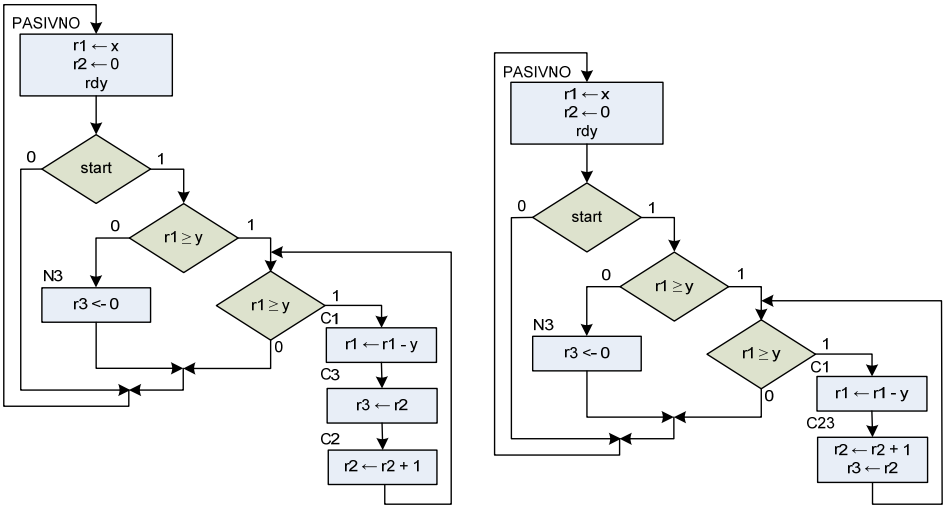
Problem u radu interfejsa koji postoji u ASMD dijagramu sa Sl. 9-14 može se prevazići uvođenjem još jednog registra, *r*3, koji će se koristi za pamćenje količnika (Sl. 9-15(a)). ASMD sa Sl. 9-15(a) radi korektno za $x \geq y$ (s napomenom da je sada količnik smešten u *r*3, a ne više u *r*2). Nažalost, i u ovom ASMD dijagramu postoji greška - u slučajevima kad rezultat treba da bude nula (tj. onda kad važi $x<y$), registar *r*3 ostaje nepromenjen, umesto da bude obrisan. To se događa usled toga što se upis u *r*3 obavlja samo unutar petlje.

Međutim, u slučajevima kad važi $x<y$, u petlju se ne ulazi, a $r3$ zadržava zatečenu vrednost (tj. rezultat prethodnog izračunavanja). Ovaj problem se može rešiti ako se uvede još jedno grananje koje će ispitati uslov $x<y$ (odnosno uslov $r1\geq y$) neposredno pre ulaska u petlju, kao što je prikazano na Sl. 9-15(b). Međutim, uvođenje dodatnog stanja, C3, ima za posledicu produženje vremena izračunavanja, koje sada iznosi: $2+3q$. (Ponekad, projektant mora da razmotri i sporije rešenje, da bi nakon toga, eventualno, došao do bržeg).

Pr. 9-11 Optimizacija sekvencijalnog delitelja - parelelizacija petlje

U ovom primeru, polazeći od ASMD dijagrama sa Sl. 9-15(b), isprobaćemo različite varijante realizacije petlje, u smislu promene redosleda operacija i uvođenja paralelizma u izvršenju operacija. Konačni cilj je naći rešenje koje će biti brže od polaznog.

Zamena redosleda stanja C2 i C3 daje pogrešno rešenje (Sl. 9-16(a)). Očigledno, upis u $r3$ se obavlja prerano, pa je usled toga rezultat za jedan manji od ispravnog. Još jedna varijanta izvođenja petlje, koja je iz sličnih razloga kao prethodna takođe pogrešna, dobija se spajanjem stanja C2 i C3 u jedinstveno stanje C23 (Sl. 9-16(b)). Bez obzira na to što je unutar bloka stanja C23 operacija upisa u registar $r2$ napisana ispred operacije upisa u registar $r3$, ove dve operacije se izvršavaju u paraleli. Budući da između operacija $r2 \leftarrow r2 + 1$ i $r3 \leftarrow r2$ postoji zavisnost, ASMD sa Sl. 9-16(b) nije ekvivalentan korektnom ASMD dijagramu sa Sl. 9-15(b), već je njegovo ponašanje identično ponašanju pogrešnog ASMD dijagrama sa Sl. 9-16(a) (zato što se u $r3$ ne upisuje vrednost registra $r2$ iz tekuće, već iz prethodne iteracije petlje).

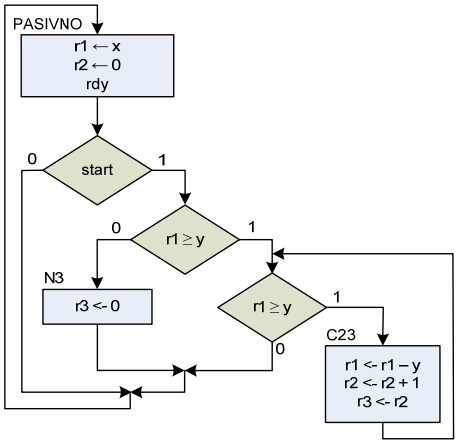


(a) (b)

Sl. 9-16 Varijacije u petlji ASMD dijagrama sekvencijalnog delitelja (pogrešna rešenja).

Varijanta petlje koja daje ne samo ispravno već i brže rešenje jeste ona kod koje su sve tri operacije, koje čine telo petlje, spojene u jedno stanje, C123 (Sl. 9-17). ASMD dijagram sa Sl. 9-17 je ispravan uprkos tome što su narušena oba pravila prevođenja algoritma u ASMD: prvo, zbog toga što su u isto stanje raspoređene međusobno zavisne naredbe ($r2 \leftarrow r2 + 1$ i $r3 \leftarrow r2$), i drugo, zato što je ispitivanje uslova $r1\geq y$ pridruženo stanju C123 u kome se $r1$ koristi kao odredišni registar naredbe dodele. Zbog toga što se $r1$ ispituje u istom stanju u kome se i menja, broj iteracija petlje biće za jedan veći od ispravnog broja

(kao u ASMD dijagramu sa Sl. 9-13(b) iz Pr. 9-9). Međutim, ova nedoslednost u radu kompenzuje se time što se u *r*3 prepisuje vrednost registra *r*2 ne iz tekuće, već iz prethodne iteracije (kao u ASMD dijagramu sa Sl. 9-16(b)). Konačan rezultat je takav da po izlasku iz petlje *r*3 sadrži tačan rezultat! Izračunavanje traje samo 2+*q* taktnih ciklusa.



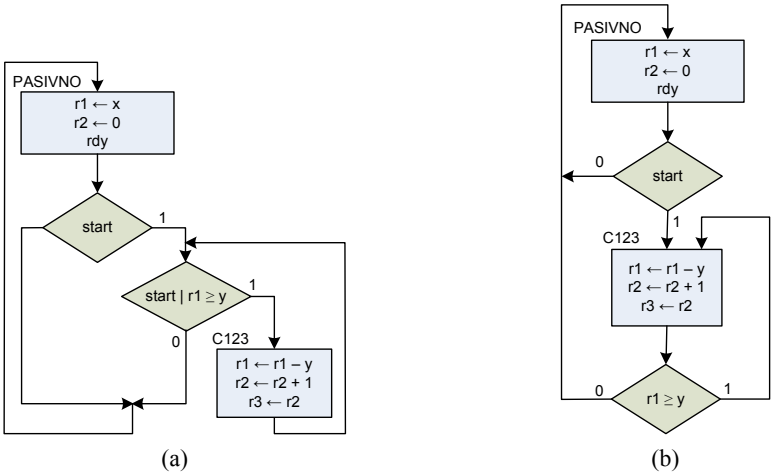
Sl. 9-17 ASMD dijagram sekvencijalnog delitelja sa paralelizovanom petljom.

Pr. 9-12 Optimizacija sekvencijalnog delitelja - eliminacija stanja N3

Pažljivom analizom rada ASMD dijagrama iz Pr. 9-11 možemo zaključiti da bi stanje N3 moglo biti eliminisano ako se obezbedi da ASMD uđe u petlju i za $x<y$, jer bi tada naredba registarskog prenosa $r3 \leftarrow r2$ iz stanja C123 imala isti efekta kao naredba $r3 \leftarrow 0$ iz stanja N3. (Zato što pri napuštanju stanja PASIVNO, *r*2 ima vrednost 0). Drugim rečima, treba obezbediti da ASMD iz stanja PASIVNO pređe u stanje C123 bez obzira na ishod ispitivanja $r1 \geq y$, a da nakon toga nastavi da koristi ovo ispitivanje kao test za kraj petlje. Razdvajanje prvog ulaska u stanje C123 i ulaska u isto ovo stanje koje je posledica ponavljanja petlje može se ostvariti na osnovu ispitivanja vrednosti ulaznog upravljačkog signala *start*, koji ima vrednost ´1´ pri prelasku iz stanja PASIVNO u stanje C123, a ´0´ za sve vreme dok u petlji traje izračunavanje količnika. (Ovo podrazumeva da sistem koristi ¨prijateljski¨ korisnik, koji se pridržava zahteva da trajanje *start*=´1´ mora biti tačno jedan taktni ciklus.)

Na Sl. 9-18(a) je prikazan ASMD dijagram u kome je ispitivanje uslova za kraj petlje prošireno testiranjem ulaznog signala *start* (″|″ označava ″ili″ operaciju). Aktiviranje signala *start*, koji traje tačno jedan taktni ciklus, ima kao jedinu posledicu inicijalni ulazak u petlju. Svako naredno eventualno ponavljanje petlje zavisi isključivo od sadržaja registra *r*1. Efekat pridruživanja signala *start* uslovu $r1 \geq y$ ispoljava se pri $x<y$. Drugim rečima, umesto da se stanje C123 ne izvrši ni jedanput, *start*=´1´ forsira da se ovo stanje ipak izvrši. U tom slučaju, stanje C123 se izvršava tačno jedanput zato što *x* (tj. *r*1) nije jednako ili veće od *y*. Činjenica da se nakon izvršenja stanja C123 u ASMD-u sa Sl. 9-18(a) sadržaji registara *r*1 i *r*2 razlikuju od onih koje su oni imali nakon izvršenja stanja N3 u prethodnoj varijanti ASMD-a nije od značaja budući da korisnik očekuje konačan rezultat u registru *r*3.

Na Sl. 9-18(b) je prikazana funkcionalno identična varijanta ASMD dijagrama sa Sl. 9-18(a). Razlika je u tome što je sada pojednostavljeno grananje koje je pridruženo stanju PASIVNO, jer ne sadrži nepotrebno ispitivanje uslova $r1 \geq y$.



Sl. 9-18 ASMD dijagram sekvencijalnog delitelja sa eliminisanim stanjem N3: (a) prva varijanta; (b) druga varijanta.

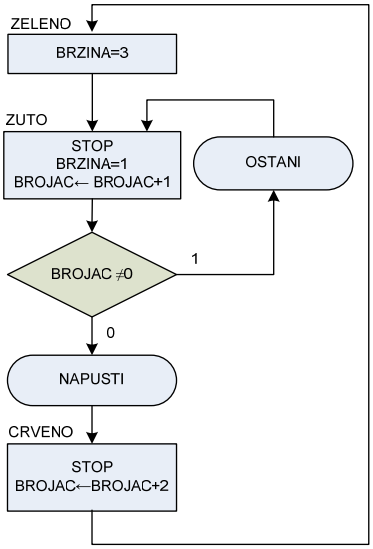
ASMD Milijevog tipa. U opštem slučaju, ASM dijagram se sastoji od blokova stanja, grananja i uslovnih izlaza povezanih granama (v. 6.1.2). U prethodnim primerima, blokovi uslovnog izlaza nisu korišćeni. ASM dijagram koji ne sadrži blokove uslovnog izlaza predstavlja konačni automat **Murovog** tipa. ASM dijagrami koji osim blokova stanja i grananja, sadrže i blokove uslovnog izlaza predstavljaju konačne automate **Milijevog** tipa. Isti algoritam se često može opisati ASMD dijagramom bilo Murovog bilo Milijevog tipa. U mnogim slučajevima, ASMD dijagrami Milijevog tipa su brži (tj. sadrže manji broj stanja) u odnosu na funkcionalno identične Murove ASMD dijagrame.

Operacije koje se javljaju u blokovima stanja nazivaju se *bezuslovnim* operacijama, budući da se izvršavaju uvek kad se ASMD nađe u datom stanju. Blokovi uslovnog izlaza se koriste da bi se predstavile *uslovne* operacije, tj. operacije koje se ponekad (ali ne i uvek) izvršavaju kada je ASMD u određenom stanju. Blokovi uslovnog izlaza nisu sami po sebi stanja, već se mogu smatrati ¨decom¨ nekog roditeljskog stanja. S obzirom na to što predstavljaju uslovne operacije, oni uvek slede posle jednog ili više blokova grananja. Kad se u ASMD dijagramu Milijevog tipa prate strelice, onda se polazeći od bloka stanja uvek prođe kroz jedan ili više blokova grananja dok se ne stigne do bloka uslovnog izlaza.

Sve operacije koje su navedene u bloku stanja i svim povezanim blokovima uslovnog izlaza i grananja koji slede (bez blokova stanja između njih) obavljaju se u istom taktnom ciklusu. U suštini, kombinacija blokova grananja i blokova uslovnog izlaza omogućava projektantu da realizuje ugnježdenu *if-then-else* konstrukciju koja se izvršava u jednom taktnom ciklusu. U takvom ASMD dijagramu, veliki broj uslova se može ispitivati u paraleli, što može bitno da ubrza rad algoritma.

Pr. 9-13 Jednostavan ASMD dijagram Milijevog tipa

ASMD dijagram sa Sl. 9-19 sadrži dva uslovna izlazna signala: OSTANI i NAPUSTI. Signal OSTANI ima vrednost 1 za vreme dok ASMD ostaje u stanju ŽUTO zbog *brojač* ≠ 0. Signal NAPUSTI ima vrednost 1 samo u poslednjem taktnom periodu u kome je ASMD u stanju ŽUTO. Uočimo da signali OSTANI i NAPUSTI nikada nisu aktivni u isto vreme. U tabeli T. 9-2 je ilustrovan rad ASMD dijagrama sa Sl. 9-19 pod pretpostavkom da taktni period iznosi 0.5 s i da je na početku rada sadržaj 3-bitnog registra *brojač* jednak ″000″:



Sl. 9-19 ASMD Miljevog tipa iz Pr. 9-13.

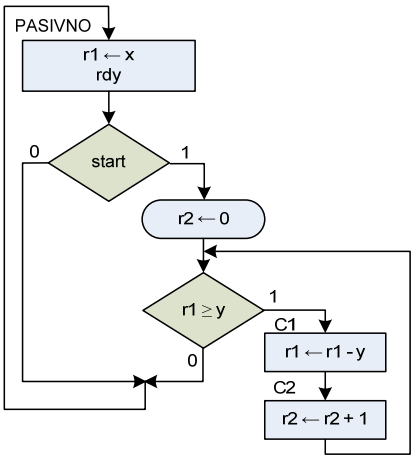
T. 9-2 Ilustracija rada ASMD dijagrama iz Pr. 9-13

Vreme (s)	Tekuće stanje	Signali i promenljive
0.0	ZELENO	STOP=0 BRZINA=11 BROJAČ=000 OSTANI=0 NAPUSTI=0
0.5	**ŽUTO**	**STOP=1 BRZINA=01 BROJAČ=000 OSTANI=0 NAPUSTI=1**
1.0	CRVENO	STOP=1 BRZINA=00 BROJAČ=001 OSTANI=0 NAPUSTI=0
1.5	ZELENO	STOP=0 BRZINA=11 BROJAČ=011 OSTANI=0 NAPUSTI=0
2.0	**ŽUTO**	**STOP=1 BRZINA=01 BROJAČ=011 OSTANI=1 NAPUSTI=0**
2.5	ŽUTO	STOP=1 BRZINA=01 BROJAČ=100 OSTANI=1 NAPUSTI=0
3.0	ŽUTO	STOP=1 BRZINA=01 BROJAČ=101 OSTANI=1 NAPUSTI=0
3.5	ŽUTO	STOP=1 BRZINA=01 BROJAČ=110 OSTANI=1 NAPUSTI=0
4.0	ŽUTO	STOP=1 BRZINA=01 BROJAČ=111 OSTANI=1 NAPUSTI=0
4.5	**ŽUTO**	**STOP=1 BRZINA=01 BROJAČ=000 OSTANI=0 NAPUSTI=1**
5.0	CRVENO	STOP=1 BRZINA=00 BROJAČ=001 OSTANI=0 NAPUSTI=0
5.5	ZELENO	STOP=0 BRZINA=11 BROJAČ=011 OSTANI=0 NAPUSTI=0
6.0	ŽUTO	STOP=1 BRZINA=01 BROJAČ=011 OSTANI=1 NAPUSTI=0
. . .	. . .	. . .

Između 0.5 s i 1.0 s, ASMD je u stanju ŽUTO, ali pošto je BROJAČ = 0, ASMD nastavlja dalje putanjom koja vodi u stanje CRVENO. Na ovoj putanji se nalazi blok uslovnog izlaza koji postavlja signal NAPUSTI. Ovaj uslovni signal je aktivan u toku celog taktnog ciklusa (0.5 - 1.0 s), baš kao bezuslovni signal STOP. Signal OSTANI, koji se nalazi na drugoj putanji, ostaje neaktivan u toku ovog taktnog ciklusa. Između 2.0 s i 2.5 s, ASMD je ponovo u stanju ŽUTO, ali budući da je sada vrednost registra BROJAČ različita od nule, ASMD nastavlja putanjom koja ga ostavlja u istom stanju. Ova putanja prolazi kroz blok uslovnog izlaza koji postavlja signal OSTANI. Signal NAPUSTI je neaktivan tokom ovog taktnog ciklusa. Sadržaj registra BROJAČ ponovo postaje nula u vremenu 4.5 - 5.0 s (s obzirom na to što je BROJAČ trobitni registar). To je zadnji taktni ciklus u kojem je ASMD u stanju ŽUTO. Znači, u toku ovog taktnog ciklusa, signal NAPUSTI je aktivan, a signal OSTANI nije.

Pr. 9-14 Milijeva verzija sekvencijalnog delitelja - eliminacija stanja INIT

Na Sl. 9-13(b) iz Pr. 9-9 prikazana je jedna korektna varijanta Murovog ASMD dijagrama sekvencijalnog delitelja sa četiri stanja i samo dva registra (*r*1 i *r*2). Na Sl. 9-14 je prikazan jedan neuspešan pokušaj eliminacije stanja INIT iz ovog ASMD dijagrama. U ASMD-u sa Sl. 9-15(b) uveden je registar *r*3 kako bi se ispravila greška u korisničkom interfejsu koja je postojala u ASMD-u sa Sl. 9-14.



Sl. 9-20 Milijev ASMD sekvencijalnog delitelja sa eliminisanim stanjem INIT.

Problem koji postoji u ASMD dijagramu sa Sl. 9-14 posledica je činjenice da se u stanju PASIVNO vrši bezuslovan upis u registar *r*2. Korišćenjem bloka uslovnog izlaza moguće je eliminisati stanje INIT, a da se pri tom ne naruši sadržaj registra *r*2 za vreme dok ASMD u stanju PASIVNO čeka da signal *start* postane aktiviran (Sl. 9-20). U ASMD dijagramu Milijevog tipa sa Sl. 9-20, upis nule u *r*2 se vrši u stanju PASIVNO, ali samo onda kad važi *start*=´1´, kao što je to za slučaj *x*=14 i *y*=7 prikazano u tabeli T. 9-3.

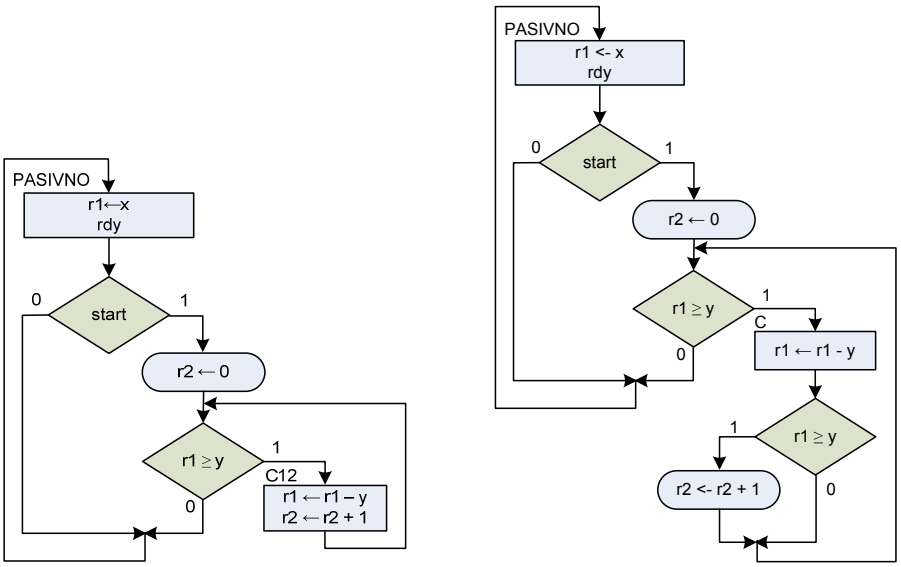
T. 9-3 Ilustracija rada ASMD dijagrama Milijevog tipa sa Sl. 9-20.

Tekuće stanje				
PASIVNO	r1=?	r2=?	start=0	rdy=1
PASIVNO	r1=?	r2=?	start=0	rdy=1
PASIVNO	**r1=14**	**r2=?**	**start=1**	**rdy=1**
C1	r1=14	r2=0	start=0	rdy=0
C2	r1=7	r2=0	start=0	rdy=0
C1	r1=7	r2=1	start=0	rdy=0
C2	r1=0	r2=1	start=0	rdy=0
PASIVNO	r1=0	r2=2	start=0	rdy=1
PASIVNO	r1=?	r2=2	start=0	rdy=1

Vrsta iz tabele T. 9-3 napisana masnim slovima ukazuje na situaciju u kojoj se ispoljava dejstvo uslovnog upisa nule u registar *r*2. Naravno, efekat ove operacije je vidljiv tek u trenutku delovanja naredne rastuće ivice taktnog signala, odnosno tek onda kad ASMD iz stanja PASIVNO pređe u stanje C1. Zapazimo da ASMD Milijevog tipa sa Sl. 9-20 izračunava količnik za 2+2**q* taktnih ciklusa, što je za jedan taktni ciklus brže u odnosu na korektan Murov ASMD dijagram sa dva registra iz Pr. 9-9.

Pr. 9-15 Milijeva verzija sekvencijalnog delitelja - spajanje stanja C1 i C2

ASMD sa Sl. 9-20 zahteva dvostruko duže vreme izračunavanja u odnosu na najbrže rešenje koje ne koristi blokove uslovnog izlaza (rešenje iz Pr. 9-12). Da bi smo postigli isti nivo performansi i sa Milijevom varijantom sekvencijalnog delitelja, neophodno je paralelizovati operacije koje se izvršavaju unutar petlje. Razmotrimo, najpre, **neispravan** Milijev ASMD dijagram sa Sl. 9-21(a).



Sl. 9-21 Milijev ASMD dijagram sekvencijalnog delitelja sa paralelizovanom petljom: (a) neispravno rešenje; (b) ispravno rešenje.

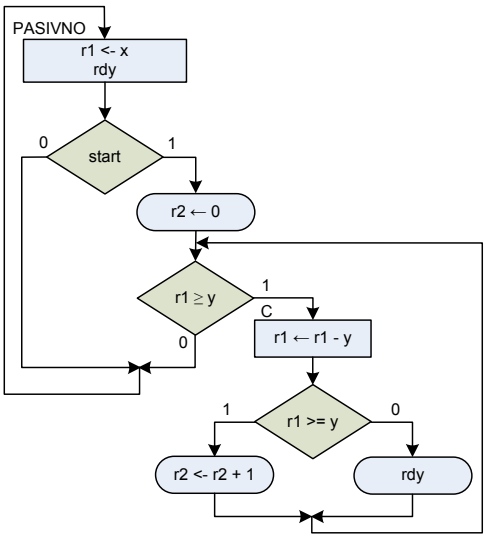
U trenutku kada se nakon obavljenog izračunavanja ASMD sa Sl. 9-21(a) vrati u stanje PASIVNO, registar *r*2 je inkrementiran jedanput više nego što bi trebalo (zbog toga što je ispitivanje uslova $r1 \geq y$ pridruženo stanju C12 u kojem se *r*1 koristi kao odredišni registar naredbe registarskog prenosa). U Pr. 9-10, ovaj problem je bio rešen uvođenjem registra *r*3, koji je služio za čuvanje ispravnog količnika. Međutim, bolje rešenje bi bilo ono koje ne zahteva uvođenje dodatnog registra. U tom cilju, neophodno je u stanju C12 vršiti upis u *r*2 ako ASMD ostaje u petlji, a zadržati zatečenu vrednost samo onda kad ASMD napušta petlju da bi se vratio u stanje PASIVNO. Ovakav rad ASMD-a zahteva uvođenje bloka uslovnog izlaza (Sl. 9-21(b)). ASMD dijagram sa Sl. 9-21(b) izračunava tačan količnik za $2+q$ taktnih ciklusa i to korišćenjem samo dva umesto tri registra, što znači da je podjednako brz kao i najbrži Murov ASMD i pri tom koristi manji broj registara.

Pr. 9-16 Milijeva verzija sekvencijalnog delitelja - ranije postavljanje signala *rdy*

Dva najbrža ASMD-a sekvencijalnog delitelja (Murov iz Pr. 9-12 i Milijeve iz Pr. 9-15) izračunavaju količnik za $2+q$ taktnih ciklusa. Konstanti član 2 potiče od zahteva da korisnik pre nego što inicira novo deljenje mora da čeka bar 2 taktna ciklusa nakon što sistem postavi signal *rdy*. Kod Murovih ASMD dijagrama situacija *rdy* =´1´ je isto što i biti u

stanju PASIVNO. Međutim, uz pomoć bloka uslovnog izlaza moguće je signal *rdy* postavi jedan taktni ciklus ranije. Postoje dva razloga koji opravdavaju ranije postavljanje signala *rdy*. Prvo, u toku poslednjeg taktnog ciklusa u kome je ASMD sa Sl. 9-21(b) u stanju C, registar *r*2 već sadrži ispravan količnik. Drugo, korisnik nije svestan tekućeg stanja sistema, već se oslanja na signal *rdy* koji ukazuje na trenutak kada je količnik izračunat i dostupan na izlazu *y*.

ASMD sa Sl. 9-22 postavlja signal *rdy* u stanju PASIVNO, ali takođe i u poslednjem taktnom ciklusu stanja C. Sve dok je ASMD u stanju C, test $r1 \geq y$ sa dna petlje se obavlja u isto vreme kada i test $r1 \geq y$ sa vrha petlje. U taktnim ciklusima kad ASMD treba da ostane u petlji, *r*2 se inkrementira. U taktnom ciklusu kada ASMD treba da napusti petlju, postavlja se signal *rdy*. Tabela T. 9-4 ilustruje rad ASMD dijagrama sa slike za x=14 i y=7:



Sl. 9-22 Konačni oblik Milijevog ASMD dijagrama sekvencijalnog delitelja.

T. 9-4 Ilustracija rada ASMD dijagrama iz Pr. 9-16.

Tekuće stanje				
PASIVNO	r1=?	r2=?	start=0	rdy=1
PASIVNO	r1=14	r2=?	start=1	rdy=1
C	r1=14	r2=0	start=1	rdy=0
C	r1=7	r2=1	start=0	rdy=0
C	**r1=0**	**r2=2**	**start=0**	**rdy=1**
PASIVNO	r1=4089	r2=2	start=0	rdy=1

U poslednjem taktnom ciklusu stanja C, *r*2 već sadrži korektan količnik i signal *rdy* je postavljen. Umesto da čeka još jedan taktni ciklus, kako bi se sistem vratio u stanje PASIVNO, korisnik može već od ovog momenta da otpočne sa odbrojavanjem dva ciklusa.

ASMD sa Sl. 9-22 izračunava korektan količnik korišćenjem dva registra za samo $1+q$ taktnih ciklusa. Znači, sekvencijalni delitelj realizovan na osnovu Milijevog ASMD dijagrama sa Sl. 9-22 je jednostavniji i brži od bilo kog rešenja zasnovanog na ASMD dijagramima Murovog tipa.

9.2.2. Razrada

ASMD dijagram je vrsta grafičke reprezentacije algoritma s preciznom informacijom o tajmingu i indikacijom o tome koje operacije se obavljaju u paraleli, a koje sekvencijalno. Međutim, krajnji cilj projektovanja jeste realizacija fizičkog sistema, tj. precizna specifikacija strukture sistema (na primer, u vidu logičke šeme). ASMD dijagram nam govori kako sistem radi, ali nam ne govori ništa o tome kako povezati digitalne komponente da bi se ostvarilo željeno ponašanje. Sledeća faza projektovanja, razrada, upravo uključuje jedan deo ukupne transformacije algoritma u hardversku strukturu.

U fazi razrade, cilj projektanta je da RTL sistem, čije ponašanje je prethodno precizno opisao ASMD dijagramom, podeli na dva razdvojena ali povezana podsistema: upravljačku jedinicu i staza podataka (v. Sl. 9-8). Uobičajeno je da se i u ovoj fazi projektovanja koristi ASM dijagram, ali samo za opis ponašanja upravljačke jedinice, dok se staza podataka predstavlja u vidu strukturnog opisa. ASM dijagram koji je pridružen upravljačkoj jedinici više ne sadrži naredbe registarskog prenosa i relacije (jer su registri prebačeni u stazu podataka), ali zato sadrži detaljan opis aktivnosti upravljačke jedinice, u vidu postavljanja upravljačkih i ispitivanja statusnih signala, koje su neophodne da bi se u stazi podatak obavila izračunavanja i registarski prenosi i to u redosledu i na način kako je to predviđeno polaznim ASMD dijagramom. Premda je nakon obavljene razrade struktura sistema složenija, a opis ponašanja sadrži više detalja, ponašanje celokupnog sistema, gledano sa strane spoljašnjeg sveta, ostaje identično polaznom ASMD dijagramu.

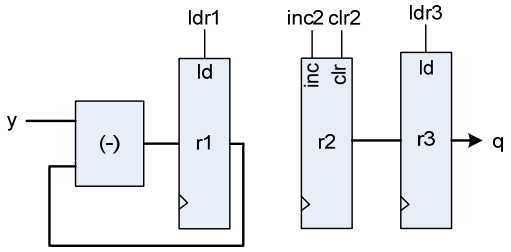
Pr. 9-17 Razrada sekvencijalnog delitelja – rešenje 1.

Kao što je opisano u uvodnom delu poglavlja 9.2, projektovanje RTL sistema na način ¨odozgo-naniže¨ uključuje tri faze: opis ponašanja, razradu i realizaciju, U primerima iz odeljka 9.2.1 predstavljene su različite varijante opisa ponašanja sekvencijalnog delitelja. U ovom primeru biće razmatrana druga faza projektovanja (razrada) ovog kola. Za ilustraciju razrade opisa ponašanja sekvencijalnog delitelja u strukturu koju će činiti upravljačka jedinica i staza podataka možemo uzeti bilo koji korektan ASMD dijagram iz odeljka 9.2.1. Na primer, usredsredimo se na ASMD sa Sl. 9-18(b) iz Pr. 9-12, koji je najjednostavniji (u pogledu broja stanja), ali ujedno i najbrži Murov ASMD sekvencijalnog delitelja. Za početak, ignorisaćemo signale *start* i *rdy*, budući da se radi o upravljačkom ulazu i statusnim izlazu sistema, koji ionako ostaju nepromenjeni u fazi razrade. Faza razrade je prevashodno fokusirana na hardversku realizaciju naredbi registarskog prenosa i relacionih izraza iz polaznog ASMD dijagrama.

Treba napomenuti da, po pravilu, ne postoji smo jedna već mnoštvo staza podataka različitih hardverskih struktura u kojima se mogu obaviti izračunavanja opisana polaznim ASMD-om. Na projektantu je da odluči koje će hardverske komponente koristiti za konstrukciju staze podataka, vodeći se kriterijumima kao što su: brzina rada, složenost i raspoloživost komponenti. Jedina dva striktna zahteva koja se pri tom postavljaju su da staza podataka treba da obezbedi: 1) obavljanje svih naredbi registarskog prenosa i relacionih izraza iz ASMD dijagrama i 2) paralelno izvršenje skupova naredbi i relacionih izraza iz istih stanja ASMD dijagrama. U konkretnom primeru, staza podataka sekvencijalnog delitelja treba da omogući izvršenje sledećih naredbi registarskog prenosa: $r1 \leftarrow x$, $r2 \leftarrow 0$, $r1 \leftarrow r1-y$, $r2 \leftarrow r2+1$ i $r3 \leftarrow r2$, kao i ispitivanje relacionog izraza $r1 \geq y$, i da pri tom omogući paralelno (istovremeno) izvršenje sledeća dva skupa operacija: ($r1 \leftarrow x$, $r2 \leftarrow 0$) – u stanju PASIVNO i ($r1 \leftarrow r1-y$, $r2 \leftarrow r2+1$, $r3 \leftarrow r2$ i $r1 \geq y$) – u stanju C123.

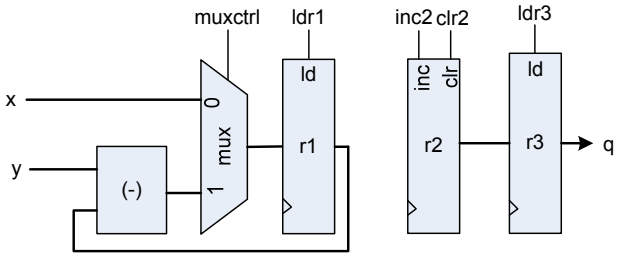
Najlakši (mada ne uvek i najbolji) način da se ovo postigne sastoji se u tome da se za ugradnju u stazu podataka biraju takve registarske komponente koje će pored osnovne funkcije memorisanja podataka omogućiti i neku dodatnu obradu koja se može iskoristiti za realizaciju celokupne (ili barem jednog dela) naredbe registarskog prenosa. Na primer, u konkretnom primeru, za realizaciju registra *r*2 možemo upotrebiti brojač. Brojač, kao registarska komponenta, pored memorisanja podatka dodatno omogućava brisanje (resetovanje) i inkrementiranje svog sadržaja - upravo one dve funkcije koje su potrebne za izvođenje naredbi registarskog prenosa iz stanja PASIVNO i C123 u kojima se *r*2 koristi kao odredišni registar, tj. *r*2←0 i *r*2←*r*2+1. Ukoliko se projektant odluči da za realizaciju registra *r*2 umesto brojača koristi prihvatni registar, on će morati da obezbedi dodatnu kombinacionu logiku kako bi nadomestio nedostatak funkcija resetovanja i inkrementiranja. Izbor prihvatnog registra za realizaciju registra *r*2 nije pogrešno rešenje, već samo rešenje koje zahteva nešto veće angažovanje projektanta. Da bi smo zadržali jednostavnost primera, usvojićemo da se *r*2 realizuje uz pomoć brojača. Za razliku od *r*2, registri *r*1 i *r*3 se pune vrednostima koje ne zavise isključivo od njihovog tekućeg sadržaja. Iz tog razloga, za realizaciju *r*1 i *r*3 razumno je koristiti najjednostavnije registarske komponente, koje osim bazičnih registarskih funkcija upisa i memorisanja podataka, poseduju još samo funkciju dozvole upisa.

Nakon što smo odlučili koje tipove registara ćemo koristiti u stazi podataka, neophodno je razmotriti na koji način će registri biti povezani. Za trenutak, usredsredimo se samo na naredbe registarskog prenosa iz stanja C123. U ovom stanju, *r*1 se puni razlikom *r*1-*y*, *r*2 se inkrementira, dok se u *r*3 prepisuje vrednosti iz registra *r*2. Sve ove operacije se izvršavaju u paraleli. Za izračunavanje razlike *r*1-*y* koristićemo oduzimač. Punjenje registra *r*3 vrednošću iz registra *r*2 se lako ostvaruje, tako što se izlaz brojačkog registra koji realizuje *r*2 direktno poveže sa ulazom prihvatnog registra koji realizuje *r*3. Kada bi se *r*1, *r*2 i *r*3 kao odredišni registri koristili samo u stanju C123, staza podataka sekvencijalnog delitelja izgledala bi kao na Sl. 9-23.



Sl. 9-23 Staza podataka sa oduzimačem. *Napomena*: *ld* – signal dozvole upisa u registar; *clr* – signal resetovanja brojača; *inc* – signal dozvole inkrementiranja brojača.

Uočimo da u stazi podataka sa Sl. 9-23 nije moguće realizovati naredbu registarskog prenosa *r*1←*x* iz stanja PASIVNO i to iz prostog razloga što ne postoji mogućnost da se u registar *r*1 upiše *x*. Pristup koji se koristi kad u različitim taktnim ciklusima u isti registar treba upisivati vrednosti koje potiču iz različitih izvora, sastoji se u ugradnji multipleksera na ulazu registra (slično kao u Pr. 9-3). U konkretnom primeru, za izbor ulaznog podatka registra *r*1 treba koristiti multiplekser 2-u-1. Jedan ulaz ovog multipleksera treba spojiti s izlazom oduzimača, a drugi s ulazim portom *x*. Nakon ugradnje multipleksera, staza podataka dobija oblik kao na Sl. 9-24. Uočimo da se vrednosti za upis u *r*1 bira posredstvom selekcionog ulaza multipleksera, *muxctri*. Multiplekser propušta *r*1-*y* za *muxctrl*=´1´, a *x* za *muxctrl*=´0´.



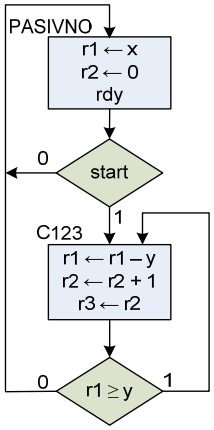
Sl. 9-24 Staza podataka sa multiplekserom.

Staza podataka sa Sl. 9-24, iako u stanju da realizuje sve naredbe registarskog prenosa iz ASMD dijagrama sa Sl. 9-18(b), ne pruža mogućnost ispitivanja relacionog izraza $r1 \geq y$. U hardveru, relacioni izraz se tipično realizuje u vidu kombinacione komponente koja generiše signal čija vrednost ukazuje na rezultat poređenja. U konkretnom primeru, koristićemo komparator s jednim izlazom, *r1y*, čija će vrednost biti *r1y*=′1′ ako važi $r1 \geq y$, odnosno *r1y*=′0′, u suprotnom slučaju.

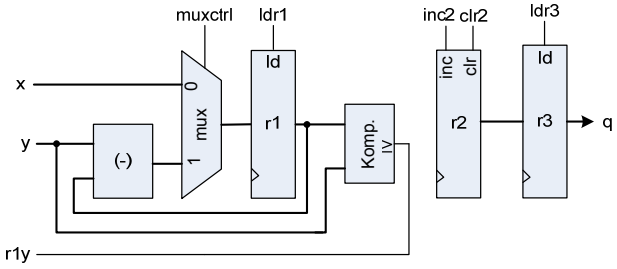
Opisani postupak razrade sekvencijalnog delitelja ilustrovan je na Sl. 9-25. Na Sl. 9-25(a) ponovljen je ASMD sa Sl. 9-18(b), dok je staza podataka, projektovana u ovom primeru, prikazana na Sl. 9-25(b). Na Sl. 9-25(c) data je tabela operacija u kojoj su za svaku naredbu registarskog prenosa iz ASMD dijagrama navedene vrednosti upravljačkih signala kojim se inicira njeno izvršenje u stazi podataka sa Sl. 9-25(b). Oznaka ″X″ znači da vrednost konkretnog upravljačkog signala nije od značaja za datu naredbu. U ovoj tabeli su, takođe, za oba stanja ASMD dijagrama, PASIVNO i C123, uvrštene vrste sa ″zbirnim″ vrednosti upravljačkih signala kojim se u stazi podataka iniciraju sve naredbe iz datog stanja. Uočimo da među naredbama registarskog prenosa iz tabele operacija postoji i naredba $r3 \leftarrow r3$. Ova naredba nije navedena u ASDM dijagramu, ali se podrazumeva u stanju PASIVNO, u kojem *r*3 zadržava svoju vrednost.

Sada kad su poznati svi detalji staze podataka, prevođenje ASMD dijagrama u ASM dijagram upravljačke jedinice predstavlja, u suštini, mehanički postupak. ASM dijagram upravljačke jedinice, prikazan na Sl. 9-25(d), ima identičan oblik (isti skup na isti način povezanih stanja) kao polazni ASMD, s tom razlikom što su u njegovim blokovima stanja umesto naredbi registarskog prenosa navedena imena upravljačkih signala koji su aktivni u datom stanju. Takođe, relacioni izraz je zamenjen testiranjem statusnog signala *r1y*. Konačno, blok dijagram sa Sl. 9-25(e) pokazuje spregu upravljačke jedinice i staze podataka.

Treba napomenuti da prelaz sa ASMD dijagrama, koji opisuje ponašanje celokupnog sistema na ASM koji opisuje ponašanje upravljačke jedinice, uvek ima u vidu strukturu staze podataka. Premda je obično moguće da za isti ASMD postoji veći broj staza podataka različite strukture koje mogu realizovati naredbe registarskog prenosa i relacione izraze sadržane u ASMD dijagramu, svaka od tih varijanti zahtevaće različiti ASM dijagram upravljačke jedinice, prilagođen strukturi konkretne staze podataka.



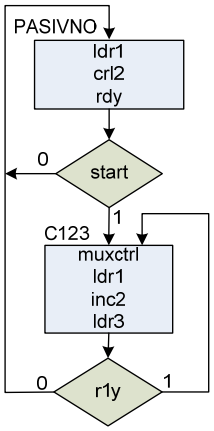
(a)



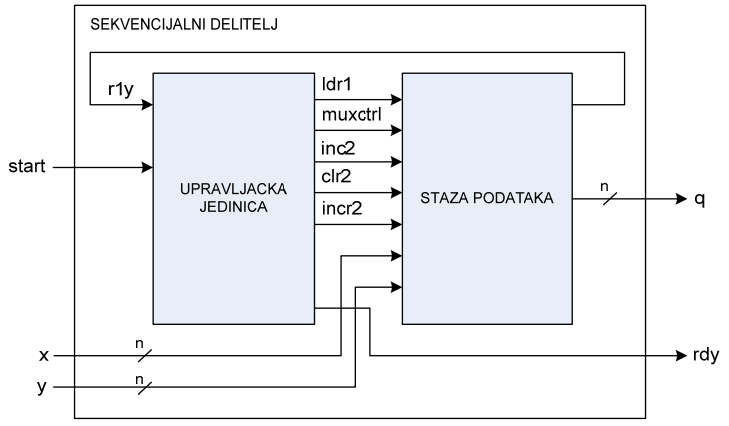
(b)

	muxctrl	ldr1	inc2	clr2	ldr3
$r1 \leftarrow x$	0	1	X	X	X
$r2 \leftarrow 0$	X	X	0	1	X
$r3 \leftarrow r3$	X	X	X	X	0
PASIVNO	**0**	**1**	**0**	**1**	**0**
$r1 \leftarrow r1 - y$	1	1	X	X	X
$r2 \leftarrow r2 + 1$	X	X	1	0	X
$r3 \leftarrow r2$	X	X	X	X	1
C123	**1**	**1**	**1**	**0**	**1**

(c)



(d)

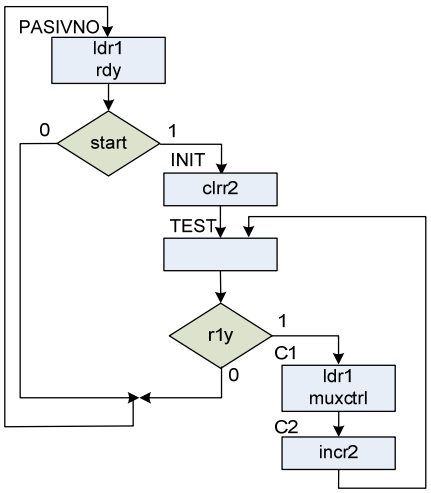


(e)

Sl. 9-25 Razrada sekvencijalnog delitelja: (a) ASMD; (a) staza podataka; (b) ASM dijagram upravljačke jedince; (c) sistemski blok dijagram.

Pr. 9-18 Razrada sekvencijalnog delitelja - rešenje 2

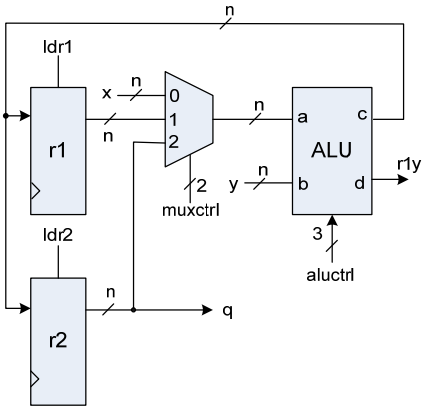
Pažljivom analizom staze podataka iz prethodnog primera možemo zaključiti da ona, iako je razvijena u skladu sa ASMD dijagramom sekvencijalnog delitelja iz Pr. 9-12, takođe može da realizuje i bilo koji korektan ASMD dijagrama iz Pr. 9-8 i Pr. 9-9. Međutim, to nije morao biti slučaj; šta više, retko se dešava da jedna staza podataka može poslužiti za realizaciju više različitih ASMD-ova. Ono što izdvaja ASMD sa Sl. 9-18(b) od ostalih ASMD dijagrama predstavljenih u Pr. 9-8 i Pr. 9-9 jeste maksimalan paralelizam. Postoji jedno neformalno pravilo koje kaže da staza podataka projektovana za maksimalan paralelizam može realizovati bilo koji ASMD sa manjim nivoom paralelizma koji je izveden na osnovu istog polaznog algoritma. Na primer, prvi ASMD iz Pr. 9-8 (Sl. 9-12(a)), karakterističan po strogo sekvencijalnom načinu rada, može biti realizovan korišćenjem staze podataka iz prethodnog primera (Sl. 9-25). Na Sl. 9-26 je prikazan ASM dijagram upravljačke jedinice izveden na osnovu ASMD dijagrama sa Sl. 9-12(a) i staze podataka sa Sl. 9-25(b). U ovom ASMD dijagramu, registar *r*3 se ne koristi, što znači da će izlaz delitelja biti *r*2 umesto *r*3.



Sl. 9-26 Razrađeni ASM dijagram iz Pr. 9-18. ***Napomena*****: u blokovima stanja navedena su samo imena onih signala koji u datom stanju imaju vrednost ´1´.**

Pr. 9-19 Razrada sekvencijalnog delitelja - rešenje 3

Kod sekvencijalnih ASMD-ova, kakav je npr. ADMD sa Sl. 9-12(a), važi ograničenje da se u jednom stanju može obaviti najviše jedna operacija, dok kod paralelnih, kakav je npr. ASMD sa Sl. 9-18(b), ovo ograničenje ne postoji. Za realizaciju paralelnog ASMD-a neophodna je paralelna staza podataka, odnosno staza podatka koja će biti u mogućnosti da u jednom taktnom ciklusu istovremeno izvrši više različitih operacija, kao što je to slučaj sa stazom podataka sa Sl. 9-25(a). U Pr. 9-18 je pokazano kako se sekvencijalni ASMD može realizovati u paralelnoj stazi podataka. U ovom primeru predstavljena je sekvencijalna staza podataka koja je namenski projektovana za sekvencijalni ASMD sa Sl. 9-12(a) iz Pr. 9-8. Budući da sekvencijalna staza podataka u svakom taktnom ciklusu treba da izvrši samo jednu operaciju, razumno je očekivati da će njena hardverska složenosti biti manja od složenosti paralelne staze podataka.



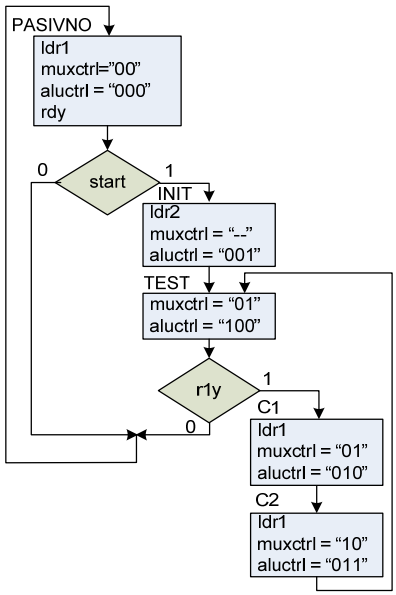
(a)

Operacija	aluctrl	c	d
Propuštanje	000	a	-
″Sve nule″	001	0	-
Oduzimanje	011	a-b	-
Inkrementiranje	010	a+1	-
Veće-jednako	100	-	a≥b

(b)

Naredba/ relacija	ldr1	ldr2	muxctrl	aluctrl
*r*1←*x*	1	X	00	000
*r*2←0	-	1	- -	001
*r*1←*r*1-*y*	1	-	01	010
*r*2←*r*2+1	-	1	10	011
*r*1≥*y*	-	-	01	100

(c)



(d)

Sl. 9-27 Sekvencijalni delitelj sa centralizovanom ALU: (a) staza podataka; (b) tabela operacija ALU; (c) tabela operacija staze podataka; (d) ASM dijagram upravljačke jedinice.

Uobičajen načina za realizaciju sekvencijalnih staza podataka zasnovan je na upotrebi ALU kao centralne jedinice uz pomoć koje se obavljaju sve operacije iz polaznog sekvencijalnog ASMD dijagrama. Na Sl. 9-27(a) je prikazana struktura staze podataka zasnovane na ALU za ASMD sa Sl. 9-12(a). Kod rešenja ovog tipa, multiplekseri na ulazima registara nisu potrebni, jer se podaci koji se upisuju u registre uvek uzimaju iz istog izvor, tj. sa izlaza ALU jedinice. S druge strane, budući da se ALU koristi za obavljanje različitih operacija nad operandima koji potiču iz različitih izvora, multiplekseri su neophodni na njenim ulaznim portovima. Na Sl. 9-27(b) je data tabela operacija ALU jedinice. Skup podržanih operacija je određen naredbama registarskog prenosa i relacionim izrazima koje treba realizovati. Prvo, ALU treba da poseduje mogućnost da podatak sa ulaznog porta *a* prenese na svoj izlaz *c*. Ova operacija, propuštanje, koristi se za realizaciju naredbe registarskog

prenosa $r1 \leftarrow x$. Drugo, radi realizacije naredbe $r2 \leftarrow 0$, ALU treba da je u mogućnosti da na svom izlazu postavi ″sve nule″. Treća operacije, inkrementiranje, koristi se za realizaciju naredbe $r2 \leftarrow r2+1$, a četvrta, oduzimanje, za realizaciju naredbe $r1 \leftarrow r1-y$. Poslednja, peta operacija, veće-jednako, koristi se za realizaciju ispitivanja uslova $r1 \geq y$. Rezultat ovog poređenja se generiše na jednobitnom izlazu *d*. Više detalja u vezi realizacije opisane ALU jedinice može se naći u Pr. 9-21.

Na Sl. 9-27(c) je data tabela operacija kompletne staze podataka. Konačno, na Sl. 9-27(d) je prikazan ASM dijagram upravljačke jedinice, koji je izveden na osnovu ASMD dijagrama sa Sl. 9-18(b) i tabele operacija sa Sl. 9-27(d).

Primeri razrade opisa ponašanja u strukturu ¨upravljačka jedinica – staza podataka¨, opisani u primerima iz ovog odeljka, ukazuju da se u fazi razrade projektant susreće sa širokim spektarom mogućnih rešenja. Staza podataka zasnovana na centralizovanoj ALU (ilustrovana u Pr. 9-19), koja obavlja sva potrebna izračunavanja, predstavlja jedan kraj ovog spektra. Druga krajnost je paralelna staza podataka, ilustrovana u Pr. 9-17 i Pr. 9-18, kod koje se sva izračunavanja predviđena algoritmom obavljaju zasebnim hardverskim jedinicama. Pristup zasnovan na centralizovanoj ALU tipično zahteva manji utrošak hardvera, ali zato može da realizuje samo sekvencijalne ASMD-ove. Na primer, ASMD dijagrami koji predviđaju više od jednog izračunavanja po taktnom ciklusu ne mogu se ostvariti u stazi podataka zasnovanoj na centralizovanoj ALU, zato što ALU može da obavi samo jednu operaciju u jednom taktnom ciklusu.

9.2.3. Realizacija

U završnoj faza projektovanja, projektant se bavi realizacijom hardvera RTL sistema. Tokom ove faze, razrađena struktura RTL sistema se opisuje u jeziku za opis hardvera, kao što je VHDL, a zatim sintetiše u hardver pomoću softvera za sintezu. Pri tom se ASM dijagram upravljačke jedinice opisuje u vidu konačnog automata, dok se staza podataka opisuje funkcionalnim/strukturnim VHDL kôdom. Za realizaciju staze podataka projektanti često koriste pretprojektovane komponente (registri, ALU, multiplekseri). Konačni VHDL opis RTL sistema dobija se strukturnim povezivanjem upravljačke jedinice i staze podataka.

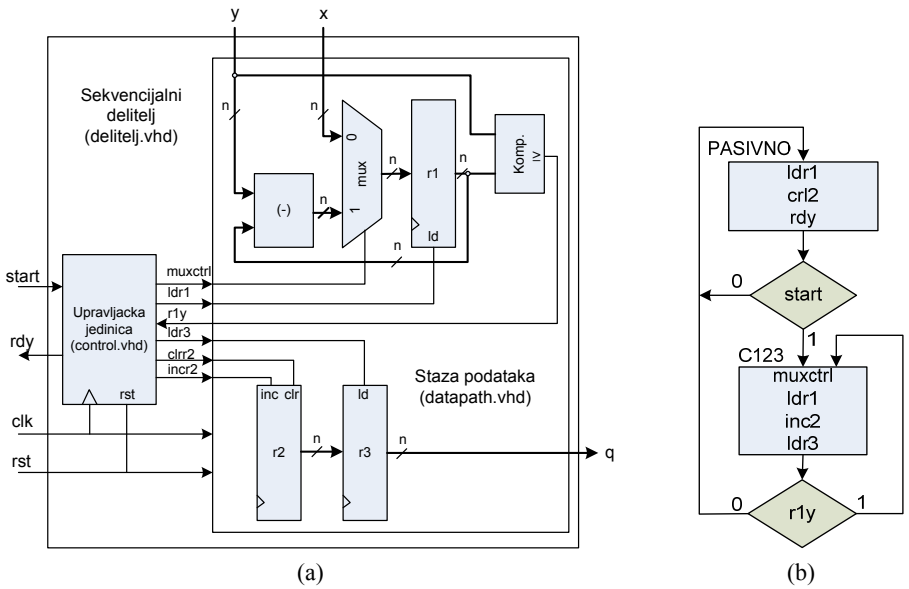
Prilikom korišćenja VHDL-a u kombinaciji sa softverom za sintezu, kreativni deo posla projektanta se obično završava okončanjem faze razrade. Ponekad, projektanti preskaču čak i fazu razrade i prepuštaju softveru za sintezu da na osnovu VHDL opisa ASMD dijagrama automatski obavi sintezu hardvera. Međutim, ovakav pristup je primenljiv samo pri projektovanju relativno jednostavnih RTL sistema, koji u ograničenom obimu koriste aritmetičke operacije. U svakom slučaju, važno je da projektant dobro razume mogućnosti i ograničenja softvera za sintezu, kako bi bio u stanju da kreira efikasan i korektan VHDL kôd.

Rezultat druge faze projektovanja je opis sistema u vidu strukture koju čine upravljačka jedinica i staza podataka, zajedno sa ASM dijagramom upravljačke jedinice. Međutim, čak i na ovom nivou, može se desiti da pojedini moduli iz staze podataka nisu raspoloživi u vidu pretprojektovanih komponenti ili su previše složena za automatsku sintezu. Takvi moduli moraju se nezavisno projektovati i opisati u VHDL-u. Ukoliko se radi o složenim modulima, proces njihovog projektovanja može zahtevati dodatu razradu. Međutim, budući da je njihov interfejs već definisan u fazi razrade staze podataka globalnog sistema, projektant može slobodno da se usredsredi na projektovanje konkretnog modula siguran da

će se on bez problema uklopiti u strukturu staze podataka. U svakom slučaju, konačni rezultat projektovanja koje sledi koncept ″odozgo-naniže″ je hijerarhijski VHDL opis koji se može razložiti na bibliotečke (pretprojektovane) komponente i module opisane funkcionalnim i/ili strukturnim kôdom koji se može sintetizovati.

Pr. 9-20 Realizacija sekvencijalnog delitelja – paralelna staza podataka

U ovom primeru biće predstavljena realizacija (u vidu VHDL opis) sekvencijalnog delitelja na osnovu staze podataka i ASM dijagrama upravljačke jedinice koji su razvijeni u Pr. 9-17. Sl. 9-28(a) prikazuje detaljan blok dijagram sekvencijalnog delitelja i ujedno ukazuje na način organizacije VHDL projekta. Na Sl. 9-28(b) je ponovljen ASM dijagram upravljačke jedinice iz Pr. 9-17. VHDL projekat sadrži tri VHDL modula (datoteke): *datapath.vhd* – opis staze podataka, *control.vhd* – opis upravljačke jedinice i *delitelj.vhd* – vršni modul u kome su moduli staza podataka i upravljačka jedinica, dostupni u vidu VHDL komponenti, strukturno povezani.



(a) (b)

Sl. 9-28 Razrađeni projekat sekvencijalnog delitelja: (a) hardverski blok dijagram sistema; (b) ASM dijagram upravljačke jedinice.

U nastavku sledi kompletan VHDL kôd sekvencijalnog delitelja. Opis staze podataka (datoteka *datapath.vhd*) sadrži tri procesa koji su ″centrirani″ oko tri registra, *r*1, *r*2 i *r3*. Prvi proces opisuje deo staze podataka koji obuhvata registar *r*1 zajedno sa multiplekserom i oduzimačem (linije 17-31). Drugi proces opisuje registar *r*2, koji igra ulogu brojača (linije 33-44), a treći registar *r*3 (linije 46-55). Komparator za uslov ″veće-jednako″ realizovan je pomoću konkurentne WHEN naredbe (linija 57). U liniji 58, sadržaj registra *r*3 se prosleđuje na izlazni port staze podataka za količnik, *q*. VHDL kôd iz datoteke *control.vhd* opisuje upravljačku jedinicu sekvencijalnog delitelja korišćenjem dvosegmentnog kôdnog šablona. U datoteci *delitelj.vhd*, komponente *datapath* i *control* su instancirane i povezane na način kao na Sl. 9-28(a).

```
1 -- fajl datapath.vhd ------------------------------------------
2 LIBRARY IEEE;
```

```
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.NUMERIC_STD.ALL;
-----------------------------------------------------------------
ENTITY datapath IS
   PORT(x,y : IN STD_LOGIC_VECTOR(11 DOWNTO 0);
        q : OUT STD_LOGIC_VECTOR(11 DOWNTO 0);
        muxctrl,ldr1,clr2,inc2,ldr3 : IN STD_LOGIC;
        r1y : OUT STD_LOGIC;
        clk, rst : IN STD_LOGIC);
END datapath;
-----------------------------------------------------------------
ARCHITECTURE datapath OF datapath IS
   SIGNAL r1,r2,r3 : UNSIGNED(11 DOWNTO 0);
BEGIN
- registar r1 ------------------------------------------------
  PROCESS(clk,rst)
BEGIN
   IF(rst = '1') THEN
      r1 <= (OTHERS => '0');
   ELSIF(clk'EVENT AND clk='1') THEN
      IF(ldr1='1') THEN
         IF(muxctrl='1') THEN
            r1 <= r1 - UNSIGNED(y);
         ELSE
            r1 <= UNSIGNED(x);
         END IF;
      END IF;
   END IF;
END PROCESS;
-- registar r2 (brojac) ----------------------------------------
  PROCESS(clk,rst)
  BEGIN
     IF(rst = '1') THEN
        r2 <= (OTHERS => '0');
     ELSIF(clk'EVENT AND clk='1') THEN
        IF(clr2='1') THEN
           r2 <= (OTHERS=>'0');
        ELSIF(inc2='1') THEN
           r2 <= r2 + 1;
        END IF;
     END IF;
  END PROCESS;
-- registar r3 ------------------------------------------------
  PROCESS(clk,rst)
  BEGIN
     IF(rst = '1') THEN
        r3 <= (OTHERS => '0');
     ELSIF(clk'EVENT AND clk='1') THEN
        IF(ldr3 ='1') THEN
           r3 <= r2;
        END IF;
     END IF;
  END PROCESS;
  -- komparator -----------------------------------------------
  r1y <= '0' WHEN r1<UNSIGNED(y) ELSE '1';
```

```
58   q <= STD_LOGIC_VECTOR(r3);
59 END datapath;
60 -- fajl control.vhd ------------------------------------------
61 LIBRARY IEEE;
62 USE IEEE.STD_LOGIC_1164.ALL;
63 ----------------------------------------------------------------
64 ENTITY control IS
65   PORT(start, r1gey : IN STD_LOGIC;
66        rdy : OUT STD_LOGIC;
67        muxctrl,ldr1,ldr3,clr2,inc2 : OUT STD_LOGIC;
68        clk,rst : IN STD_LOGIC);
69 END control;
70 ----------------------------------------------------------------
71 ARCHITECTURE control OF control IS
72    TYPE state IS (PASIVNO, C123);
73    SIGNAL pr_state, nx_state : state;
74 BEGIN
75 - registar stanja ----------------------------------------------
76  PROCESS (clk,rst)
77  BEGIN
78    IF(rst='1')THEN
79      pr_state <=PASIVNO;
80    ELSIF(clk'EVENT AND clk='1') THEN
81      pr_state <= nx_state;
82    END IF;
83  END PROCESS;
84 - logika sledećeg stanja/izlaza ---------------------------------
85  PROCESS(start,r1gey,pr_state)
86   BEGIN
87     nx_state <= pr_state;
88     rdy<='0';muxctrl<='0';ldr1<='0';ldr3<='0';clr2<='0';inc2<='0';
89     CASE pr_state IS
90       WHEN PASIVNO =>
91         ldr1<='1'; clr2<='1';rdy<='1';
92         IF(start='1') THEN
93           nx_state <= C123;
94         END IF;
95       WHEN C123 =>
96         ldr1 <= '1';inc2 <= '1';ldr3 <= '1';muxctrl <= '1';
97         IF(start = '0' AND r1y = '0') THEN
98           nx_state <= PASIVNO;
99         END IF;
100       END CASE;
101   END PROCESS;
102  END control;
103    -- fajl delitelj.vhd ----------------------------------------
104  LIBRARY IEEE;
105  USE IEEE.STD_LOGIC_1164.ALL;
106  USE delitelj_pck.ALL; -- sadrzi deklaracije komponeti
107 ----------------------------------------------------------------
108  ENTITY delitelj IS
109    PORT(x, y : IN STD_LOGIC_VECTOR(11 DOWNTO 0);
110         q : OUT STD_LOGIC_VECTOR(11 DOWNTO 0);
111         rdy : OUT STD_LOGIC;
112         start, clk, rst : IN STD_LOGIC);
```

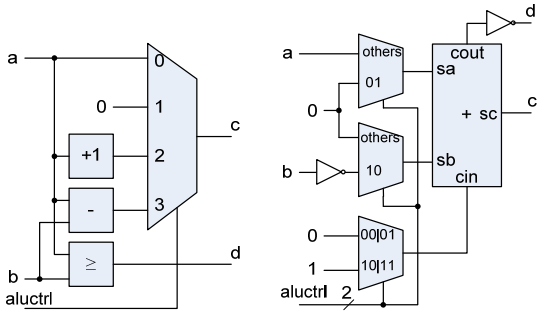
```
113  END delitelj;
114  ------------------------------------------------------------------
115  ARCHITECTURE delitelj OF delitelj IS
116    SIGNAL muxctrl,ldr1,clr2,inc2,ldr3,r1y : STD_LOGIC;
117  BEGIN
118    CTRL: control PORT MAP(start=>start,rdy=>rdy,muxctrl=>muxctrl,
119                           ldr1=>ldr1,ldr3=>ldr3,clr2=>clr2,
120                           inc2=>inc2,r1y=>r1y,clk=>clk,rst=>rst);
121    DTPH: datapath PORT MAP(x=>x,y=>y,q=>q,muxctrl=>muxctrl,
122                           ldr1=>ldr1,ldr3=>ldr3,clr2=>clr2,
123                           inc2=>inc2,r1y=>r1y,clk=>clk,rst=>rst);
124  END delitelj;
```

Pr. 9-21 Realizacija sekvencijalnog delitelja – sekvencijalna staza podataka

U ovom primeru predstavljena je realizacija, u vidu VHDL opisa, sekvencijalnog delitelja zasnovanog na stazi podataka sa ALU iz Pr. 9-19. Najpre će biti kreiran VHDL opis ALU jedinice, koji će potom, kao komponenta, biti iskorišćen u opisu staze podataka. VHDL projekat sadrži četiri datoteke: *alu.vhd* (opis ALU jedinice), *datapath.vhd* (opis staze podataka), *control.vhd* (opis upravljačke jedinice) i *delitelj.vhd* (vršni modul sa instanciranim i spojenim komponentama staze podataka i upravljačke jedinice).

Arhitektura *alu_v1* (linije 14-25) predstavlja direktnu realizaciju ALU, kreiranu shodno tabeli operacija sa Sl. 9-27(b), a po ugledu na opis ALU iz Pr. 4-14. Međutim, zbog većeg broja aritmetičkih operatora, ovakav VHDL opis ALU (a to važi i za opis iz Pr. 4-14) nije racionalan u pogledu hardverske sinteze. Sintetizovan hardver biće funkcionalno korektan, ali će sadržati čak tri aritmetičke komponente i to po jedan oduzimač, inkrementer i komparator, kao što je u vidu konceptualnog dijagrama prikazano na Sl. 9-29(a). Iz tog razloga, korisno je ispitati mogućnost deobe funkcija u cilju smanjena broja operatora sadržanih u opisu (v. 4.5.2).

Čest je slučaj da su većina operacija višefunkcionalnih aritmetičkih jedinica, kakva je i konkretna ALU, zasnovane na operaciji sabiranja. Tako se oduzimanje svodi na sabiranje umanjenika i potpunog komplementa umanjioca, a inkrementiranje na sabiranje sa konstantom 1. Čak se i ispitivanje uslova ″veće-jednako″ može realizovati posredstvom oduzimanja, i to na osnovu znaka razlike dva broja koji se porede. Kao što je poznato iz osnova digitalne elektronike, izlazni prenos koji se generiše prilikom oduzimanja neoznačenih binarnih brojeva ukazuje na znak razlike. Konkretno, ako je razlika negativna, izlazni prenos će imati vrednost 1, a ako je jednaka nuli ili pozitivna, vrednost izlaznog prenosa će biti 0. Imajući to u vidu, ishod poređenja $a \geq b$ identičan je komplementu vrednosti izlaznog prenosa pri operaciji a-b. Polazeći od jednog ovakvog razmatranja, na Sl. 9-29(b) je prikazan konceptualni dijagram optimizovane ALU za primenu u stazi podataka sekvencijalnog delitelja. Centralno mesto u ovom kolu zauzima binarni sabirač sa signalima ulaznog (*cin*) i izlaznog (*cout*) prenosa. U zavisnosti od vrednosti upravljačkog signala *aluctrl*, na ulaze sabirača se posredstvom multipleksera dovodi odgovarajuća kombinacija sabiraka i vrednosti ulaznog prenosa, na način kako je dato u tabeli sa Sl. 9-29(c). Propuštanje vrednosti sa ulaza a na izlaz c ostvaruje se tako što se a sabira s nulom. Postavljanje svih nula na izlazu c rezultat je operacije 0 + 0. Inkrementirana vrednost a se dobija tako što sabirač obavlja operaciju a + 0, uz *cin*=′1′. Za izračunavanje razlike $a - b$ i poređenje $a \geq b$, kolo se podešava tako da sabirač obavlja operaciju $a + \overline{b} + 1$.



(a) (b)

Operacija	aluctrl	sa	sb	cin
Propuštanje (c = a)	00	a	0	0
Sve nule (c = 0)	01	0	0	0
Oduzimanje (c = a – b)	10	a	$\bar{b}$	1
Inkrement (c = a + 1)	11	a	0	1

(c)

Sl. 9-29 ALU sekvencijalnog delitelja: (a) neoptimizovao rešenje; (b) rešenje optimizovano primenom deobe funkcija; (c) funkcionalna tabela kola sa Sl. 9-29(b).

Arhitektura *alu_v2* (linije 27-40) kreirana je shodno konceptualnom dijagramu sa Sl. 9-29(b). Naredba dodele iz linija 31 realizuje sabirač sa signalom ulaznog prenosa (v. Pr. 4-18). Uočimo da su vektorski signali *sa*, *sb* i *sc* (deklarisani u liniji 28) za jedan bit duži od dužine ulaznih portova *a* i *b*. Ovo produženje je neophodno kako izlazni prenos ne bi bio izgubljen prilikom sabiranja. Princip je taj da ako se na *N*+1 bitne ulaze sabirača dovedu *N*-bitni sabirci prošireni nulom s leve strane, tada će se na krajnjem levom bitu izlaznog signala *sc* sabirača, *sc*(*N*), generisati izlazni prenos *N*-bitnog sabiranja. U linijama 32 i 33, na izlazne portove *c* i *d* se prenose signali sume i izlaznog prenosa sabirača. Tri naredbe *when* koje slede u nastavku kôda, služe da bi se u zavisnosti od zahtevane operacije, a shodno tabeli sa Sl. 9-29(c), na ulaze sabirača postavile odgovarajuće vrednosti sabiraka i ulaznog prenosa.

```
1  -- fajl: ALU.vhd ------------------------------------------------
2  LBRARY IEEE;
3  USE IEEE.STD_LOGIC_1164.ALL;
4  USE IEEE.NUMERIC_STD.ALL;
5  ----------------------------------------------------------------
6  ENTITY alu IS
7   GENERIC(N : NATURAL);
8   PORT(a, b : IN STD_LOGIC_VECTOR(N-1 DOWNTO 0);
9        aluctrl : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
10       d : OUT STD_LOGIC;
11       c : OUT STD_LOGIC_VECTOR(N-1 DOWNTO 0));
12 END alu;
13 -- neoptimizovano resenje ---------------------------------------
14 ARCHITECTURE alu_v1 OF alu IS
15   SIGNAL dif, inc : STD_LOGIC_VECTOR(N-1 DOWNTO 0);
16 BEGIN
17   dif <= STD_LOGIC_VECTOR(UNSIGNED(a) - UNSIGNED(b));
18   inc <= STD_LOGIC_VECTOR(UNSIGNED(a) + 1);
19   c <= a               WHEN aluctrl = "00" ELSE
20        (OTHERS => '0') WHEN aluctrl = "01" ELSE
21        dif             WHEN aluctrl = "10" ELSE
22        inc;
23   d <= '1' WHEN a >= b ELSE
24        '0';
25 END alu_v1;
26 -- optimizovano resenje-------------------------------------------
27 ARCHITECTURE alu_v2 OF alu IS
```

```
  SIGNAL sa, sb, sc : UNSIGNED(N DOWNTO 0);
  SIGNAL cin : UNSIGNED(0 DOWNTO 0);
BEGIN
  sc <= sa + sb + cin;
  c  <= STD_LOGIC_VECTOR(sc(N-1 DOWNTO 0));
  d  <= NOT sc(N);
  sa <= (OTHERS => '0') WHEN aluctrl = "01" ELSE
        UNSIGNED('0' & a);
  sb <= UNSIGNED(NOT('0' & b)) WHEN aluctrl = "10" ELSE
       (OTHERS => '0');
  cin <= "1" WHEN aluctrl = "10" OR aluctrl = "11" ELSE
         "0";
END alu_v2;
- fajl datapath.vhd ------------------------------------------------
 LIBRARY IEEE;
 USE IEEE.STD_LOGIC_1164.ALL;
 USE delitelj_pck.ALL; -- sadrzi deklaracije komponeti
 ENTITY datapath IS
   GENERIC(N : NATURAL := 8);
   PORT(ldr1, ldr2 : IN STD_LOGIC;
        x, y : IN STD_LOGIC_VECTOR(N-1 DOWNTO 0);
        muxctrl, aluctrl : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
        r1y : OUT STD_LOGIC;
        q : OUT STD_LOGIC_VECTOR(N-1 DOWNTO 0);
        clk : IN STD_LOGIC);
 END datapath;
-------------------------------------------------------------------
 ARCHITECTURE datapath OF datapath IS
  SIGNAL alu_c, alu_a, r1, r2 : STD_LOGIC_VECTOR(N-1 DOWNTO 0);
  SIGNAL alu_d : STD_LOGIC;
 BEGIN
 -- registar r1 -----------------------------------------------------
 PROCESS(clk)
 BEGIN
   IF(clk'EVENT AND clk='1') THEN
     IF(ldr1='1') THEN
        r1 <= alu_c;
     END IF;
   END IF;
 END PROCESS;
 -- registar r2 -----------------------------------------------------
 PROCESS(clk)
 BEGIN
   IF(clk'EVENT AND clk='1') THEN
     IF(ldr2='1') THEN
        r2 <= alu_c;
     END IF;
  END IF;
 END PROCESS;
 -- mux -------------------------------------------------------------
 alu_a <= x  WHEN muxctrl = "00" ELSE
          r1 WHEN muxctrl = "01" ELSE
          r2;
 -- ALU -------------------------------------------------------------
 alu_1: alu GENERIC MAP (N => N)
```

```
83              PORT MAP(a=>alu_a, b=>y, aluctrl=>aluctrl,
84                       c=>alu_c, d=>alu_d);
85  -- izlaz ------------------------------------------------------
86  q <= r2;
87  r1y <= alu_d;
88 END datapath;
89 - fajl control.vhd ----------------------------------------------
90  LIBRARY IEEE;
91 USE IEEE.STD_LOGIC_1164.ALL;
92 ----------------------------------------------------------------
93  ENTITY control IS
94  PORT(clk, rst, start : IN STD_LOGIC;
95       r1y : IN STD_LOGIC;
96       ldr1, ldr2 : OUT STD_LOGIC;
97       rdy : OUT STD_LOGIC;
98       muxctrl, aluctrl : OUT STD_LOGIC_VECTOR(1 DOWNTO 0));
99  END control;
100  ---------------------------------------------------------------
101 ARCHITECTURE control OF control IS
102   TYPE state IS (PASIVNO, INIT, TEST, C1, C2);
103   SIGNAL pr_state, nx_state : state;
104 BEGIN
105   -- registar stanja  ----------------------------------------
106   PROCESS (clk,rst)
107   BEGIN
108     IF(rst='1')THEN
109       pr_state <=PASIVNO;
110     ELSIF(clk'EVENT AND clk='1') THEN
111       pr_state <= nx_state;
112     END IF;
113   END PROCESS;
114   -- logika sledeceg stanja-izlaza ----------------------------
115   PROCESS(start,r1y,pr_state)
116   BEGIN
117     nx_state <= pr_state;
118     rdy<='0';ldr1<='0';ldr2<='0';muxctrl<="00";aluctrl<="00";
119     CASE pr_state IS
120       WHEN PASIVNO =>
121         ldr1<='1';rdy<='1';
122         IF(start='1') THEN
123           nx_state <= INIT;
124         END IF;
125       WHEN INIT =>
126         ldr2<='1';aluctrl<="01";
127         nx_state <= TEST;
128         WHEN TEST =>
129           muxctrl<="01";aluctrl<="10";
130           IF(r1y = '1') THEN
131             nx_state <= C1;
132           ELSE
133             nx_state <= PASIVNO;
134           END IF;
135         WHEN C1 =>
136           ldr1<='1';muxctrl<="01";aluctrl<="10";
137           nx_state <= C2;
```

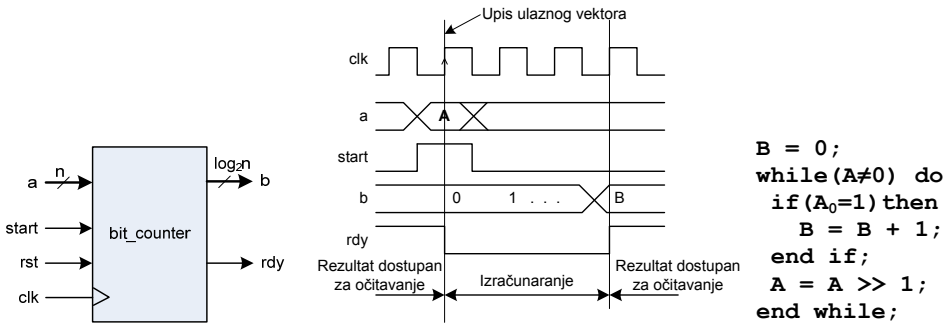
```
138         WHEN C2 =>
139           ldr2<='1';muxctrl<="10";aluctrl<="11";
140           nx_state <= TEST;
141      END CASE;
142    END PROCESS;
143 END control;
144 - fajl delitelj.vhd ------------------------------------------
145 LIBRARY IEEE;
146 USE IEEE.STD_LOGIC_1164.ALL;
147 USE delitelj_pck.ALL; -- sadrzi deklaracije komponeti
148 ---------------------------------------------------------------
149 ENTITY delitelj IS
150   GENERIC(N : NATURAL := 8);
151   PORT(clk,rst,start : IN STD_LOGIC;
152        x, y : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
153        q : OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
154        rdy : OUT STD_LOGIC);
155 END delitelj;
156 ---------------------------------------------------------------
157 ARCHITECTURE delitelj OF delitelj IS
158   SIGNAL ldr1,ldr2,r1y : STD_LOGIC;
159   SIGNAL muxctrl, aluctrl : STD_LOGIC_VECTOR(1 DOWNTO 0);
160 BEGIN
161   CTRL: control
162     PORT MAP (start=>start,rdy=>rdy,muxctrl=>muxctrl,
163               aluctrl=>aluctrl,ldr1=>ldr1,ldr2=>ldr2,
164               r1y=>r1y,clk=>clk,rst=>rst);
165   DTPH: datapath
166     GENERIC MAP(N=>N)
167     PORT MAP(x=>x,y=>y,q=>q,muxctrl=>muxctrl,aluctrl=>aluctrl,
168              ldr1=>ldr1,ldr2=>ldr2,r1y=>r1y,clk=>clk);
169 END delitelj;
```

9.3. Primeri RTL sistema

Pr. 9-22 Sekvencijalni brojač jedinica

Brojač jedinica (*bit_counter*) je kolo koje prebrojava 1-ce u ulaznom binarnom vektoru. U Pr. 8-24 predstavljena je kombinaciona varijantu ovog kola. U ovom primeru, realizovaćemo sekvencijalni brojač jedinica čiji je blok dijagram prikazan na Sl. 9-30(a). Ulazni vektor dužine n bita se dovodi na ulaz a. Rad brojača jedinica je ilustrovan vremenskim dijagramom sa Sl. 9-30(b). Prebrojavanje 1-ca se startuje aktiviranjem signala *start* u trajanju od jednog taktnog perioda. Nakon što završi rad, kolo postavlja rezultat na izlaz b i aktivira izlazni signal *rdy*. Za brojanje jedinica biće iskorišćen postupak (algoritam) koji je, u vidu pseudo kôda, dat na Sl. 9-30(c).

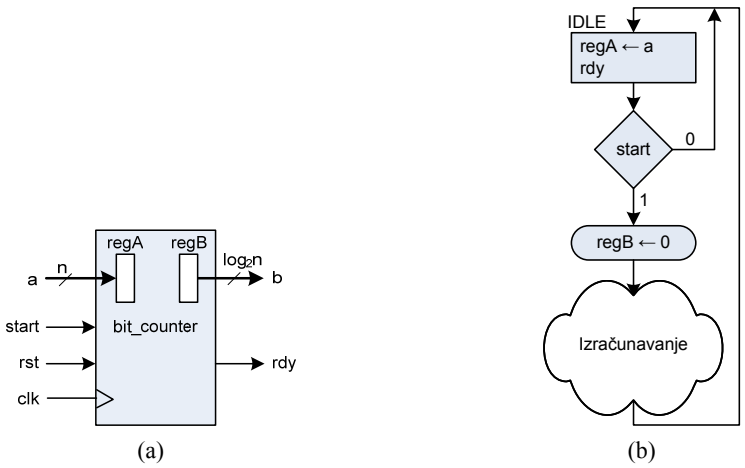
Algoritam. U algoritmu za brojanje 1-ca koriste se dve promenljive: A (sadrži ulazni vektor koje se obrađuje) i B (služi za odbrojanje 1-ca). Algoritam radi tako što pomera sadržaj promenljive A udesno i broji koliko puta se na poziciji najmanje težine (A_0) javila 1-ca. Algoritam se završava onda kad u promenljivoj A više nema 1-ca. Konačni rezultat je dostupan u promenljivoj B.



(a) (b) (c)

Sl. 9-30 Brojač jedinica: (a) blok dijagram; (b) vremenski dijagram; (c) algoritam.

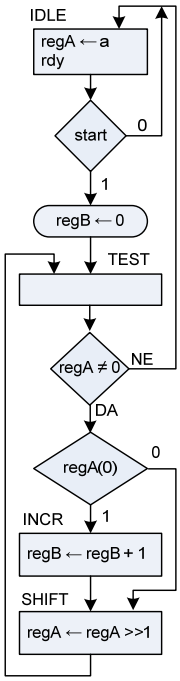
Interfejs i inicijalizacija. Za hardversku realizaciju brojača jedinica potrebna su dva registra: *regA* (odgovara promenljivoj *A*) i *regB* (odgovara promenljivoj *B*) (Sl. 9-31(a)). Na početku rada, u registar *regA* se upisuje ulazni vektor. Izlaz registra *regB* je ujedno i izlaz *b* ovog kola. Na Sl. 9-31(b) je prikazan segment ASMD dijagrama koji opisuje rad interfejsa i inicijalizaciju registara *regA* i *regB*. Stanje IDLE je pasivno (neaktivno) stanje brojača jedinica. U ovom stanju, postavljen je signal *rdy* koji ukazuje korisniku da je kolo spremno za rad. Takođe, u ovom stanju, u registar *regA* se upisuje vrednost sa ulaza *a*. Pri *start*=′1′, u registar *regB* se upisuju ″sve nule″, stanje IDLE se napušta i počinje izračunavanje.



(a) (b)

Sl. 9-31 Interfejs i registri brojača jedinica: (a) registri; (b) segment ASMD dijagrama.

***ASMD dijagram*.** Na Sl. 9-32(a) je prikazan ASMD dijagram brojača jedinica dobijen direktnom konverzijom algoritma sa Sl. 9-30(c). Uvedena su tri nova stanja, TEST, INCR i SHIFT. Uslovi iz *while* i *if* naredbe su pridruženi istom, ″praznom″ stanju TEST s obzirom na to što se oba odnose na istu promenljivu (registar). Registar *regB* se inkrementira u stanju INCR, dok se registar *regA* pomera za jednu poziciju udesno u stanju SHIFT. Na Sl. 9-32(b) je ilustrovan rad ASMD dijagrama za slučaj kada je ulazni vektor oblika *a*=″0101″. Zapazimo da izračunavanje traje 10 taktnih ciklusa.

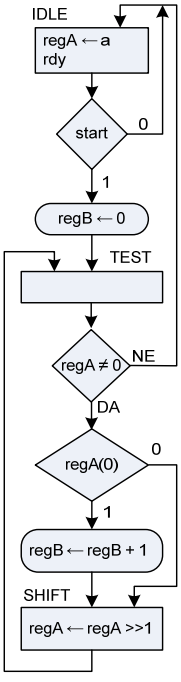


(a)

clk	start	Tekuće stanje	regA	regB	Sledeće Stanje	rdy
0	1	IDLE	xxxx	xx	TEST	1
1	0	TEST	0101	00	INCR	0
2	0	INCR	0101	00	SHIFT	0
3	0	SHIFT	0101	01	TEST	0
4	0	TEST	0010	01	SHIFT	0
5	0	SHIFT	0010	01	TEST	0
6	0	TEST	0001	01	INCR	0
7	0	INCR	0001	01	SHIFT	0
8	0	SHIFT	0001	10	TEST	0
9	0	TEST	0000	10	IDLE	0
10	0	IDLE	0000	10	IDLE	1
11	0	IDLE	xxxx	10	IDLE	1

(b)

Sl. 9-32 Inicijalni ASMD brojača jedinica; (a) ASMD; (b) ilustracija rada za *a*=0101.



(a)

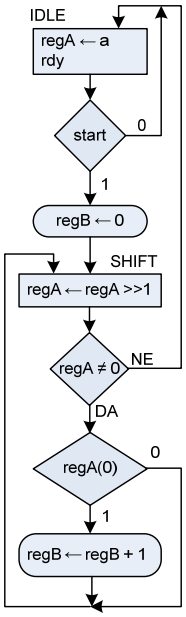
clk	start	Tekuće stanje	regA	regB	Sledeće Stanje	rdy
0	1	IDLE	xxxx	xx	TEST	1
1	0	TEST	0101	00	SHIFT	0
2	0	SHIFT	0101	01	TEST	0
3	0	TEST	0010	01	SHIFT	0
4	0	SHIFT	0010	01	TEST	0
5	0	TEST	0001	01	SHIFT	0
6	0	SHIFT	0001	10	TEST	0
7	0	TEST	0000	10	IDLE	0
8	0	IDLE	0000	10	IDLE	1
9	0	IDLE	xxxx	10	IDLE	1

(b)

Sl. 9-33 ASMD sa izbačenim stanjem INCR: (a) ASMD; (b) ilustracija rada za *a*=0101.

Prva optimizacija koja se može učiniti u polaznom ASMD dijagramu brojača jedinica sastoji se u izbacivanju stanja INCR i pridruživanju naredbe *regB* ← *regB* + 1 stanju TEST (Sl. 9-33(a)). Ova manipulacija je dozvoljena zato što vrednost registra *regB* nema uticaja na ishod ispitivanja u stanju TEST. Uočimo da je u ASMD dijagramu sa Sl. 9-33(a) operacija inkrementiranja registra *regB* napisana u zaobljenom pravougaoniku, tj. tretira se kao uslovna naredba dodele koja se izvršava u stanju TEST, ali samo **pod uslovom** ako važi *regA≠0* i *regA(0)=´* 1´. Eliminisanjem jednog stanja iz petlje, rad ASMD dijagrama je ubrzan tako da sada brojanje 1-ca u vektoru ″0101″ traje 8 taktnih ciklusa, kao što se to može videti u tabeli sa Sl. 9-33(b).

Druga optimizacija brojača jedinica odnosi se na eliminaciju stanja SHIFT iz ASMD dijagrama sa Sl. 9-33(a). To se postiže prebacivanjem naredbe registarskog prenosa *regA* ← *regA* << 1 u stanje TEST, koje je sada nazvano SHIFT (Sl. 9-34(a)). Na prvi pogled, čini se da ova manipulacija remeti tok izvršenja algoritma, jer se u istom stanju sadržaj registra *regA* i modifikuje i ispituje. Međutim, ova nedoslednost nema uticaja na konačni rezultat. Prvo, kad se u stanje SHIFT ulazi iz stanja IDLE, u stanju SHIFT se za ispitivanje uslova koristi stara vrednost registra *regA*, tj. *a*, baš kao u ASMD dijagramu sa Sl. 9-33(a). Drugo, u poslednoj iteraciji petlje, tj. onda kada registar *regA* ima vrednost ″sve nule″, iniciraće se još jedno, nepotrebno pomeranje udesno sadržaja ovog registra. Međutim, eventualna pogrešna vrednost u registru *regA* na kraju izračunavanja nije od značaja budući da se rezultat nalazi u registru *regB*. U svakom slučaju, pomeranje udesno vrednosti ″sve nule″ kao rezultat daje takođe ″sve nule″. S obzirom na to što je sada broj stanja (taktnih ciklusa) po jednoj iteraciji smanjen na samo jedno stanje, brzina rada kola je značajno povećana. Kao što se može videti iz table sa Sl. 9-34(b), za prebrojavanje 1-ca u vektoru ″0101″ sada je potrebno 5 taktnih ciklusa, u odnosu na 8 iz prethodne varijante ASMD dijagrama.

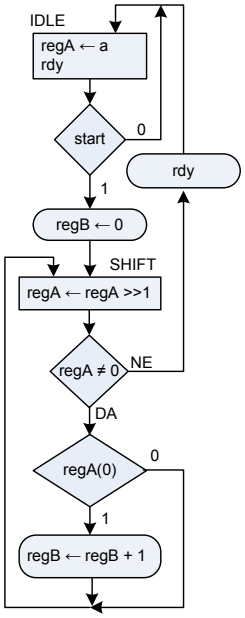


(a)

clk	start	Tekuće stanje	regA	regB	Sledeće Stanje	rdy
0	1	IDLE	xxxx	xx	SHIFT	1
1	0	SHIFT	0101	00	SHIFT	0
2	0	SHIFT	0010	01	SHIFT	0
3	0	SHIFT	0001	01	SHIFT	0
4	0	SHIFT	0000	10	IDLE	0
5	0	IDLE	0000	10	IDLE	1
6	0	IDLE	xxxx	10	IDLE	1

(b)

Sl. 9-34 ASMD sa izbačenim stanjem SHIFT: (a) ASMD; (b) ilustracija rada za *a*=0101.



(a)

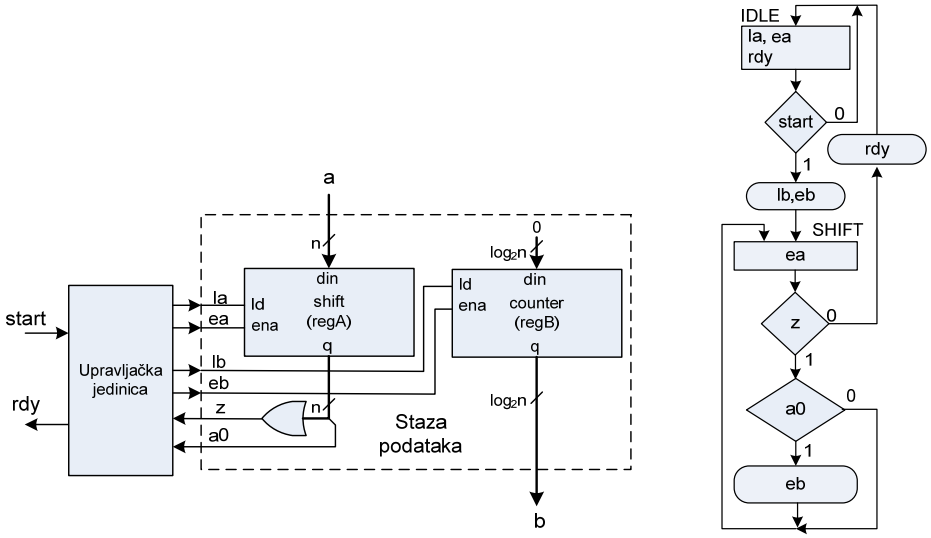
clk	start	Tekuće stanje	regA	regB	Sledeće Stanje	rdy
0	1	IDLE	xxxx	xx	SHIFT	1
1	0	SHIFT	0101	00	SHIFT	0
2	0	SHIFT	0010	01	SHIFT	0
3	0	SHIFT	0001	01	SHIFT	0
4	0	SHIFT	0000	10	IDLE	1
5	0	IDLE	0000	10	IDLE	1

(b)

Sl. 9-35 Konačna verzija ASMD dijagrama brojača jedinica; (a) ASMD; (b) ilustracija rada za *a*=″0101″.

Poslednja u nizu optimizacija ASMD dijagrama brojača jedinica tiče se ranijeg postavljanja izlaznog statusnog signala *rdy*. Iz tabele sa Sl. 9-34(b) vidimo da se indikacija završenog izračunavanja (*rdy* = ´1´) postavlja u 5. taktnom ciklusu, iako je konačan rezultat dostupan već u 4. taktnom ciklusu. To znači da se ukupno vreme izračunavanja može smanjiti za jedan taktni ciklus, ako se signala *rdy* aktivira u stanju SHIFT u toku poslednje iteracije petlje, tj. onda kad je uslov *regA=0* ispunjen (Sl. 9-35(a)). Tabela sa Sl. 9-35(b) ilustruje rad konačne varijante ASMD dijagrama za ulazni vektor ″0101″. Treba napomenuti da je značajno ubrzanje rada u odnosu na polazni ASMD dijagram ostvareno zahvaljujući maksimalno ispoljenom paralelizmu u izvršenju operacija obuhvaćenih petljom.

Razrada. ASMD dijagram brojača jedinica sadrži dve promenljive, *regA* i *regB* kojima će u stazi podataka odgovarati dve registarske komponente (v. Sl. 9-31(a)). Za manipulaciju promenljivom *regA* može se koristiti pomerački registar sa pomeranjem udesno (zbog operacije *regA*←*regA*>>1 u stanju SHIFT). Zbog operacije *regA*←*a* iz stanja IDLE, ovaj registar treba da poseduje mogućnost paralelnog upisa. S druge strane, za *regB* potreban je brojač (zbog *regB*←*regB*+1 iz bloka uslovnog izlaza u stanju SHIFT). Dodatno, ovaj brojač treba da ima i mogućnost sinhronog resetovanja (zbog *regB*←0 u stanju IDLE). Takođe, obe registarske komponente moraju imati ulaz za dozvolu rada, kako bi njihov rad mogao da se onemogući u stanjima u kojima se ne obavljaju njihove operacije. Blok dijagram sekvencijalnog brojača jedinica dobijen razradom ASMD dijagram sa Sl. 9-35(a) prikazan je na Sl. 9-36(a). Strukturu čine upravljačka jedinica i staze podataka koja sadrži dve opisane registarske komponente.



(a) (b)

Sl. 9-36 Sekvencijalni brojača jedinica u fazi razrade: (a) strukturni dijagram; (b) ASM dijagram upravljačke jedinice.

Upravljački ulaz pomeračkog registra (komponenta *shift*) su: *ld* - dozvola paralelnog upisa i *ena* dozvola rada. Za *ld*=*ena*=´1´, ulazni vektora *a* se upisuje u pomerački registar, dok se za *ld*=´0´ i *ena*=´1´, sadržaj registra pomera za jednu poziciju udesno s upisom nule na krajnju levu bitsku poziciju. Brojač (komponenta *counter*) poseduje istoimene upravljačke ulaze. Za *ld*=*ena*=´1´, u brojač se paralelno upisuju ″sve nule″, koje su fiksno postavljene na njegovom ulazu *din*. Na taj način, realizovana je funkcija sinhronog resetovanja. Za *ld*=´0´ i *ena*=´1´, sadržaj brojača se uvećava za 1. Izlaz brojača je ujedno i izlaz *b* sekvencijalnog brojača jedinica. Za testiranje uslova *regA*=0, koristi se *n*-to ulazno NILI kolo. Izlaz NILI kola, *z*, imaće vrednost *z*=´1´ samo ako pomerački registar sadrži sve nule (tj. ako važi *regA=0*). Za spregu s upravljačkom jedinicom, osim *z*, koristi se i krajnji desni bit pomeračkog registra, označene kao *a0*. Radi jednostavnosti prikaza, izostavljeni su signali takta, *clk*, i asinhronog resetovanja, *rst,* koji pobuđuju upravljačku jedinicu i obe registarske komponente.

Nakon što smo precizno opisali stazu podataka, u mogućnosti smo da kreiramo ASM dijagram upravljačke jedinice (Sl. 9-36(b)). Ovaj dijagram ima identičnu strukturu kao ASMD dijagram sa Sl. 9-35(a), s tom razlikom što su naredbe registarskog prenosa i relacije zamenjene naredbama postavljanja upravljačkih signala i ispitivanjima statusnih signala staze podataka. Na primer, blok stanja SHIFT sada sadrži samo ime signala *ea*. To znači da će u ovom stanju važiti *ea*=´1´ i *la*=*eb*=*lb*=´0´. Pod ovim uslovima, pomerački registar obavlja pomeranje udesno, dok sadržaj brojača ostaje nepromenjen, što ima isti efekat kao naredba registarskog prenosa *regA*←*regA*>>1 sadržana u istoimenom stanju iz ASMD dijagrama. Blok grananja koji sledi neposredno posle bloka stanja SHIFT, u ASMD dijagramu sadrži ispitivanje *regA*=0. U ASM dijagramu upravljačke jedinice, ovo ispitivanje je zamenjeno testiranjem statusnog signala staze podataka, *z*.

Realizacija. U nastavku je dat kompletan VHDL kôd projekta sekvencijalnog brojača jedinica, pod pretpostavkom da je ulaz port *a* 8-bitni, a izlaz port *b* 3-bitni. Datoteka *datapath.vhd* sadrži opis staze podataka. Kôd sadrži dva procesa. Prvi proces opisuje rad

pomeračkog registra (linije 19-32), a drugi brojača (linije 34-47). Napomenimo da su opisi pomeračkog registra i brojača kreirani shodno njihovim specifikacijama koje potiču iz faze razrade. Konkurentna naredba *when* iz linije 49 postavlja statusni signal *z* ako je sadržaj registra *regA* jednak ″sve nule″. Sledeći deo VHDL kôda, *control.vhd*, opisuje upravljačku jedinicu na osnovu ASM dijagrama sa Sl. 9-36(b). Ovaj opis je usklađen sa dvosegmentnim kodnim šablonom za konačne automate. Konačno, u vršnom modulu, *ones_counter.vhd*, komponente *datapath* i *control* su instancirane i međusobno povezane..

```
1  -------datapath.vhd ------------------------------------------------
2  LIBRARY IEEE;
3  USE IEEE.STD_LOGIC_1164.ALL;
4  USE IEEE.NUMERIC_STD.ALL;;
5  ENTITY datapath IS
6     PORT(a : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
7          b : OUT STD_LOGIC_VECTOR(3 DOWNTO 0);
8          la,ea,lb,eb : IN STD_LOGIC;
9          a0,z : OUT STD_LOGIC;
10         clk : IN STD_LOGIC;
11         rst : IN STD_LOGIC);
12 END datapath;
13 ---------------------------------------------------------------------
14 ARCHITECTURE datapath OF datapath IS
15   SIGNAL regA : STD_LOGIC_VECTOR(7 DOWNTO 0);
16   SIGNAL regB : UNSIGNED(3 DOWNTO 0);
17 BEGIN
18  -- regA - pomeracki registar ----------------------------------
19  PROCESS (clk,rst)
20  BEGIN
21     IF(rst='1')THEN
22        regA <=(OTHERS => '0');
23     ELSIF(clk'EVENT AND clk='1') THEN
24        IF(ea ='1') THEN
25           IF(la = '1') THEN
26              regA <= a;
27           ELSE
28              regA <= '0' & regA(7 DOWNTO 1);
29           END IF;
30        END IF;
31     END IF;
32  END PROCESS;
33  -- regB - brojac ---------------------------------------------
34  PROCESS (clk,rst)
35  BEGIN
36     IF(rst='1')THEN
37        regB <=(OTHERS => '0');
38     ELSIF(clk'EVENT AND clk='1') THEN
39        IF(eb ='1') THEN
40           IF(lb = '1') THEN
41              regB <= (OTHERS => '0');
42           ELSE
43              regB <= regB + 1;
44           END IF;
45        END IF;
46     END IF;
47  END PROCESS;
```

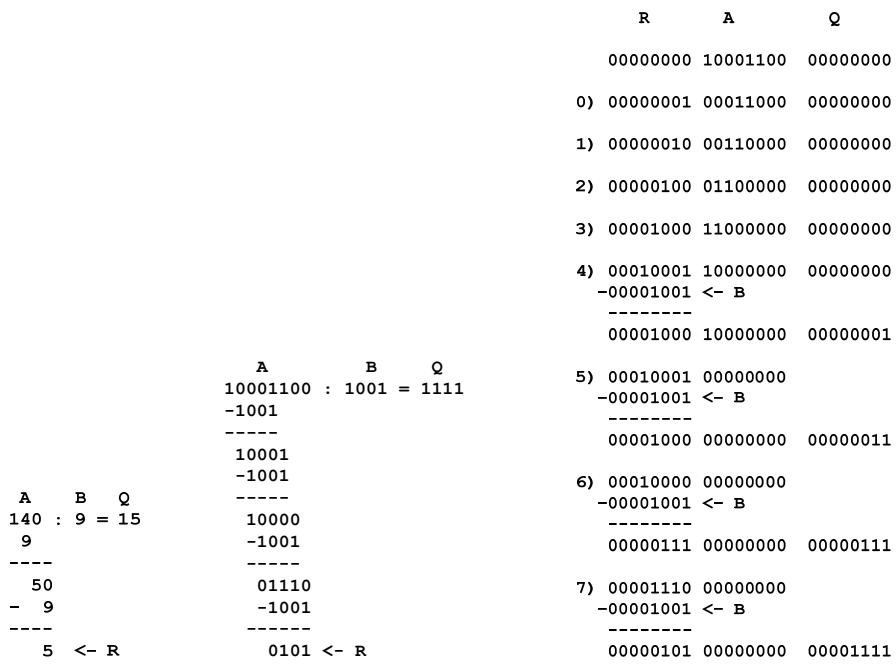
```
 b <= STD_LOGIC_VECTOR(regB);
 z <= '0' WHEN regA = "00000000" ELSE '1';
 a0 <= regA(0);
END datapath;
---- control.vhd -----------------------------------------------
IBRARY IEEE;
SE IEEE.STD_LOGIC_1164.ALL;
NTITY control IS
 PORT(start: IN STD_LOGIC;
      rdy : OUT STD_LOGIC;
      z, a0 : IN STD_LOGIC;
      la,ea,lb,eb : OUT STD_LOGIC;
      clk : IN STD_LOGIC;
      rst : IN STD_LOGIC);
END control;
-----------------------------------------------------------------
ARCHITECTURE control OF control IS
  TYPE state IS (IDLE, SHIFT);
  SIGNAL pr_state, nx_state : state;
BEGIN
 -- registar stanja ------------------------------------------
 PROCESS (clk,rst)
   BEGIN
     IF(rst='1')THEN
        pr_state <= IDLE;
     ELSIF(clk'EVENT AND clk='1') THEN
        pr_state <= nx_state;
     END IF;
  END PROCESS;
  -- logika sledeceg stanja/izlaza ----------------------------
  PROCESS(pr_state,start,z,a0)
  BEGIN
     nx_state <= pr_state;
     la <= '0'; ea <= '0';
     lb <= '0'; eb <= '0';
     rdy <= '0';
     CASE pr_state IS
        WHEN IDLE =>
           rdy <= '1';
           la <= '1';
           ea <= '1';
           IF(start = '1') THEN
              lb <= '1';
              eb <= '1';
              nx_state <= SHIFT;
           END IF;
        WHEN SHIFT =>
           ea <= '1';
           IF(z = '0') THEN
              rdy <= '1';
              nx_state <= IDLE;
           ELSIF(z = '1' AND a0 = '1') THEN
                eb <= '1';
            END IF;
      END CASE;
```

```
103   END PROCESS;
104 END control;
105 -----ones_counter.vhd ----------------------------------------
106 IBRARY IEEE;
107 SE IEEE.STD_LOGIC_1164.ALL;
108 NTITY bit_counter IS
109  PORT(a : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
110       b : OUT STD_LOGIC_VECTOR(3 DOWNTO 0);
111       start : IN STD_LOGIC;
112       rdy : OUT STD_LOGIC;
113       clk : IN STD_LOGIC;
114       rst : IN STD_LOGIC);
115 END bit_counter;
116 ----------------------------------------------------------------
117 ARCHITECTURE bit_counter OF bit_counter IS
118   COMPONENT datapath IS
119     PORT(a : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
120          b : OUT STD_LOGIC_VECTOR(3 DOWNTO 0);
121          la,ea,lb,eb : IN STD_LOGIC;
122          a0,z : OUT STD_LOGIC;
123          clk : IN STD_LOGIC;
124          rst : IN STD_LOGIC);
125   END COMPONENT;
126   COMPONENT control IS
127     PORT(start: IN STD_LOGIC;
128          rdy : OUT STD_LOGIC;
129          z, a0 : IN STD_LOGIC;
130          la,ea,lb,eb : OUT STD_LOGIC;
131          clk : IN STD_LOGIC;
132          rst : IN STD_LOGIC);
133   END COMPONENT;
134   SIGNAL la, ea, lb, eb, z, a0 : STD_LOGIC;
135 BEGIN
136   CTRL: control PORT MAP(start,rdy,z,a0,la,ea,lb,eb,clk,rst);
137   DPHT: datapath PORT MAP(a,b,la,ea,lb,eb,a0,z,clk,rst);
138 END bit_counter;
```

Pr. 9-23 Sekvencijalni delitelj – ubrzani algoritam

U ovom primeru predstavljena je još jedna varijanta sekvencijalnog delitelja, koja, za razliku od ranije razmatranih varijanti ovog kola, umesto uzastopnom oduzimanju delioca od deljenika realizuje jedan brži algoritam, izveden na osnovu tradicionalnog, ″ručnog″ postupku deljenja.

Sl. 9-37(a) prikazuje primer ručnog deljenja decimalnih brojeva 140 i 9. Prvi korak je pokušaj deljenja cifre najveće težine deljenika, 1, deliocem, 9. Pošto se 9 ne sadrži u 1, uzimamo sledeću cifru deljenika, 4, i proveravamo da li se 9 sadrži u 14. Sada je deljenje moguće, što daje prvu cifru količnika 1 (=14/9) i ostatak 5 (=14-9x1). Deljenje nastavljamo tako što s desne strane ostatak dopisujemo sledeću (i poslednju) cifru deljenika, 0, i delimo 50 sa 9. To daje drugu cifru količnika 5 (=50/9) i ostatak 5 (=50-5x9). Time je deljenje završeno. Rezultat je Q=15 (količnik) i R=5 (ostatak). Postupak će biti identičan i ako umesto decimalnih delimo binarne brojevi, s pojednostavljenjem da svaki bit količnika može imati samo dve vrednosti, 0 ili 1 (Sl. 9-37(b)).



(a) (b) (c)

Sl. 9-37 Ilustracija postupka ″ručnog″ deljenja: (a) primer deljenja decimalnih brojeva; (b) primer deljenja binarnih brojeva; (c) modifikovan postupak.

Naš zadatak je da korišćenjem opisanog postupka deljenja projektujemo kolo koje na ulazu dobija dva neoznačena *n*-bitna cela broja, *a* i *b*, i generiše dva *n*-bitna cela broja, *q* i *r*, gde je *q* količnik, a *r* ostatak deljenja *a/b*. Procedura koja je ilustrovana na Sl. 9-37(b) može se sprovesti i na način kao na Sl. 9-37(c), gde je ″spuštanje″ cifara deljenika zamenjeno operacijom pomeranja. Deljenik, sadržan u promenljivoj *A*, proširen je s leve strane ciframa ostatka, *R*, koje su na početku sve nule. U svakom koraku, ostatak i deljenik se zajedno pomeraju za jednu bitsku poziciju ulevo, a zatim se ispituje da li se delilac sadrži u ostatku. Odgovarajući bit količnika dobija vrednost 0 ako deljenje nije moguće. Ako je deljenje moguće, odgovarajući bit količnika dobija vrednost 1, a ostatak se koriguje oduzimanjem delioca. Opisani postupak se ponavlja *N* puta, gde je *N* broj bita operanada, odnosno sve dok svi bitovi deljenika ne napuste promenljivu *A*.

```
R = 0;
for C = 0 to N-1 do
 (R,A) = (R,A) << 1;
 if(R ≥ B) then
  Qc = 1;
  R = R - B;
 else
  Qc = 0;
 end if;
end for;
```

(a)

```
R = 0;
for C = 0 to N-1 do
 (R,A) = (R,A) << 1;
 if(R ≥ B) then
  Q = (Q << 1)1;
  R = R - B;
 else
  Q = (Q << 1)0;
 end if;
end for;
```

(b)

```
R = 0;
C = N-1;
do
 (R,A) = (R,A) << 1;
 if(R ≥ B) then
  Q = (Q << 1)1;
  R = R - B;
 else
  Q = (Q << 1)0;
 end if;
  C = C - 1;
while(C≥0);
```

(c)

Sl. 9-38 Algoritam deljenja: (a) polazni pseudo kôd; (b) pseudo kôd sa pomeranjem količnika; (c) pseudo kôd sa *do-while* petljom.

***Algoritam*.** Algoritam opisanog postupka deljenja dat je u obliku pseudo kôda na Sl. 9-38(a). (R,A) označava $2n$-bitnu promenljivu dobijenu konkatenacijom n-bitnih promenljivih R i A. Petlja se izvršava N puta, tako što se u svakoj iteraciji generiše jedan bit količnika, Q_C, počev od C=0, do C=N-1. Bukvalna interpretacija algoritma značila bi da se u i-toj iteraciji petlje, i-ti bit količnika direktno postavlja na 0 ili 1. Za hardversku realizaciju, pogodnije je umesto direktnog postavljanja bitova koristiti pomeranje količnika Q ulevo sa serijskim upisom vrednosti Q_C. Pseudo kôd s ovom modifikacijom prikazan je na Sl. 9-38(b). Operacija $Q\leftarrow (Q<<1)_1$ označava pomeranje ulevo sa serijskim upisom 1-ce na krajnju desnu poziciju. Slično, operacija $Q\leftarrow (Q<<1)_0$ pomera Q ulevo za jednu poziciju i serijski upisuje 0.

Radi jednostavnije konverzije pseudo kôda u ASMD dijagram, učinećemo još jednu modifikaciju: *for* petlju zamenićemo ekvivalentnom *do while* petljom (Sl. 9-38(c)). Kod *do while* petlje, ispitivanje uslova za kraj petlje je na kraju, a ne na početku petlje, kao što je to slučaj kod *for* varijante. To znači da se telo *do while* izvršava barem jedanput, za razliku od *for* petlje, čije telo može da se izvrši nula ili više puta. Međutim, u konkretnom slučaju ova nedoslednost nije od značaja, zato što postoji prirodno ograničenje da N mora biti veće ili jednako 1. Takođe, zapazimo da se brojač petlje, C, koristi samo za odbrojavanje iteracija, ali ne i kao argument u nekoj drugoj naredbi. Cilj je obezbediti da se petlja izvrši tačno N puta, a da pri tom nije bitno da li se brojač kreće od 0 do N-1 ili od N-1 do 0, kao što je to u *do-while* varijanti.

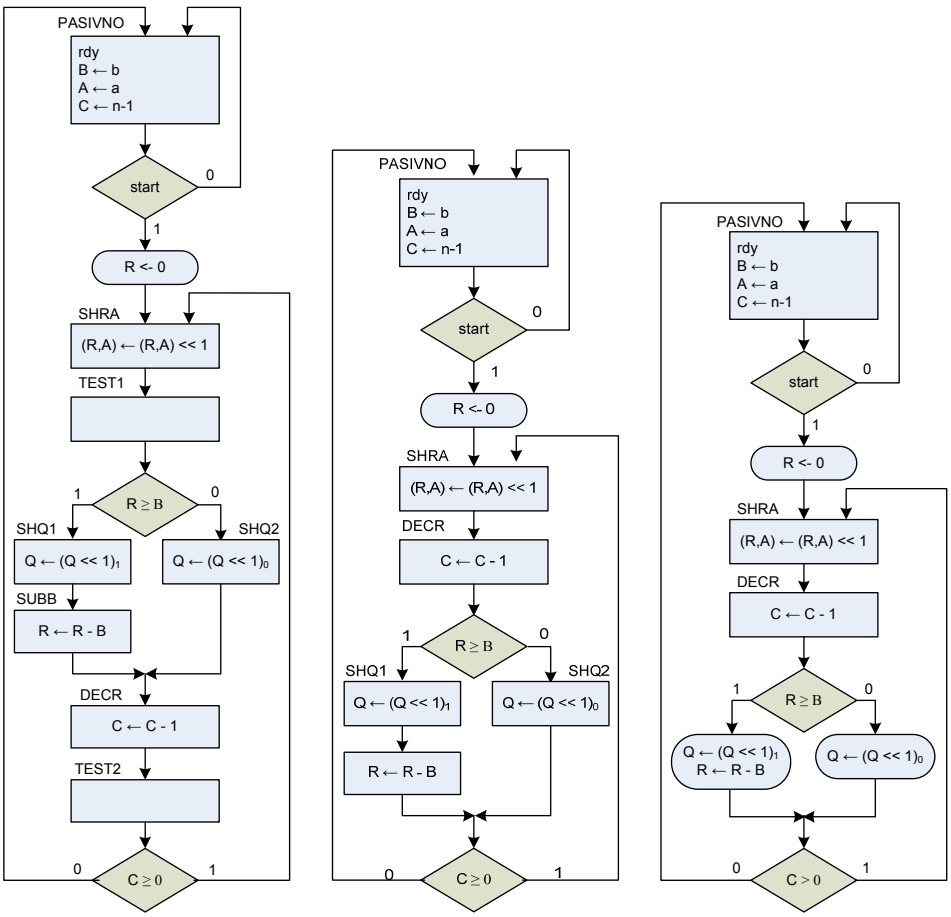
***ASMD dijagram*.** Direktnom konverzijom pseudo kôda sa Sl. 9-38(d), primenom pravila za konverziju iz odeljka 9.2.1, dobijamo ASMD dijagram sa Sl. 9-39(a). Uočimo da ovaj ASMD dijagram dodatno sadrži signale *start* i *rdy*. Ova dva signala su deo interfejsa sekvencijalnog delitelja, koji je identičan kao u Pr. 9-5. Zapazimo da se telo petlje izvršava za 6 ili 5 taktnih ciklusa, u zavisnosti od toga da li je u tekućoj iteraciji R jednako ili veće od B ili ne. Međutim, ovo je tek polazni oblik ASMD dijagrama i treba očekivati da će se njegovom optimizacijom postići značajno poboljšanje.

Prazni blokovi stanja TEST1 i TEST2 uvedeni su iz razloga da bi se prilikom ispitivanja uslova grananja koristile ažurne vrednosti promenljivih. U uslovu grananja se garantovano koristi aktuelna vrednost registra ako se upis u registar i grananje ne vrše u istom stanju. Pri tom nije neophodno da blok stanja koji deli ove dve operacije bude prazan, već je bitno da se u tom stanju ne modifikuje sadržaj registara koji se koristi u uslovu grananja. Imajući to u vidu, oba prazna stanja iz ASMD dijagrama sa Sl. 9-39(a) mogu se eliminisati ako se promeni redosled naredbi u petlji, tako da naredba dekrementiranja brojača petlje pređe s kraj na početak petlje (Sl. 9-39(b)). Razmak između stanja upisa u registar R (stanje SHRA) i grananja $R\geq B$ je očuvan, kao i između stanja DECR, u kome se brojač C dekrementira i ispitivanja uslova $C\geq 0$. Na ovaj način, broj taktnih ciklusa potrebnih za jednu iteraciju petlje smanjen je na 3, odnosno 4.

Sledeća optimizacija tiče se stanja SHQ1, SHQ0 i SUBB. Kao prvo, uočimo da se stanja SHQ1 i SUBB mogu spojiti u jedno stanje (SHQ1SUBB), u kome će se u paraleli izvršavati obe naredbe registarskog prenosa, $Q\leftarrow(Q<<1)_1$ i $R\leftarrow R-B$. Ovo spajanje stanja je dozvoljeno zato što su naredbe $Q\leftarrow(Q<<1)_1$ i $R\leftarrow R-B$ međusobno nezavisne, u tom pogledu da ni jedna od njih ne koristi rezultat one druge. Na ovaj način, broj taktnih ciklusa po jednoj iteraciji postaje 3, bez obzira na ishod testiranja $R\geq B$.

Pripajanjem izračunavanja iz stanja SHQ1SUBB i SHQ0 stanju DECR, odnosno zamenom blokova stanja SHQ1SUBB i SHQ0 blokovima uslovnog izlaza, postigli bismo ubrzanje za

još jedan taktni ciklus po iteraciji. Međutim, s takvom modifikacijom, stanju DECR bilo bi pripojeno i ispitivanje C≥0 s kraja petlje, što bi narušilo neophodan vremenski razmak između ovog ispitivanja i naredbe koja dekrementira brojač *C*. Kao posledica toga, petlja bi imala jedan prolazak više, jer bi se izvršila na samo za *C*=*n*-1, ..., 0, već i za *C*=-1. Međutim, ovaj nedostatak se može lako otkloniti ako se uslov za izlazak iz petlje promeni na C>0. Na taj način dolazimo do ASMD dijagrama sa Sl. 9-39(c) koji zahteva 2 taktna ciklusa po iteraciji. Imajući to u vidu, deljenje *n*-bitnih brojeva trajaće 2*n*+1 taktnih intervala.

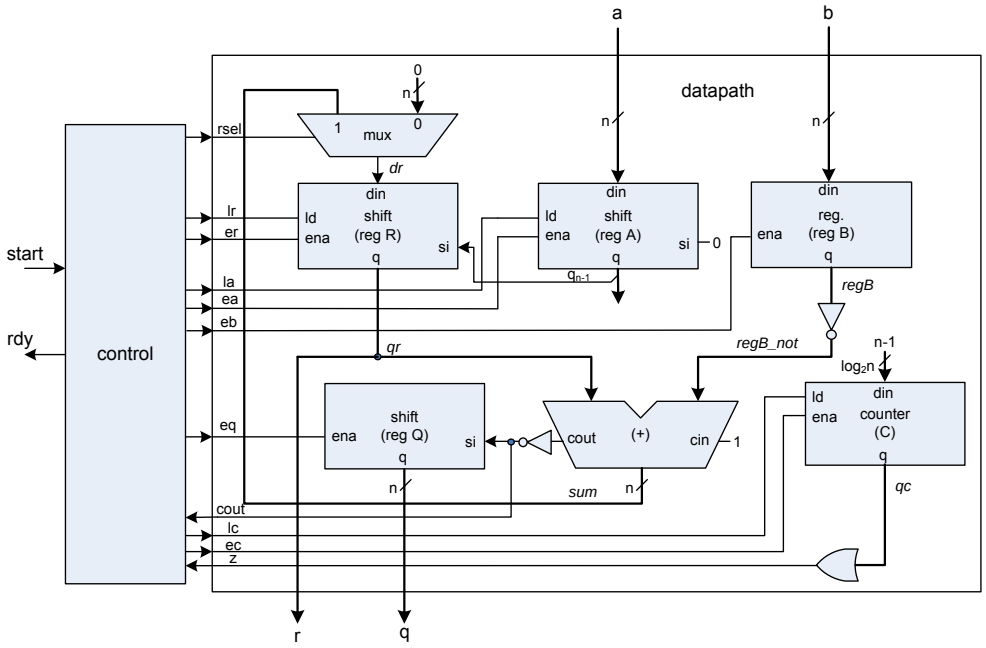


Sl. 9-39 ASMD dijagrami sekvencijalnog delitelja : (a) ASMD dobijen konverzijom softverskog algoritma; (b) ASMD nakon eliminacije stanja TEST1 i TEST2; (c) ASMD nakon spajanja stanja INCR, SHQ1, SUBB i SHQ2.

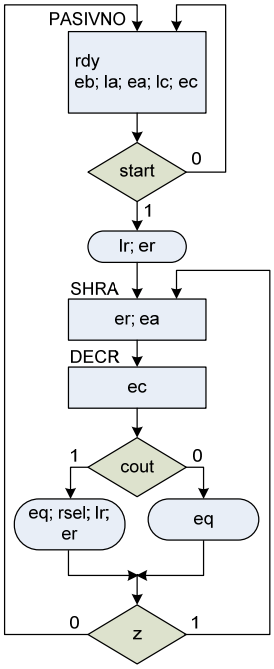
Postavlja se pitanje da li je moguće izračunavanja koja se obuhvaćena petljom obaviti u samo jednom taktnom ciklusu. Međutim, pre nego što damo odgovor na ovo pitanje, usvojimo da je ASMD dijagram sa Sl. 9-39(c) konačna verzija ASMD dijagrama sekvencijalnog delitelja i obavimo njegovu razradu.

Razrada. ASMD dijagram sadrži pet promenljivih: *A, B, Q, R* i *C*. Promenljive *A, B* i *Q*

učestvuju u operacijama pomeranja i zato je normalno da se za njihovu realizaciju upotrebe pomerački registri. Zajedničko pomeranje kroz registre *R* i *A* ostvariće se povezivanjem krajnjeg levog izlaznog bita registra *A* na serijski ulaz registra *R*. Osim u naredbi registarskog prenosa iz stanja SHRA, koja pomera *R* i *A*, promenljiva *A* se javlja kao odredište u još jednoj naredbi, *A*←*a*, iz stanja PASIVNO. Ova dodatna operacija zahteva da pomerački registar *A* poseduje mogućnost paralelnog upisa. Funkciju paralelnog upisa treba da poseduje i pomerački registar *R*. U stanju PASIVNO *R* se resetuje, a u stanju DECR u *R* se upisuje razlika *R*-*B*. Izbor podataka za paralelni upis u registar *R*, 0 ili *R*-*B*, može se ostvariti pomoću multipleksera. Promenljiva *B* se koristi za memorisanje delioca, koji se ne menja tokom izračunavanja. Iz tog razloga, za promenljivu *B* možemo koristiti prihvatni registar. Promenljiva *B*, zajedno s promenljivom *R*, koristi se u naredbi *R*-*B* iz stanja DECR. Za oduzimanje *R*-*B* možemo koristiti *n*-bitni binarni sabirač, pri čemu će promenljiva *B* biti komplementirana, a na ulazni prenos sabirača postavljeno ´1´. Za realizaciju relacije *R*≥*B* nije potrebno dodatno kolo u stazi podataka, pošto informaciju o tome da li je *R* veće ili jednako od *B* imamo na izlazu za izlazni prenos sabirača koji se već koristi za oduzimanje *B* od *R*. Izlazni prenos će imati vrednost ´0´ ako važi *R*≥*B*, odnosno ´1´ ako važi *R*<*B*. Šta više, komplement izlaznog prenosa je upravo ona vrednost koju treba serijski upisati u pomerački registar *Q*, što omogućuje da se izlazni prenos sabirača posredstvom invertora direktno poveže na serijski ulaz pomeračkog registra *Q*. Konačno, promenljiva *C*, koja se koristi kao brojač petlje, može se realizovati pomoću brojača naniže s mogućnošću paralelnog upisa. Pri tom početnu vrednost, *N*-1, treba, kao konstantu, postavlja na paralelne ulaze ovog brojača. Na Sl. 9-40 je prikazana struktura sekvencijalnog delitelja, nakon izvršene razrade. ASM dijagram upravljačke jedinice prikazan je na Sl. 9-41.



Sl. 9-40 Razrađena struktura sekvencijalnog delitelja iz Pr. 9-23.



Sl. 9-41 ASM dijagram upravljačke jedinice sekvencijalnog delitelja iz Pr. 9-23.

***Realizacija*.** Nakon razrade ASMD dijagrama, sledi opis razrađene strukture u VHDL-u. VHDL opisi staze podataka (datoteka *datapath.vhd*) i upravljačke jedinice (datoteka *control.vhd*) dati su u nastavku. U datom kôdu nedostaju opisi komponenti koje se koriste za konstrukciju staze podataka (*shift* - pomerački registar ulevo s paralelnim upisom, *counter* - brojač naniže s paralelnim upisom i *adder* - binarni sabirač), kao i vršni modul koji objedinjuje upravljačku jedinicu i stazu podataka. Zapazimo da se u ovom primeru koristi nešto drugačiji pristup za kreiranje VHDL opisa staze podataka. Umesto da je celokupan opis sadržan u jednoj arhitekturi, koriste se nezavisno kreirani VHDL moduli registarskih komponenti i sabirača. Međutim, iako koristi komponente, VHDL kôd staze podataka nije u potpunosti strukturnog tipa. Primetimo da se za opis prihvatnog registra *B* u arhitektura staze podataka koristi proces; pomerački registri, brojač i sabirač su kreirani instanciranjem odgovarajućih komponenti, dok su multiplekser i invertori opisani konkurentnim kôdom. VHDL opis ASM dijagrama upravljačke jedinice, napisan u skladu sa dvosegmentnim kôdnim šablonom, sadržan je u datoteci *control.vhd.*

```
1  ---- datapath.vhd ------------------------------------------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  ---------------------------------------------------------------------
5  ENTITY datapath IS
6    PORT (a,b : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
7          rsel,lr,er,la,ea,eb,eq,lc,ec,rst,clk : IN STD_LOGIC;
8          cout,z : OUT STD_LOGIC;
9          q,r : OUT STD_LOGIC_VECTOR(7 DOWNTO 0));
10 END datapath;
11 ---------------------------------------------------------------------
12 ARCHITECTURE structure OF datapath IS
```

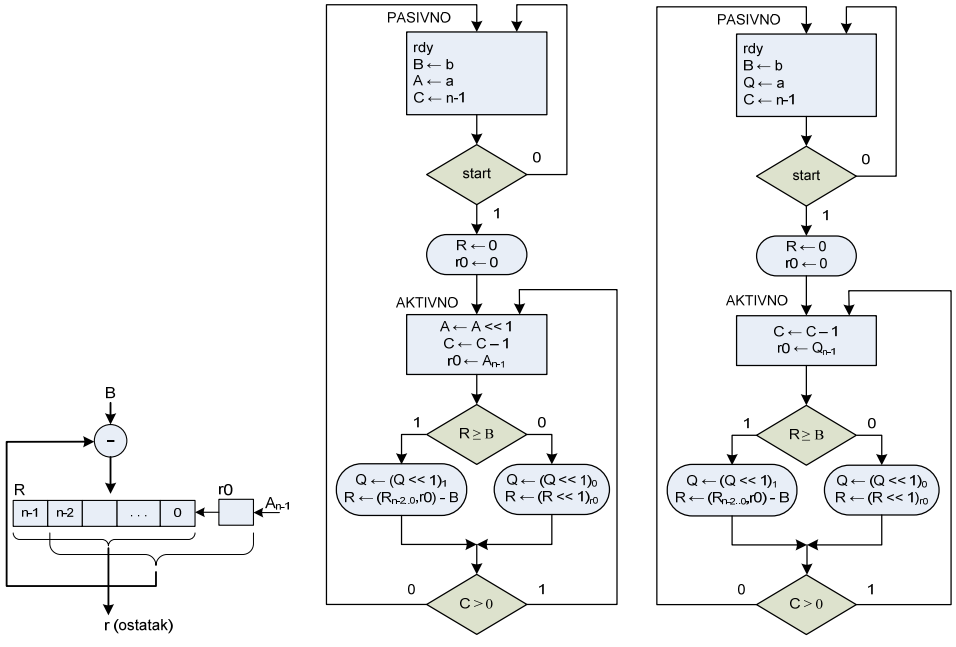
```
  COMPONENT shift IS
     GENERIC (N : INTEGER);
     PORT (si,ld,ena,rst,clk : IN STD_LOGIC;
           din : IN STD_LOGIC_VECTOR(N-1 DOWNTO 0);
           q : OUT STD_LOGIC_VECTOR(N-1 DOWNTO 0));
  END COMPONENT;
  COMPONENT counter IS
     GENERIC (N : INTEGER);
     PORT (ld,ena,rst,clk : IN STD_LOGIC;
           din : IN STD_LOGIC_VECTOR(N-1 DOWNTO 0);
           q : OUT STD_LOGIC_VECTOR(N-1 DOWNTO 0));
  END COMPONENT;
  COMPONENT adder IS
     GENERIC (N : INTEGER);
     PORT (cin : IN  STD_LOGIC;
           x,y : IN  STD_LOGIC_VECTOR(N-1 DOWNTO 0);
           s   : OUT STD_LOGIC_VECTOR(N-1 DOWNTO 0);
           cout: OUT STD_LOGIC);
  END COMPONENT;
  SIGNAL regB : STD_LOGIC_VECTOR(7 DOWNTO 0);
  SIGNAL regB_not : STD_LOGIC_VECTOR(7 DOWNTO 0); -- komplement B
  SIGNAL qa,dr,qr,sum : STD_LOGIC_VECTOR(7 DOWNTO 0);
  SIGNAL qc : STD_LOGIC_VECTOR(2 DOWNTO 0); -- izlaz brojaca C
  SIGNAL co, co_not : STD_LOGIC;
BEGIN
  -- prihvatni registar B ---------------------------------------
  REG_B: PROCESS (clk,rst)
  BEGIN
     IF(rst='1')THEN
        regB <=(OTHERS => '0');
     ELSIF(clk'EVENT AND clk='1') THEN
        IF(eb ='1') THEN
           regB <= b;
        END IF;
     END IF;
  END PROCESS;
  REG_A: shift GENERIC MAP(N=>8)
               PORT MAP(si=>'0',ld=>la,ena=>ea,rst=>rst,
                        clk=>clk,din=>a,q=>qa);
  REG_R: shift GENERIC MAP(N=>8)
               PORT MAP(si=>qa(7),ld=>lr,ena=>er,
                        rst=>rst,clk=>clk,din=>dr,q=>qr);
  REG_Q: shift GENERIC MAP(N=>8)
               PORT MAP(si=>co_not,ld=>'0',ena=>eq,
                       rst=>rst,clk=>clk,
                       din=>(OTHERS=>'0'),q=>q);
  ADD  : adder GENERIC MAP(N=>8)
               PORT MAP(cin=>'1',x=>qr,y=>regB_not,
                        s=>sum,cout=>co);
  CNT  : counter GENERIC MAP(N=>3)
                 PORT MAP(ld=>lc,ena=>ec,rst=>rst,
                          clk=>clk,din=>"111",q=>qc);
  regB_not <= not regB;
  dr <= sum WHEN rsel = '1' ELSE (OTHERS => '0');
  z <= '1' WHEN qc = "000" ELSE '0'; -- multiplekser za reg R
```

```
68   co_not <= not co; -- invertor na izlazu sabiraca
69   cout <= co_not;
70   r<=qr;
71 END structure;
72 -- control.vhd ------------------------------------------------
73 LIBRARY ieee;
74 USE ieee.std_logic_1164.all;
75 ---------------------------------------------------------------------
76 ENTITY control IS
77   PORT (clk,rst,start,cout,z : IN STD_LOGIC;
78         rdy,rsel,la,ea,eb,lc,ec,lr,er,eq : OUT STD_LOGIC);
79 END control;
80 ---------------------------------------------------------------------
81 ARCHITECTURE ASM OF control IS
82   TYPE state IS (PASIVNO, SHRA, DECR);
83   SIGNAL pr_state, nx_state : state;
84 BEGIN
85   -- registar stanja ------------------------------------------
86   PROCESS (clk,rst)
87   BEGIN
88      IF(rst='1')THEN
89         pr_state <=PASIVNO;
90      ELSIF(clk'EVENT AND clk='1') THEN
91         pr_state <= nx_state;
92      END IF;
93   END PROCESS;
94   -- logika sledeceg stanja/izlaza ----------------------------
95   PROCESS(start,z,cout,pr_state)
96    BEGIN
97       nx_state <= pr_state;
98       rdy<='0';la<='0';ea<='0';eb<='0';lc<='0';
99       ec<='0';lr<='0';er<='0';eq<='0';rsel<='0';
100      CASE pr_state IS
101        WHEN PASIVNO =>
102          rdy<='1';eb<='1';la<='1';ea<='1';lc<='1';ec<='1';
103          IF(start = '1') THEN
104            lr<='1'; er<='1';
105            nx_state <= SHRA;
106          END IF;
107        WHEN SHRA =>
108          er<='1';ea <='1';
109          nx_state <= DECR;
110        WHEN DECR =>
111          ec<='1';
112          IF(cout = '1') THEN
113            eq<='1';rsel<='1';lr<='1';er<='1';
114          ELSE
115            eq<='1';
116          END IF;
117          IF(z = '1') THEN
118            nx_state <= PASIVNO;
119          ELSE
120            nx_state <= SHRA;
121          END IF;
122      END CASE;
```

```
123     END PROCESS;
124   END ASM;
```

Pr. 9-24 Sekvencijalni delitelj - dodatna optimizacija

Sekvencijalni delitelj iz Pr. 9-23 obavlja deljenje dva n bitna broja za $2n$ taktnih ciklusa. Razlog za to su dva stanja u unutrašnjoj petlji ASMD dijagrama (SHRA i DECR) (v. Sl. 9-39(c)). Ukoliko bismo uspeli da ova dva stanja sažmemo u jedno, broj potrebnih taktnih ciklusa bio bi smanjen na n. U stanju INCR, pod uslovom da važi $R \geq B$, u R se upisuje razlika R-B, a novo stanje je SHRA. U stanju SHRA, R i A se zajedno pomeraju za jednu poziciju ulevo. Da bismo stanja SHRA i DECR objedinili u jedno stanja (neka se zove AKTIVNO), potrebno je da postoji mogućnost da se u jednom koraku (taktu) na pozicije veće težine registra R upiše razliku i u isto vreme na poziciju najmanje težine registra R prenese vrednost bita najveće težine registra A. To se može postići uvođenjem dodatnog flip-flopa, $r0$, za poziciju najmanje težine registra R (Sl. 9-42(a)). S ovim proširenjem, za izračunavanje razlike koriste se n-1 nižih bita registra R zajedno sa flip-flopom $r0$. Budući da $r0$ sadrži bit najveće težine registra A iz prethodnom ciklusa, efekat je isti kao da je pre oduzimanja obavljeno $(R,A) \leftarrow (R,A) << 1$. Onda kad važi $R < B$, u R se ne upisuje razlika već se R pomera ulevo sa serijskim upisom vrednosti iz $r0$. Ostatak deljenja (r) jednak je razlici koja je se u poslednjem ciklusu upisuje u registar R - zbog toga je neophodno da R ima svih n bita. ASMD dijagram sekvencijalnog delitelja s jednim ciklusom po iteraciji prikazan je na Sl. 9-42(b).



(a) (b) (c)

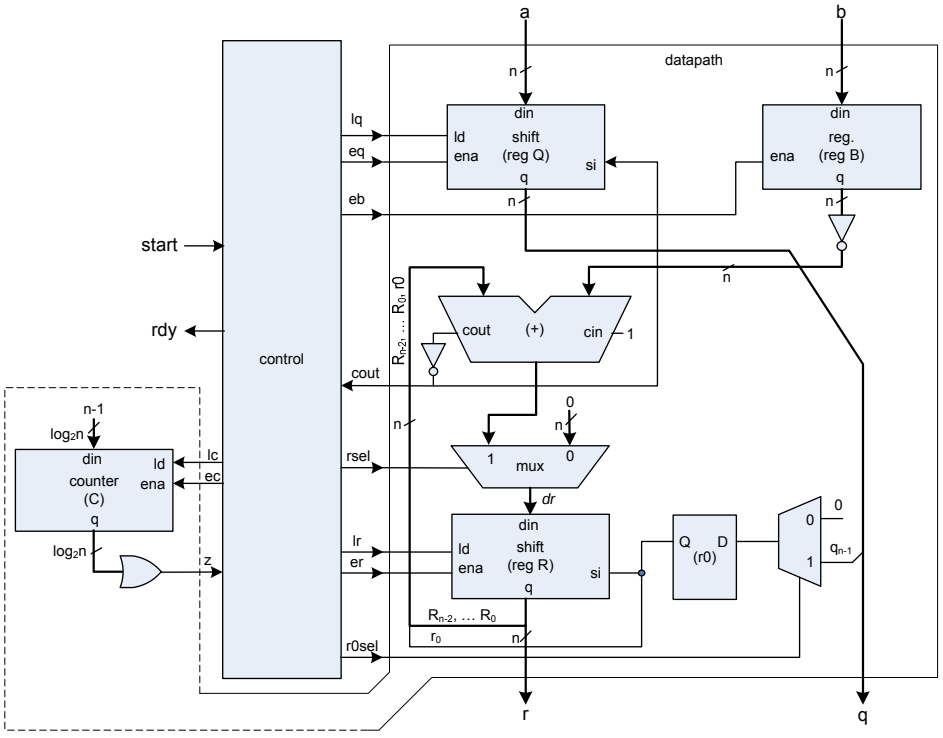
Sl. 9-42 Dodatno ubrzanje sekvencijalnog delitelja iz Pr. 9-23: (a) princip; (b) ASMD (prva varijanta); (c) ASMD (druga varijanta, sa objedinjenim registrima A i Q).

Dodatna optimizacija, u pogledu uštede hardvera, može se postići ukoliko se za promenljive A i Q, umesto dva, iskoristi jedan, zajednički registar. Promenljiva A iz ASMD

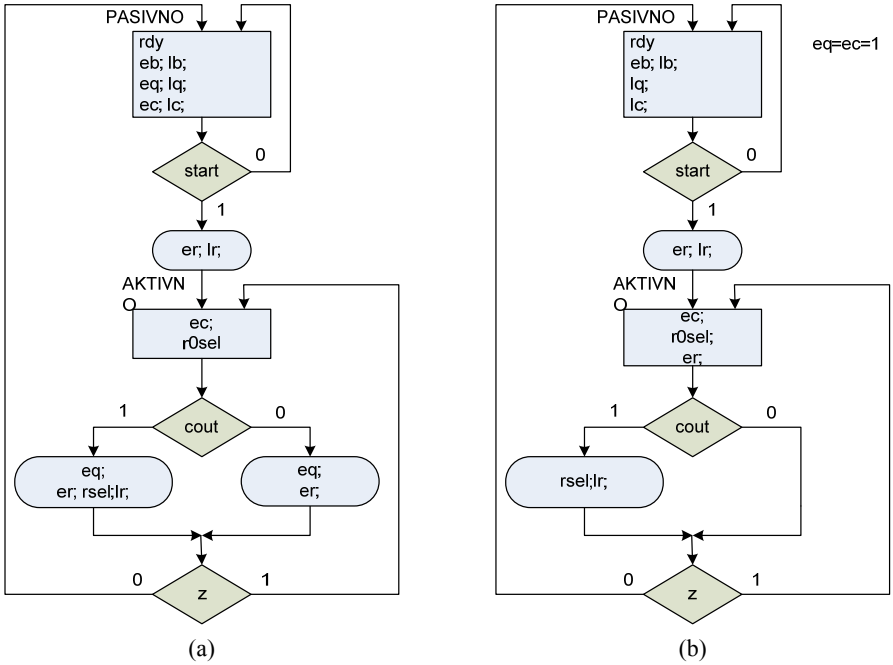
dijagrama sa Sl. 9-42(b), na početku rada, u stanju PASIVNO, dobija vrednost deljenika, *a*, a zatim se pomera ulevo s upisom ´0´ na poziciju najmanje težine (operacija $A \leftarrow A << 1$ iz stanja AKTIVNO). Na kraju deljenja, posle *n* taktnih ciklusa, sadržaj registra *A* biće jednak svim nulama. S druge strane, u registar *Q* se serijski upisuje jedan bit količnika u svakom ciklusu deljenja, tako da na kraju deljenja *Q* sadrži kompletan količnik. Na Sl. 9-42(c) je prikazan ASMD dijagram sekvencijalnog delitelja u kome se za promenljive *A* i *Q* koristi isti registar *Q*. Na početku, u *Q* se upisuje deljenik, *a*. Kako deljenje odmiče, tako se iz *Q* istiskuje *a*, a upisuje količnik.

Strukturni blok dijagram sekvencijalnog delitelja, kreiran na osnovu ASMD dijagrama sa Sl. 9-42(c), prikazan je na Sl. 9-43. Za realizaciju registra *Q* koristi se pomerački registar s mogućnošću paralelnog upisa. Za *r*0 se koristi D flip-flop s dodatnim multiplekserom posredstvom kojeg se u flip-flop upisuje ´0´ (u stanju PASIVNO), odnosno bit najveće težine registra *Q* (u stanju AKTIVNO). Na Sl. 9-44(a) je prikazan ASM dijagram upravljačke jedinice, koji je kreiran na osnovu ASMD dijagrama sa Sl. 9-42(c) i strukture sa Sl. 9-43. Napomenimo da su i dalje moguća izvesna pojednostavljenja ovog ASM dijagrama. Zapazimo da su upravljački signali *eq* i *ec* aktivni u oba stanja, PASIVNO i AKTIVNO, i da zbog toga mogu biti postavljeni fiksno na *eq*=*ec*=´1´, umesto da se generišu iz upravljačke jedinice. Takođe, signal *er* je aktivan u obe grane uslovnog koraka iz stanja AKTIVNO i zbog toga može biti prebačen u blok stanja AKTIVNO. Na Sl. 9-44(b) je prikazan ASM dijagram upravljačke jedinice nakon izvršenih modifikacija.

Čitaocu se prepušta da samostalno kreira odgovarajući VHDL opis.



Sl. 9-43 Razrađena struktura sekvencijalnog delitelja iz Pr. 9-24.



Sl. 9-44 ASM dijagram upravljačke jedinice sekvencijalnog delitelja iz Pr. 9-24: (a) početna varijanta; (b) pojednostavljena varijanta.

10. LITERATURA

[1] Pong P. Chu, *RTL Hardware Design Using VHDL – Coding for Efficiency, Portability, and Scalability*, John Wiley & Sons, Inc., 2006.

[2] Volnei A. Pedroni, *Circuit Design with VHDL*, MIT Press, 2004.

[3] Douglas L. Perry, *VHDL Programming by Examples*, *Forth Edition*, McGrow-Hill, 2002.

[4] Parag K. Lala, *Principles of Modern Digital Design*, John Wiley & Sons, Inc., 2007.

[5] Peter J. Ashenden, *The Designer's Guide to VHDL, (Systems on Silicon*), Morgan Kaufmann Publishers Inc., 3rd edition, 2008.

[6] Pong P. Chu, *FPGA Prototyping by VHDL Examples, Xilinx Spartan-3 Version*, John Wiley & Sons, Inc., 2008.

[7] Stephen Brown, Zvonko Vranesic, *Fundametals of Digital Logic with VHDL Desing*, McGraw-Hill Higher Education; 3rd edition, 2008.

[8] John F. Wakerly, *Digital Design: Principles and Practices*, Prentice Hall; 4th edition, 2005.

[9] Rendy H. Katz, Gaetano Borriello, *Contemporary Logic Design*, Pearson Education Inc.,; 2nd edition, 2005.

[10] Mark Gordon Arnold, *Verilog Digital Computer Design – Algorithms into Hardware*, Prentice Hall PTR, 1999.

[11] Daniel D. Gajski, *Principles of Digital Design*, Prentice-Hall International, Inc., 1997.
