2.

Hijerarhijski i parametrizovani dizajn

Složenost savremenih digitalnih sistema je tolika da klasičan, centralizovan, pristup njihovom projektovanju rezultuje u neprihvatljivo dugačkom vremenu razvoja. Kako bi se povećala produktivnost i skratilo vreme razvoja novog digitalnog sistema, dizajneri se danas oslanjaju na tri tehnike:

- hijerarhijski, modularni dizajn,
- ponovno korišćenje prethodno razvijenih modula (*Design Reuse*),
- parametrizovani dizajn.

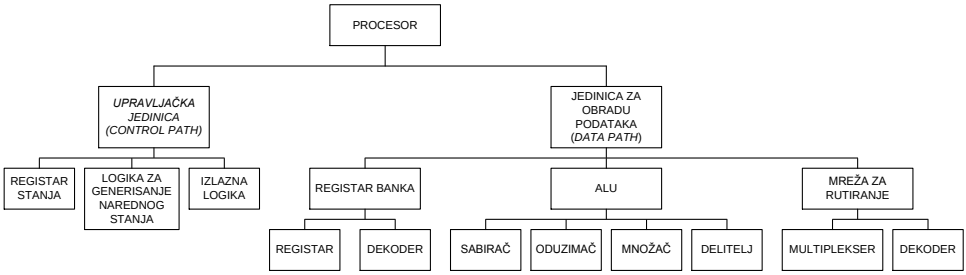
Navedene tehnike omogućavaju značajno povećanje produktivnosti i skraćivanje vremena potrebnog za projektovanje novog digitalnog sistema, oslanjajući se na podelu i paralelizaciju posla, kao i na ponovno korišćenje ranije razvijenih HDL modela.

2.1. Hijerarhijski dizajn

Pod hijerarhijskim (modularnim) dizajnom podrazumevamo hijerarhijsku dekompoziciju složenog sistema na skup međusobno povezanih modula. Svaki od ovih modula ima svoj jasno definisani interfejs, način kako se povezuje i komunicira sa svojim okruženjem, kao i funkciju koju treba da ostvaruje. Vrlo često svaki od ovih

modula ima toliko složenu strukturu da se proces modularizacije može ponovo primeniti. Rekurzivnom primenom modularizacije dobijamo hijerarhijsku predstavu strukture sistema koji projektujemo. Ovo je pristup koji je generalan, u smislu da se ne primenjuje samo prilikom projektovanja složenih digitalih sistema, već se primenjuje gotovo u svakom inženjerskom poduhvatu (prilikom razvoja softvera, fabrikacije mašina, izgradnje infrastrukturnih objekata, itd.). Hijerarhijska dekompozicija je trenutno najbolji odgovor na problem efikasnog projektovanja složenih sistema.

U slučaju projektovanja složenih digitalnih sistema, proces modularizacije obično se zaustavlja kada dođemo do modula čija se funkcionalnost može opisati korišćenjem neke od standardnih kombinacionih i sekvencijalnih mreža. Na kraju procesa modularizacije, sistem je predstavljen hijerarhijskom strukturom modula koja se može predstaviti strukturom obrnutog stabla, prikazanog na slici 2.1. Na slici 2.1 prikazana je jedna moguća hijerarhijska dekompozicija hipotetičkog mikroprocesora.



Slika 2.1. Hijerarhijska dekompozicija hipotetičkog procesora

Na vrhu stabla nalazi se jedan čvor, koren, koji predstavlja čitav sistem koji se projektuje. U našem slučaju to bi bio ceo mikroprocesor koji projektujemo. Sledeći nivo čvorova predstavlja module čijim međusobnim povezivanjem nastaje čitav sistem koji želimo da isprojektujemo. U našem slučaju sledeći nivo u hijerahijskog strukturi čine dva modula, upravljačka jedinica (*Control Path*) i jedinica za obradu podataka (*Data Path*). Ukoliko je neki od ovih modula složen (ne predstavlja standardnu kombinacionu ili sekvencijalnu mrežu), on se dalje dekomponuje na podmodule, kao što je prikazano na slici 2.1. Na primer, jedinicu za obradu podataka možemo dalje dekomponovati na:

- registarsku banku – u kojoj će se nalaziti svi unutrašnji registri procesora,
- aritmetičko logičku jedinicu (ALU) – koja je zadužena za izvođenje aritmetičkih i logičkih operacija nad podacima,
- mrežu za rutiranje – koja je zadužena za prenos podataka od registarske banke do ALU i obrnuto, kao i za prenos podataka od i ka procesoru.

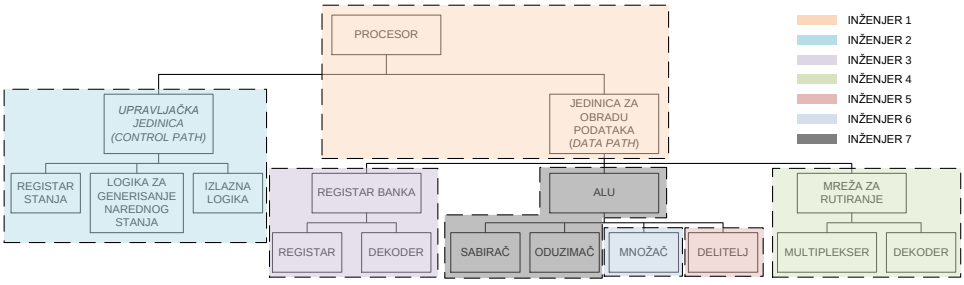
Kao što je ranije napomenuto, proces modularizacije individualnih modula završava se kada dođemo do nivoa standarnih kombinacionih i sekvencijalnih mreža, što se takođe

jasno vidi na slici 2.1. Na primer, proces dekompozicije ALU modula završava se kada stignemo do nivoa standardnih kombinacionih mreža za realizaciju osnovnih aritmetičkih i logičkih operacija (sabirač, oduzimač, množač, delitelj).

Na koji način modularizacija povećava produktivnost i skraćuje vreme potrebno za razvoj novog digitalnog sistema?

Prvi način ogleda se u mogućnosti paralelizacije procesa projektovanja, u trenutku kada smo sistem modularizovali i predstavili pomoću hijerarhijske strukture. Na kraju procesa modularizacije svaki od modula ima jasno definisan interfejs i jasno definisanu funkcionalnost koju treba da implementira. Dizajn svakog od ovih modula može se sada poveriti jednom inženjeru ili timu inženjera, u zavisnosti od složenosti modula. Proces projektovanja različitih modula može se izvoditi u paraleli, uključujući i pisanje modela modula na različitim nivoima hijerahije. Ovo je moguće zbog toga što su funkcionalnost ili struktura, kao i interfejs svakog modula dobro definisani i sve informacije koje su neophodne za njegov razvoj su na raspolaganju dizajneru. Za svaki modul jasno je kako treba da izgleda njegov interfejs, šta treba da bude njegova funkcionalnost i kakva je njegova unutrašnja struktura (od kojih podmodula se sastoji i kako su oni međusobno povezani). Ovo su sve informacije koje su potrebne dizajneru kako bi pristupio procesu modelovanja. Na slici 2.2 prikazana je jedna moguća paralelizacija procesa projektovanja mikroprocesora sa hijerarhijskom strukturom sa slike 2.1.

Na slici 2.2 prikazana je moguća paralelizacija procesa projektovanja mikroprocesora u sedam paralelnih procesa. Za svaki od paralelnih procesa zadužen je po jedan inženjer. Ukoliko pretpostavimo da je vreme potrebno za kompletiranje svakog od sedam paralelnih procesa izbalansirano (podjednako), onda je paralelizacijom procesa projektovanja mikroprocesora ostvareno njegovo ubrzanje od približno sedam puta. Naravno, u praksi je prilično teško ostvariti ovakvu idealnu paralelizaciju posla, tako da je krajnji efekat paralelizacije obično manji. Na primer, u našem primeru nerealno je da proces projektovanja mreže za rutiranje zahteva istu količinu posla kao proces projektovanja upravljačke jedinice.



Slika 2.2. Paralelizacija procesa projektovanja procesora

Za sada na žalost ne postoji opšti postupak modularizacije hardverskog sistema koja bi rezultovala u optimalnoj hijerarhijskoj dekompoziciji sistema. Jedan od razloga zbog

čega to slučaj je raznolikost kriterijuma koje bi jedna takva optimalna dekompozicija trebala da zadovolji. Kao što je poznato, prilikom projektovanja digitalnih računarskih sistema moguće je definisati različite kriterijume (ograničenja) koja jedan takav sistem treba da zadovolji. Ova ograničenja mogu se odnositi na:

- potrebnu funkcionalnost koju projektovani sistem treba da realizuje,
- cenu realizovanog sistema,
- performanse koje sistem treba da dostigne,
- fizičku veličinu sistema,
- potrošnju električne energije,
- pouzdanost,
- mogućnost verifikacije, itd.

Ispunjavanje svakog od ovih ograničenja po pravilu rezultuje u drugačijoj hijerarhijskoj dekompoziciji sistema.

Iako trenutno ne postoji opšti postupak za optimalnu modularizaciju hardverskog sistema prema zadatim ograničenjima, ipak postoji skup pravila, odnosno preporuka, kojih bi se trebalo pridržavati prilikom particionisanja dizajna.

Particionisanje na sistemskom nivou

Na najvišem hijerarhijskom nivou potrebno je odlučiti da li će kompletna funkcionalnost biti implementirana direktno u hardveru, ili će deo funkcionalnosti biti implementiran u softveru, koji će biti izvršavan na jednom ili više procesora.

Generalno pravilo na ovom nivou hijerarhije je da se sva funkcionalnost koju je moguće implementirati u softveru i implementira u softveru. Ovo je zbog toga što je razvoj i održavanje softvera daleko jednostavnije i, što je još važnije, jefinitinije od razvoja i održavanja hardvera. Ovo je upravo razlog zašto je većina savremenih digitalnih elektronskih sistema bazirana na korišćenju jednog ili više procesora. Međutim, u nekim aplikacijama, gde se zahtevaju visoke performanse ili je potrebno ostvariti vrlo nisku potrošnju električne energije, procesorski bazirani sistemi se ne mogu koristiti. U ovom slučaju potrebna funkcionalnost mora se implementirati direktno u hardveru, projektovanjem namenskih digitalnih elektronskih sistema.

Problematikom particionisanja na sistemskom nivou bavi se oblast kodizajna hardvera i softvera (*Hardware/Software Codesign*) i u ovom kursu o njoj dalje neće biti reči.

Particionisanje na nivou IP modula

Na ovom hijerarhijskom nivou hardverska komponenta celokupnog sistema deli se na blokove, takozvana IP jezgra ili IP module, koji enkapsuliraju neku jasno definisanu

funkcionalnost. Po pravilu, IP moduli realizuju neke standardne, često korišćene funkcije kao što su:

- **IP moduli za skladištenje podataka** – u ovu grupu spadaju moduli koji implementiraju različite mehanizme skladištenja podataka. U ovu grupu spadaju: SRAM, DRAM, ROM, Flash, FIFO, razne vrste bafera, itd.
- **IP moduli za prenos podataka** – u ovu grupu spadaju moduli koji implementiraju različite komunikacione protokole, kao što su: AXI, PLB, PCIe, SATA, Ethernet, USB, I2C, SPI, I2S, FireWire, itd.
- **IP moduli za obradu podataka** – u ovu grupu spadaju moduli pomoću kojih se vrši obrada podataka. U ovu grupu spadaju moduli za:
    - filtriranje (FIR i IIR filtri), izvođenje unitarnih transformacija (FFT, DCT, DWT, ...),
    - kompresiju podataka (JPEG, MPEG, H264, MP3, ...),
    - enkripciju podataka (DES, AES, RSA, ...),
    - zaštitno kodovanje podataka (CRC, Huffman, BCH, Reed-Solomon, LDPC, ...),
    - bilo koji drugi moduli koji implementiraju specifične algoritme obrade podataka.
- **Procesorski IP moduli** – ovi moduli takođe vrše obradu podataka i zapravo pripadaju prethodnoj grupi, ali se obično posmatraju odvojeno. Ovo je stoga što je, za razliku od modula iz prethodne grupe, pomoću procesorskih IP modula moguće implementirati proizvoljni algoritam obrade podataka. U ovu grupu spadaju različite vrste procesora, kao što su:
    - skalarni procesori,
    - superskalarni procesori,
    - VLIW procesori,
    - vektorski procesori,
    - višejezgarni procesori,
    - konfigurabilni procesori.

Ovi moduli uvek imaju jasno definisan interfejs, najčešće je to neki od standardnih interfejsa kao što su: AXI-Full, AXI-Lite, AXI-Stream, PLB, OPB, LocalLink. Prednost korišćenja standarnih interfejsa ogleda se u jednostavnosti integracije IP modula u različite sisteme, što povećava šansu za njihovim ponovnim korišćenjem.

Particionisanje unutar IP modula

Particionisanje unutar IP modula obično uključuje podelu na dva velika podsistema:

- Podsistem za obradu podataka, *datapath* – ovaj podsistem zadužen je za izvođenje svih elementarnih transformacija nad podacima koje posmatrani IP modul treba da obezbedi. On se dalje dekomponuje na tri podsistema:
    - podsistem za skladištenje podataka – realizovan u vidu pojedinačnih registara, registarskih banaka, jednoprostupnih ili višepristupnih memorija,
    - podsistem za realizaciju potrebnih aritmetičkih, logičkih i relacionih transformacija nad podacima – realizovan korišćenjem sabirača, oduzimača, množača, delitelja, pomerača, komparatora, itd. ,
    - podsistem za rutiranje podataka – realizovan pomoću multipleksera, demultipleksera ili kola sa stanjem visoke impedanse,
- Upravljački podsistem, *controlpath* – ovaj podsistem zadužen je za sprovođenje potrebnih koraka u procesu obrade podataka koje posmatrani IP modul treba da obezbedi. Proces obrade podataka uvek se može dekomponovati na sekvencu elementarnih aritmetičkih, logičkih i relacionih operacija, čijim izvršavanjem se ostvaruje željena funkcionalnost. Zadatak *contolpath* modula je da obezbedi pravilan redosled izvršavanja ovih elementanih operacija. Ovaj podsistem uvek se realizuje u vidu konačnog automata.

Particionisanje na nivou RT komponenti

Datapath i *controlpath* moduli obično se dalje particionišu na module koji predstavljaju standardne kombinacione i sekvencijalne mreže: multipleksere, demultipleksere, kodere, dekodere, komparatore, aritmetičke i logičke module, razne vrste registara, registarske banke, memorije malog kapaciteta, FIFO i ostale vrste bafera, brojače, konačne automate, itd.

Dalje particionisanje, ispod nivoa standardnih kombinacionih i sekvencijalnih mreža, prilikom projektovanja savremenih digitalnih sistema se ne sprovodi, jer postojeći alati mogu vrlo efikasno potpuno automatski izvršiti particiju standarnih kombinacionih i sekvencijalnih mreža na module koji zapravo predstavljaju individualne logičke kapije i flip flopove, a zatim i individualne tranzistore.

Stoga se, prilikom projektovanja savremenog digitalnog sistema, proces dalje hijerarhijske dekompozicije zaustavlja u trenutku kada se naiđe na modul koji predstavlja neku od standardnih kombinacionih ili sekvencijalnih mreža.

Hijerarhijska dekompozicija procesorskog sistema sa slike 2.1 izvedena je upravo poštujući gore navedena pravila, počevši od nivoa particionisanja unutar IP modula, zaključno sa nivoom RT komponenti.

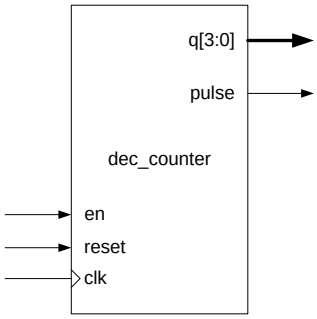
2.2. Ponovno korišćenje modula

Drugi način povećanja produktivnosti korišćenjem modularizacije ogleda se u mogućnosti ponovnog korišćenja ranije razvijenih modula (*Design Reuse*) u novom dizajnu. Umesto da trošimo vreme na razvoj nekog modula, možda možemo jednostavno iskoristiti neki od postojećih modula uz nikakve ili minimalne modifikacije. Na ovaj način se opet štedi vreme i smanjuju troškovi razvoja novog sistema. Na primer, prilikom projektovanja našeg hipotetičkog mikroprocesora, možda bi bilo moguće iskoristiti ranije razvijene module množača i delitelja. Slično, modeli multipleksera i dekodera bi takođe mogli lako da se iskoriste, koristeći ranije razvijene modele. Ukoliko nemamo odgovarajuće module, možda ih je moguće kupiti od kompanija koje se bave razvojem i prodajom standardnih modula (*IP cores*). Ako je reč o složenom modulu, možda je isplativije kupiti ga od neke kompanije, nego se upuštati u sopstveni razvoj.

Zadaci za vežbu

Zadatak 2.1.

Napisati VHDL model trocifrenog decimalnog brojača koji broji od 000 do 999, a zatim kreće od početka. Prilikom projektovanja sistema, primeniti tehniku hijerarhijskog dizajna, koristeći komponentu *dec_counter*. Komponenta *dec_counter* realizuje jednocifreni decimalni brojač koji broji od 0 do 9, a zatim kreće od početka. Interfejs komponente *dec_counter* prikazan je na slici 2.3.



Slika 2.3. Interfejs komponente dec_counter

Ulazi *en* i *reset* su sinhroni kontrolni ulazi. Kada je reset aktivan, brojač se vraća u početno stanje i na izlazu *q* se nalazi vrednost „0000“, a izlaz *pulse* ima vrednost ’0’. Brojač broji na gore samo ako je kontrolni ulaz *en* aktivan. Trenutna vrednost brojača prosleđuje se ostatku sistema preko *q* izlaza. Funkcija izlaza *pulse* jeste da signalizira da

je brojač stigao do maksimalne vrednosti, „1001“. U tom trenutku *pulse* ima vrednost ’1’, a u svim ostalim ima vrednost ’0’.

a) Nacrtati blok dijagram trocifrenog decimalnog brojača i označiti sve signale na njemu.

b) Na osnovu blok dijagrama napisati VHDL model.

c) Model pod b) realizovati korišćenjem *component* naredbe.

d) Model pod b) realizovati korišćenjem konfiguracije.

2.3. Parametrizovani dizajn

Da bi se povećala mogućnost ponovnog korišćenja nekog IP modula, on bi trebao da ima mogućnost *parametrizacije*. Pod parametrizovanim modelom nekog modula podrazumevamo model koji se jednostavno može prilagoditi tekućim potrebama, prostim konfigurisanjem skupa parametara koji su pridruženi modelu. Na primer, ukoliko napišemo model multipleksera kod koga se može jednostavno promeniti širina ulaznih i izlaznih portova za podatke, onda ga kasnije možemo koristiti i kao 8-bitni, 16-bitni ili 32-bitni multiplekser. Slična logika može se primeniti i prilikom razvoja većine ostalih standardnih kombinacionih i sekvencijalnih mreža (demultiplekseri, komparatori, registri, brojači, itd.).

U zavisnosti od toga šta se parametrizuje, svi parametrizovani modeli mogu se podeliti u dve velike grupe:

- modele sa parametrizovanom širinom,
- modele sa parametrizovanim ponašanjem.

Kod parametrizacije širine, korišćenjem odgovarajućeg broja parametara moguće je definisati potrebnu širinu ulaznih i izlaznih portova modela. U praksi je vrlo čest slučaj da je jedina modifikacija koju je neophodno izvršiti, da bi se ranije razvijeni model mogao koristiti u nekom drugom dizajnu, promena širine ulaznih ili izlaznih portova modula. Parametrizacija širine omogućena je unutar VHDL jezika postojanjem *generic* deklarativnog dela unutar deklaracije entiteta. Definisanjem odgovarajućeg broja generičkih konstanti, koji se kasnije mogu koristiti kao parametri unutar arhitekturnog tela, pa čak i u nastavku *entity* deklaracije, moguće je izvršiti parametrizaciju širine bilo kog VHDL modela. Ponekad se prilikom uvođenja parametrizacije širine ulaznih ili izlaznih portova javlja potrebna i za strukturnim modifikacijama unutar samog modela. U tom slučaju je neophodno koristiti VHDL *generate* naredbe i razviti regularne iterativne modele.

Drugu grupu parametrizovanih modela čine modeli kod kojih je parametrizovano ponašanje. Jedan tip ponašanja koji se često parametrizuje jeste parametrizacija

različitih vremenskih karakteristika modela koji se razvija (propagacionog kašnjenja, vremena uspostavljanja i držanja, vremena stabilizacije novih vrednosti na izlazima nakon nailaska rastuće, ili opadajuće, ivice sinhronizacionog signala, itd.). Ovakva vrsta parametrizovanog ponašanja od velikog je interesa prilikom razvoja simulacionih modela (VITAL IEEE standard razvijen je upravo u ovu svrhu). U ovu svrhu ponovo se koriste *generic* deklaracije unutar kojih se definišu odgovarajući parametri, koji se zatim koriste unutar samog modela za parametrizaciju vremenskih karakteristika.

Međutim, moguće je i parametrizovati funkcionalnost samog modela, uključujući ili isključujući odgovarajuće funkcije koje model obezbeđuje. Na ovaj način se model konfiguriše i obezbeđuje samo onaj deo funkcionalnosti koji je zaista neophodan u tekućem sistemu u kojem će se model koristiti. Ovaj vid parametrizacije ponašanja uključuje korišćenje odgovarajućih *if generate* naredbi unutar VHDL modela, pomoću kojih se uključuju ili isključuju odgovarajuće komponente prisutne unutar modela. Većina IP jezgara odlikuje se ovakvim tipom parametrizacije ponašanja.

Zadaci za vežbu

Zadatak 2.2.

Napisati VHDL model brojača koji broji od *m* do *n*, a zatim ponovo počinje brojanje od *m*. Koristiti dve generičke konstante, *M* i *N*, pomoću kojih je moguće zadati vrednosti za početnu i krajnju vrednost brojanja.

Zadatak 2.3.

Napisati VHDL model konvertora serijskog koda u paralelni. Konvertor prihvata podatke serijski i smešta ih u unutrašnji pomerački registar. Sadržaj ovog registra predstavlja konvertovanu vrednost koja se prosleđuje na izlaz konvertora. Napisani model treba da bude univerzalan, u smislu da je moguće menjati broj bita u pomeračkom registru, a samim tim i veličinu paralelnog izlaza konvertora. *Entity* deklaracija modela koji treba napisati prikazana je na slici 2.4.

```
entity s2p_converter is
      generic (WIDTH: natural);
      port (
         clk: in std_logic;
         si:  in std_logic;
         q:   out std_logic_vector(WIDTH-1 downto 0)
      );
end s2p_converter;
```

Slika 2.4. Entity deklaracija konvertora serijskog koda u paralelni

a) Napisati parametrizovani model konvertora koristeći *entity* deklaraciju sa slike 2.4.

b) Napisati jednostavan testbenč pomoću koga je moguće verifikovati ispravan rad napisanog modela.

Zadatak 2.4.

Definicija VHDL operatora sumiranja je vrlo jednostavna. On uzima dva operanda i vraća jedan rezultat koji predstavlja njihovu sumu. Međutim, prilikom projektovanja složenih digitalni sistema, kao što su procesori, sabirači tipično moraju da generišu i čitav niz dodatnih, statusnih signala. Najčešći statusni signali koji se koriste su indikatori nule (*zero*), znaka (*sign*) i prekoračenja opsega (*overflow*).

Pravilna za generisanje statusnih signala nisu jednostavna zbog mogućeg prekoračenja opsega. Potencijalno prekoračenje opsega utiče i na vrednost znaka rezultata i na vrednost indikatora nule. Upravo zbog toga je prvo neophodno izračunati vrednost indikatora prekoračenja opsega. *Overflow* signal treba da se generiše na osnovu sledećih pravila:

- Ako operandi imaju različit znak, do prekoračenja nikada ne može doći, jer se sabiranjem pozitivnog i negativnog broja uvek smanjuje apsolutna vrednost rezultata.
- Ako operandi i rezultat imaju isti znak, do prekoračenja nije došlo.
- Ako operandi imaju isti znak, ali rezultat je drugog znaka, došlo je do prekoračenja. Promena znaka je zapravo indikacija da je rezultat premašio maksimalnu pozitivnu ili negativnu vrednost i da se samim tim nalazi izvan mogućeg opsega.

Kada je poznata vrednost *overflow* signala moguće je odrediti vrednosti *sign* i *zero* signala.

S'obzirom da je moguće da dođe do prekoračenja opsega, rezultat sabiranja dva broja može biti različit od nule iako je *sum* izlaz jednak nuli. Na primer, ukoliko sabiramo dva 4-bitna broja „1000" i „1000", *sum* izlaz imaće vrednost „0000" zbog prekoračenja opsega, iako stvarni rezultat sumiranja nije jednak nuli. Zbog toga, statusni signal *zero* treba da je aktivan samo ako je *sum* izlaz jednak nuli i nije došlo do prekoračenja.

Na osnovu prethodno razmatranja načina generisanja *overflow* signala, jasno je da znak izračunate sume dva broja može biti različit od stvarnog znaka sume dva broja. Ukoliko posmatramo prethodni primer sabiranja dva 4-bitna broja, znak izračunate sume „0000" je '0', dok je stvarni rezultat zapravo negativan broj. Stoga je *sign* statusni signal jednak znaku izračunate sume samo ukoliko nije došlo do prekoračenja opsega. U protivnom treba da ima invertovanu vrednost.

U praksi je takođe vrlo čest slučaj da je neophodno koristiti sabirače sa različitom veličinom ulaza. Umesto da se za svaki konkretni slučaj piše novi VHDL model, moguće je napisati samo jedan, parametrizovani model. *Entity* deklaracija jednog

takvog parametrizovanog modela sabirača dva binarna broja proizvoljne širine sa *sign*, *zero* i *overflow* statusnim signalima prikazana je na slici 2.5.

```
entity para_adder_status is
    generic (WIDTH: natural);
    port (
        a, b:  in std_logic_vector(WIDTH-1 downto 0);
        cin:   in std_logic;
        sum: out std_logic_vector(WIDTH-1 downto 0);
        cout, zero, overflow, sign: out std_logic
    );
end para_adder_status;
```

Slika 2.5. Entity deklaracija parametrizovanog sabirača sa statusnim signalima

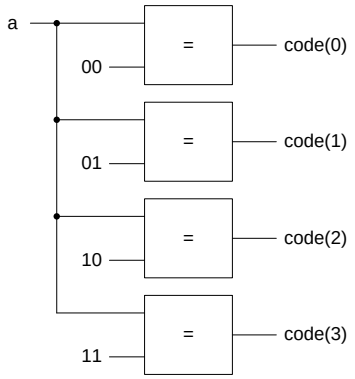
a) Napisati parametrizovani model sabirača koristeći *entity* deklaraciju sa slike 2.5.

b) Napisati jednostavan testbenč pomoću koga je moguće verifikovati ispravan rad napisanog modela.

Zadatak 2.5.

Binarni n na 2^n dekoder jeste kombinaciona mreža koja u zavisnosti od vrednosti n-bitnog ulaza, aktivira tačno jedan od 2^n izlaznih signala. Iako je lako napisati VHDL model dekodera za fiksnu veličinu ulaza, razvoj parametrizovanog modela je složeniji.

Jedan od mogućih načina da se posmatra rad dekodera je da se svaki izlazni bit tretira kao izlaz konstantnog komparatora. Dekodovani bit je aktivan samo ako je vrednost ulaznog signala jednaka nekoj, unapred definisanoj konstantnoj vrednosti. Na primer, blok dijagram 2 na 2^2 dekodera baziran na gore opisanoj ideji prikazan je na slici 2.6.



Slika 2.6. Blok dijagram 2 na 4 dekodera

Obzirom da je jedan od ulaza u komparator konstantan, sam komparator će biti uprošćen prilikom sinteze. Prikazani blok dijagram se lako može modifikovati u slučaju da je ulaz različite širine. Pravilo generisanja signala *code(i)*, u *i*-toj fazi dekodera, može se opisati pomoću sledeće VHDL naredbe:

```
code(i) <= '1' when a=std_logic_vector(unsigned(i)) else '0';
```

Koristeći **for-generate** naredbu moguće je napisati parametrizovani model binarnog dekodera sa proizvoljnom širinom ulaza. *Entity* deklaracija parametrizovanog dekodera mogla bi imati izgled kao na slici 2.7.

```
entity para_bin_decoder is
    generic (WIDTH: natural);
    port (
        a:    in std_logic_vector(WIDTH-1 downto 0);
        code: out std_logic_vector(2**WIDTH-1 downto 0)
    );
end para_bin_decoder;
```

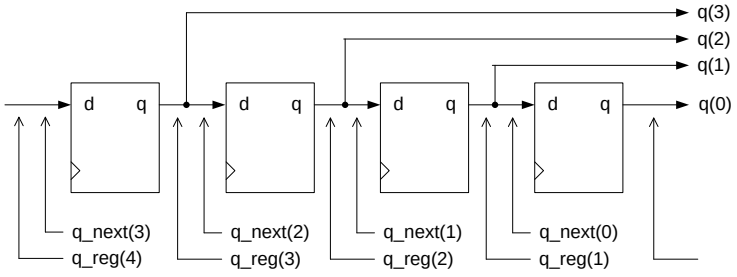
Slika 2.7. Entity deklaracija parametrizovanog binarnog dekodera

a) Napisati parametrizovani model binarnog dekodera koristeći **for-generate** naredbu i *entity* deklaraciju sa Slike 2.7.

b) Napisati jednostavan testbenč pomoću koga je moguće verifikovati ispravan rad napisanog modela.

Zadatak 2.6.

U zadatku 2.3 u okviru parametrizovanog dizajna, razmatran je konvertor serijskog koda u paralelni. Konvertor se sastoji iz niza kaskadno povezanih D flip-flopova, kao što je prikazano na blok dijagramu sa slike 2.8. u slučaju 4-bit konvertora.



Slika 2.8. Blok dijagram 4-bitnog konvertora serijskog koda u paralelni

Svaka od faza konverzije se sastoji iz jednog D flip-flopa i logike za generisanje narednog stanja, koja je u ovom slučaju obična žica koja spaja izlaz tekućeg flip-flopa sa ulazom narednog.

a) Na osnovu blok dijagrama sa slike 2.8. napisati parametrizovani model konvertora koristeći **for-generate** naredbu i *entity* deklaraciju sa slike 2.4. D flip-flop opisati pomoću **process** naredbe.

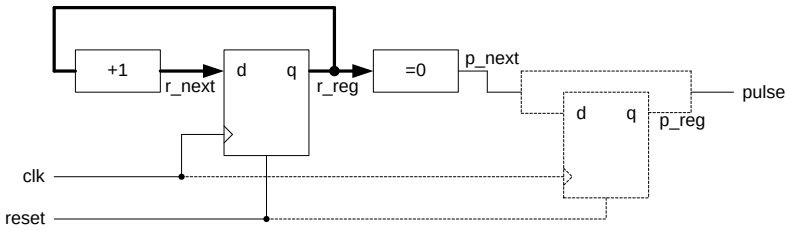
b) Ponoviti isto kao pod a), ali ovaj put D flip-flop treba koristiti kao komponentu, čiji je model napisan u zasebnom VHDL fajlu.

c) Napisati jednostavan testbenč pomoću koga je moguće verifikovati ispravan rad napisanih modela.

Zadatak 2.7.

Prilikom projektovanja modula koji će biti korišćeni u nekom složenijem dizajnu, vrlo često se javlja potreba za korišćenjem izlaznih bafera (tipično su to flip-flopovi ili registri) na nekim ili svim izlaznim signalima modula koji se projektuje. Razlozi za postavljanje izlaznih bafera mogu biti različiti, počev od raskidanja dugačkih propagacionih lanaca koji se inače mogu pojaviti i otežati zadovoljavanje potrebnih vremenskih ograničenja (potrebne učestanosti na kojoj sistem treba da pouzdano radi), do uklanjanja gličeva koji se inače mogu pojaviti na izlaznim signalima. S'obzirom da su baferi potrebni samo u nekim aplikacijama, bilo bi zgodno da se oni u VHDL modelu opišu kao opcioni deo sistema, koji se po potrebi može uključiti ili isključiti. Ovo je moguće ukoliko se definiše odgovarajući parametar i koristi uslovna *generate* naredba.

Na primer, neka je potrebno projektovati binarni brojač koji broji na gore, proizvoljne širine, određene parametrom WIDTH, i koji generiše izlazni signal *pulse* kada dostigne vrednost 0. Pomoću parametra BUFF, moguće je kontrolisati da li je neophodno ubaciti bafer na izlazni signal *pulse* ili ne. Blok dijagram sistema prikazan je na slici 2.9.



Slika 2.9. Blok dijagram brojača sa opcionim izlaznim baferom

a) Na osnovu blok dijagrama sa slike 2.9 napisati parametrizovani model sistema koristeći **if-generate** naredbu.

b) Napisati jednostavan testbenč pomoću koga je moguće verifikovati ispravan rad napisanog modela.

3.

Projektovanje *datapath* i *controlpath* modula

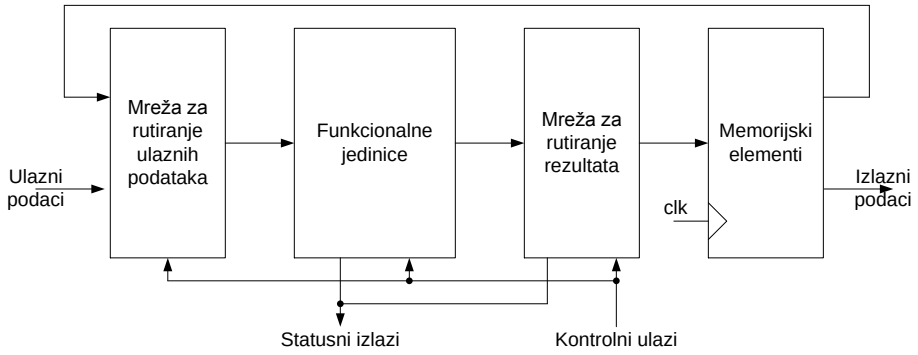
U prethodnoj glavi rečeno je da je uobičajeno da se svaki IP modul na početku hijerarhijske dekompozicije particioniše na dva velika podsistema:

- Podsistem za obradu podataka, *datapath* – zadužen za izvođenje svih elementarnih transformacija nad podacima koje posmatrani IP modul treba da obezbedi.
- Podsistem za upravljanje, *controlpath* – zadužen za kontrolisanje *datapath* modula.

Svi složeni digitalni sistemi danas se projektuju primenom RT metodologije (koja je predmet sledeće glave), a koja uvek uključuje projektovanje odgovarajućih *datapath* i *controlpath* modula. Slično, projektovanje složenih digitalnih sistema primenom sinteze visokog nivoa, koja predstavlja budućnost projektovanja složenih digitalnih sistema, takođe pretpostavlja model složenog sistema koji se na najvišem nivou hijerarhije sastoji iz *datapath* i *controlpath* modula. Zbog ovih činjenica u ovoj glavi biće prikazani standardni postupci za modelovanje i implementaciju *datapath* i *controlpath* modula.

3.1. Projektovanje *datapath* modula

Opšta struktura *datapath* modula prikazana je na slici 3.1.



Slika 3.1. Opšta struktura datapath modula

Sa slike 3.1 može se primetiti da se u opštem slučaju svaki *datapath* modul sastoji iz sledećih blokova:

- **Mreže za rutiranje ulaznih podataka funkcionalnih jedinica** – ovaj blok služi za dovođenje potrebnih podataka do funkcionalnih jedinica, kako bi se u narednom taktu mogle izvesti neophodne funkcionalne trasformacije nad njima. Rutiranje podataka do funkcionalnih jedinica može biti impementirano pomoću multipleksera (što je najčešći slučaj) ili pomoću jedne ili više magistrala. Sami podaci sastoje se iz dve grupe: ulaznih podatka u čitav *datapath* modul i unutrašnjih podataka koji predstavljaju tekuće stanje memorijskih elemenata.
- **Mreže za rutiranje rezultata** – ovaj blok služi za dovođenje rezultata obrade pomoću funkcionalnih jedinica do odabranih memorijskih elemenata, kako bi se sačuvali međurezultati, kao i konačni rezultati celokupnog procesa obrade koji je implementiran unutar IP modula. Kao i u slučaju mreže za rutiranje ulaznih podataka, rutiranje rezultata obrade pomoću funkcionalnih jedinica do memorijskih elemenata može biti impementirano pomoću multipleksera (što je najčešći slučaj) ili pomoću jedne ili više magistrala.
- **Funkcionalnih jedinica** – unutar ovog bloka implementirane su sve funkcionalne jedinice pomoću kojih je moguće realizovati sve elementarne operacije, neophodne za implementaciju celokupnog procesa obrade od strane IP modula. Ove funkcionalne jedinice mogu biti: jedinice za izvođenje aritmetičkih operacija (sabiranja, oduzimanja, množenja, delenja, itd.), jedinice za izvođenje logičkih operacija, jedinice za izvođenje relacionih operacija, itd.

- **Memorijskih elemenata** – ovaj blok služi za čuvanje trenutnih vrednosti međurezultata, kao i konačnih rezultata obrade. Kako proces obrade, koji je implementiran unutar IP modula, po pravilu zahteva veći broj taktova za kompletiranje, unutar *datapath* modula mora postojati mogućnost da se nakon svakog takta mogu sačuvati međurezultati, kako bi se mogli koristiti u narednim fazama obrade. Memorijski elementi mogu biti realizovani kao individualni flip-flopovi, registri, registarske banke ili višepristupne memorije.

Što se tiče interfejsa *datapath* modula, on obično uključuje sledeće portove:

- **Interfejs za ulazne podatke** – ovaj interfejs omogućava da se do *datapath* modula dovedu podaci koje je potrebno obraditi. Obično se realizuje kao skup individualnih višebitnih magistrala, ali se može realizovati i kao interfejs ka spoljašnoj memoriji ili FIFO baferu, ili kao interfejs koji je kompatibilan sa nekim od standarnih komunikacionih protokola (kao što su AXI, PLB, OPB, itd.), u zavisnosti od potrebe.
- **Interfejs za izlazne podatake** – ovaj interfejs služi da se rezultati obrade podataka unutar *datapath* modula prenesu ka nekom od susednih modula. Obično se realizuje kao skup individualnih višebitnih magistrala, ali se može realizovati i kao interfejs ka spoljašnoj memoriji ili FIFO baferu, ili kao interfejs koji je kompatibilan sa nekim od standarnih komunikacionih protokola (kao što su AXI, PLB, OPB, itd.) u zavisnosti od potrebe.
- **Izlazne statusne signale** – ovi izlazni portovi služe za prenos statusnih signala, o tekućem stanju *datapath* modula. Ovi signali se uglavnom vode na odgovarajuće ulaze *controlpath* modula, koji na osnovu njihovih vrednosti generiše odgovarajuće upravljačke akcije.
- **Kontrolne ulaze** – ovi ulazni portovi zapravo služe za kontrolu rada *datapath* modula. Pomoću njih moguće je kontrolisati tok podataka unutar *datapath* modula (kontrolišući mreže za rutiranje), kao i aktivnost funkcionalnih jedinica. Na ove ulazne portove se obično dovode izlazni upravljački signali iz *controlpath* modula

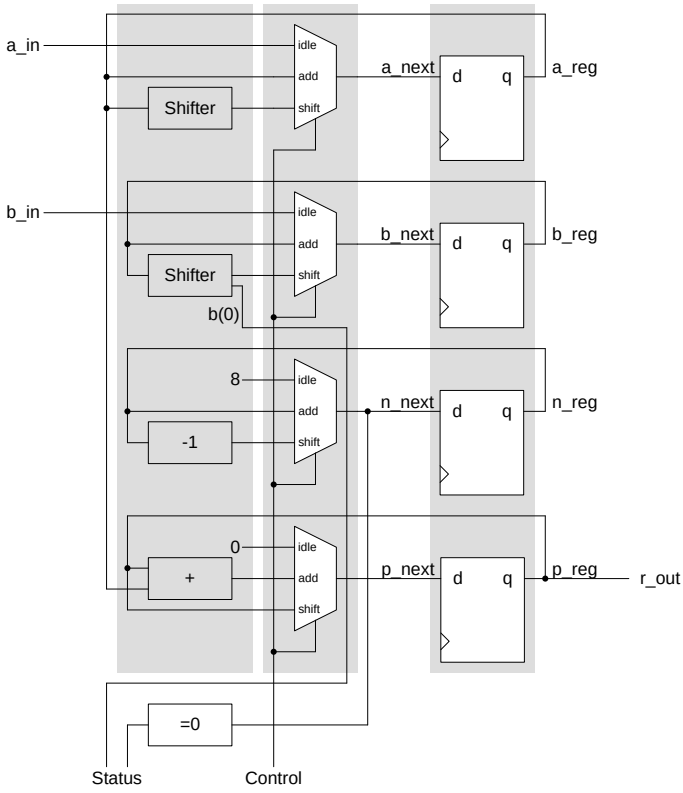
Zadaci za vežbu

Zadatak 3.1.

Na slici 3.2 prikazan je blok dijagram jednog hipotetičkog *datapath* modula. *Datapath* modul sa slike 3.2 ima sledeće portove:

- Ulazne portove **a_in** i **b_in** – preko ovih ulaznih portova se dovode podaci koje je potrebno obraditi unutar *datapath* modula. Širina ova dva ulazna porta iznosi 8 bita.

- Izlazni port **r_out** – preko ovog izlaznog porta okruženju se prosleđuje rezultat obrade podataka pomoću *datapath* modula. Širina ovog izlaznog porta iznosi 16 bita.
- Izlazni status port, **status** – reč je o 2-bitnom izlaznom portu. Nulti bit ovog porta zapravo je jednak vrednosti unutrašnjeg signala *b(0)*, dok je prvi bit povezan na izlaz komparatora sa nulom.
- Ulazni kontrolni port, **control** – 2-bitni ulazni port pomoću kojega se kontrolišu multiplekseri za rutiranje ulaznih podataka unutar *datapath* modula.



Slika 3.2. Blok dijagram hipotetičkog datapath modula

a) Napisati VHDL model *datapath* modula sa slike 3.2, pri čemu svi blokovi sa slike 3.2 treba da budu modelovani unutar istog entiteta, korišćenjem odgovarajućeg broja *process* naredbi. Svaki blok treba da ima svoju odvojenu *process* naredbu ili odgovarajuću naredbu konkurentne dodele vrednosti signalu.

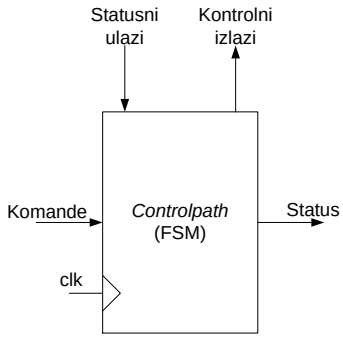
b) Napisati VHDL model *datapath* modula sa slike 3.2, u okviru kojega će čitav *datapath* modul biti modelovan korišćenjem četiri *process* naredbe. Prva process naredba treba da modeluje funkcionalnost svih memorijskih elemenata, druga funkcionalnost mreže za rutiranje rezultata, treća funkcionalne jedinice i četvrta jedinice za generisanje statusnih signala.

c) Napisati VHDL model *datapath* modula sa slike 3.2, koristeći strukturni stil modelovanja. Svaki od blokova sa slike 3.2 treba modelovati kao poseban entitet, a u top modelu čitavog *datapath* modula treba instancionirati odgovarajući broj puta svaki od napisanih entiteta i povezati ih na način prikazan na slici 3.2.

d) Napisati jednostavan testbenč pomoću kojega će biti moguće verifikovati korektnost razvijenih modela.

3.2. Projektovanje *controlpath* modula

Kao što je već rečeno, osnovna uloga *controlpath* modula je da upravlja radom asociranog *datapath* modula, kako bi se realizovao željeni postupak obrade podataka koji je implementiran u posmatranom IP modulu. Tipični interfejs *controlpath* modula prikazan je na slici 3.3.



Slika 3.3. Interfejs controlpath modula

Kao što se sa slike 3.3 može videti, interfejs *controlpath* modula sastoji se iz sledećih portova:

- **Ulaznog komandnog intefejsa** – preko ovog interfejsa okruženje može upravljati radom *controlpath* modula, zadajući mu odgovarajuće komande, a samim tim i radom čitavog IP modula. Ovaj komandni interfejs može biti vrlo jednostavan (sastavljen od jednog ili više jednobitnih ili višebitnih ulaznih

portova), ali se može realizovati tako da bude kompatibilan sa nekim od standardnih komunikacionih protokola (kao što su AXI, PLB, OPB, itd.).

- **Izlaznog statusnog interfejsa** – preko ovog interfejsa *controlpath* modul može saopštiti okruženju kakav je njegov trenutni status (da li je spreman za izvršavanje nove komande, da li je zauzet izvršavanjem neke komande, da li je tekuća komanda završena, da li se mogu se preuzeti podaci koji predstavljaju rezultat izvršavanja poslednje zadate komande, itd.). Ovaj interfejs je obično vrlo jednostavan i često se sastoji od jednog višebitnog izlaznog porta.
- **Ulaznog statusnog interfejsa** – namena ovog ulaznog interfejsa je da se preko njega *controlpath* modulu saopšti trenutni status asociranog *datapath* modula. Na osnovu ovih informacija *controlpath* modul zatim može preduzeti odgovarajuće upravljačke akcije. Ovaj interfejs je takođe vrlo jednostavan i sastoji je iz jednog ili većeg broja višebitnih ulaznih portova.
- **Izlaznog kontrolnog porta** – pomoću ovog izlaznog porta *controlpath* modul generiše upravljačke signale pomoću kojih upravlja radnom asociranog *datapath* modula. Ovaj interfejs je takođe vrlo jednostavan i sastoji je iz jednog ili većeg broja višebitnih izlaznih portova.

S'obzirom da *controlpath* modul treba da realizuje upravljački algoritam koji se uglavnom sastoji iz determinističke sekvence upravljačkih akcija, čijim izvršavanjem se ostvaruje postupak obrade koji je potrebno implementirati unutar posmatranog IP modula, prirodno je da se *controlpath* modul realizuje kao **konačni automat** (*Finite State Machine, FSM*).

Kao što je od ranije već poznato, konačni automat se formalno definiše pomoću sledećih pet objekata:

- konačnog skupa simboličkih stanja u kojima se može naći automat,
- konačnog skupa ulaznih signala na koje je automat osetljiv,
- konačnog skupa izlaznih signala koje generiše automat,
- funkcije narednog stanja, koja se koristi za određivanje narednog stanja u kome će se naći automat,
- izlazne funkcije, koja se koristi za određivanje narednih vrednosti izlaznih signala koje generiše automat.

Stanje automata predstavlja jedinstveno unutrašnje stanje u kome se automat može naći. Skup svih dozvoljenih stanja konačnog automata je konačan (otuda i naziv *konačni* automat). Kako protiče vreme, konačni automat prelazi iz jednog stanja u drugo. Naredno stanje automata određeno je funkcijom narednog stranja, koja na osnovu tekućeg stanja u kome se nalazi automat i tekućih vrednosti ulaznih signala određuje naredno stanje u kome će se naći automat. U slučaju *sinhronih* konačnih automata, ovi prelazi su kontrolisani pomoću klok signala i mogu se odigrati jedino kada se na klok

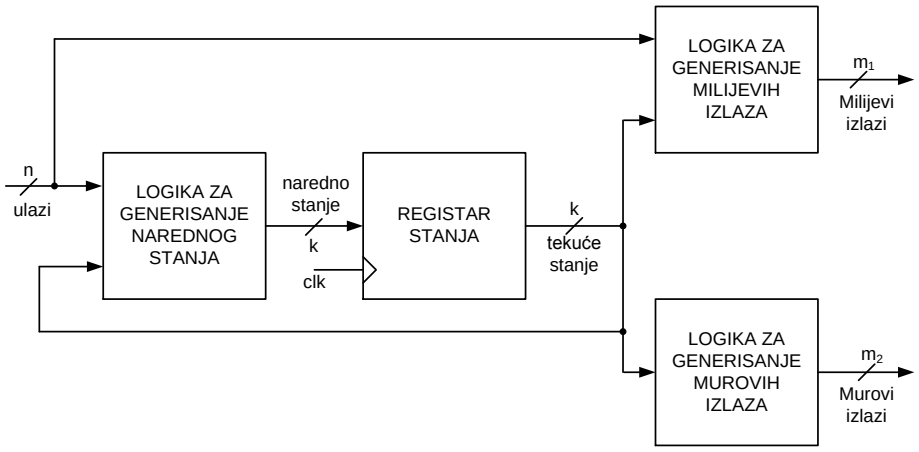
signalu pojavi rastuća (ili eventualno opadajuća) ivica. Iako postoje i *asinhroni* konačni automati, u okviru ovog kursa bavićemo se isključivo sinhronim konačnim automatima.

Izlazna funkcija automata koristi se za određivanje tekućih vrednosti izlaznih signala automata. U zavisnosti od toga šta čini ulaze u ovu funkciju, na osnovu kojih se zatim određuju tekuće vrednosti izlaznih signala automata, izlazni signali konačnih automati mogu se podeliti u dve velike grupe:

- **Murove (*Moore*) izlaze** – koji se generišu samo na osnovu tekućeg stanja u kojem se automat nalazi,
- **Milijeve (*Mealy*) izlaze** – koji se generišu na osnovu tekućeg stanja u kojem se automat nalazi, ali i na osnovu tekućih vrednosti ulaznih signala automata.

Automati koji sadrže samo izlaze Murovog tipa nazivaju se Murovi automati, dok se automati koji sadrže samo izlaze Milijevog tipa nazivaju Milijevi automati. Iako postoje automati koji su striktno Murovog ili Milijevog tipa, u praksi se najčešće sreću automati kod kojih imamo i jednu i drugu vrstu izlaza.

Blok dijagram generalnog konačnog automata prikazan je na slici 3.4.



Slika 3.4. Generalna struktura konačnog automata

Konačni automat u opštem slučaju poseduje sledeće portove:

- N ulaznih portova,
- M_1 izlaznih portova Milijevog tipa,
- M_2 izlaznih portova Murovog tipa,

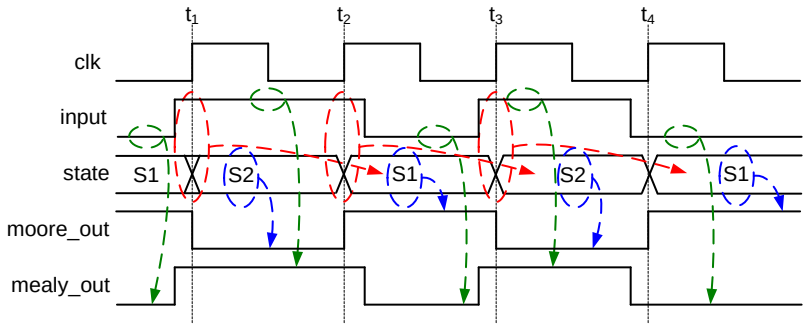
- *clk* ulaz za sinhronizaciju rada konačnog automata

Opciono, konačni automat može posedovati i sledeće portove:

- *reset* ulazni port za inicijalizaciju sadržaja registra stanja
- *clock enable* ulazni port za selekciju rastućih ivica *clk* porta na koje konačni automat treba da se aktivira

N ulaznih portova konačnog automata može biti organizovano u *N* jednobitnih ulaznih portova, ali oni takođe mogu biti i višebitni ulazni portovi. Princip rada konačnog automata je sledeći. Nailaskom rastuće ivice *clk* signala konačni automat, u zavisnosti od stanja u kome se trenutno nalazi, analizira trenutno stanje na ulaznim portovima i, koristeći funkciju narednog stanja, određuje naredno stanje u kome će se naći tokom sledeće periode *clk* signala. Istovremeno, konačni automat generiše nove vrednosti za odgovarajuće Milijeve i Murove izlaze, na osnovu trenutnog stanja u kome se nalazi i trenutnih vrednosti ulaznih portova. Slično kao kod ulaznih signala, M_1 Milijevih i M_2 Murovih izlaznih portova može biti jednobitno ili višebitno.

Na primer, vremenski dijagram rada konačnog automata sa jednim ulazom, *input*, i po jednim Murovim i Milijevim izlazom, *moore_out* i *mealy_out*, prikazan je na slici 3.5. U ovom primeru ulazni port *input* je jednobitni, kao i Murovi i Milijevi izlazni portovi *moore_out* i *mealy_out*.



Slika 3.5. Vremenski dijagram rada konačnog automata, sa jednim ulazom i dva izlaza

Sa slike 3.5 možemo videti da se konačni automat pre nailaska prve rastuće ivice *clk* signala (u trenutku t_1) nalazi u stanju *S1*. Murov izlaz (*moore_out* signal) zavisi samo od trenutnog stanja u kome se automat nalazi i možemo videti da je na visokom logičkom nivou. To znači da će automat ovaj izlaz uvek postavljati na visoki logički nivo kada se nalazi u stanju *S1*. Milijev izlaz zavisi ne samo od stanja u kome se automat trenutno nalazi već i od trenutne vrednosti ulaza, u našem slučaju ulaza *input*. Sa slike 3.5 možemo videti da Milijev izlaz našeg automata (*mealy_out* signal) prati talasni oblik

ulaznog signala *input* i ne zavisi od stanja u kome se automat nalazi. Nailaskom rastuće ivice *clk* signala u trenutku t_1 automat prelazi u novo stanje, *S2*, u kome će ostati sve do nailaska sledeće rastuće ivice, u trenutku t_2. Kada automat pređe u stanje *S2* dolazi do promene na Murovom izlazu (signal *moore_out* postaje 0), što znači da konačni automat sa slike 3.5 svaki put kada se nađe u stanju *S2* spušta *moore_out* izlaz na 0. Karakteristične promene do kojih dolazi prilikom rada automata na slici 3.5 označene su različitim bojama. Crvenom bojom označeni su trenuci promene stanja u kome se konačni automat nalazi. Plavom bojom označeni su trenuci kada dolazi do promene na Murovom izlazu. Zelenom bojom označeni su trenuci kada dolazi do promene na Milijevom izlazu.

Sa slike 3.5 može se uočiti i jedna od glavnih razlika između Milijevih i Murovih izlaza. Milijevi izlazi se generišu brže od Murovih, čim dođe do promene na nekom od ulaza koji kontroliše generisanje Milijevog izlaza. Međutim, svaka promena na nekom od kontrolišućih ulaza, potencijalno rezultuje u promeni odgovarajućeg Milijevog izlaza, što znači da će se i svi gličevi koji su eventualno prisutni na nekom od ulaza preneti i na Milijev izlaz, što može predstavljati problem. Sa druge strane, Murovi izlazi se menjaju samo prilikom nailaska rastuće ivice *clk* signala, što znači da unose kašnjenje od jedne periode *clk* signala, ali se na njima ne mogu pojaviti gličevi.

Ukoliko nam je potrebna brza reakcija automata na promene ulaznih signala, potrebno je koristiti Milijeve izlaze. Ukoliko su nam potrebni izlazni signali konačnog automata koji neće imati gličeve, potrebno je koristiti Murove izlaze.

Reprezentacija konačnih automata

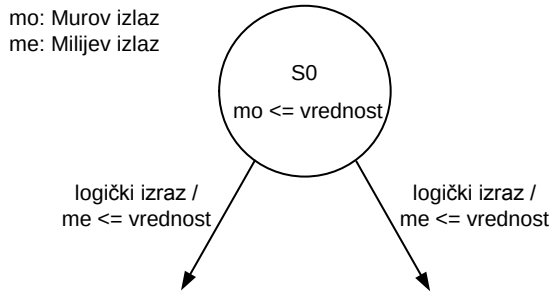
Prilikom specifikacije konačnih automata, što je prvi korak u njihovom modelovanju pomoću VHDL jezika, mogu se koristiti različiti modeli reprezentacije. Sledeća dva modela se najčešće koriste u praksi:

- opis pomoću dijagrama stanja
- opis pomoću ASM dijagrama

Opis pomoću dijagrama stanja

Opis pomoću dijagrama stanja predstavlja grafički način specifikacije željenog rada konačnog automata. Ovaj opis na kompaktan način grupiše sve neophodne informacije za potpuni opis željenog načina rada automata, a s'obzirom da je reč o grafičkom prikazu, opis je daleko razumljivi za dizajnera od tekstualnog ili tabelarnog načina opisa željenog rada konačnog automata.

Dijagram stanja sastoji se od skupa čvorova, predstavljenih pomoću ovalnih simbola, koji su međusobno povezani usmerenim granama. Generička struktura čvora koji se koristi prilikom razvoja dijagrama stanja prikazana je na slici 3.6.



Slika 3.6. Generička struktura čvora unutar dijagrama stanja konačnog automata

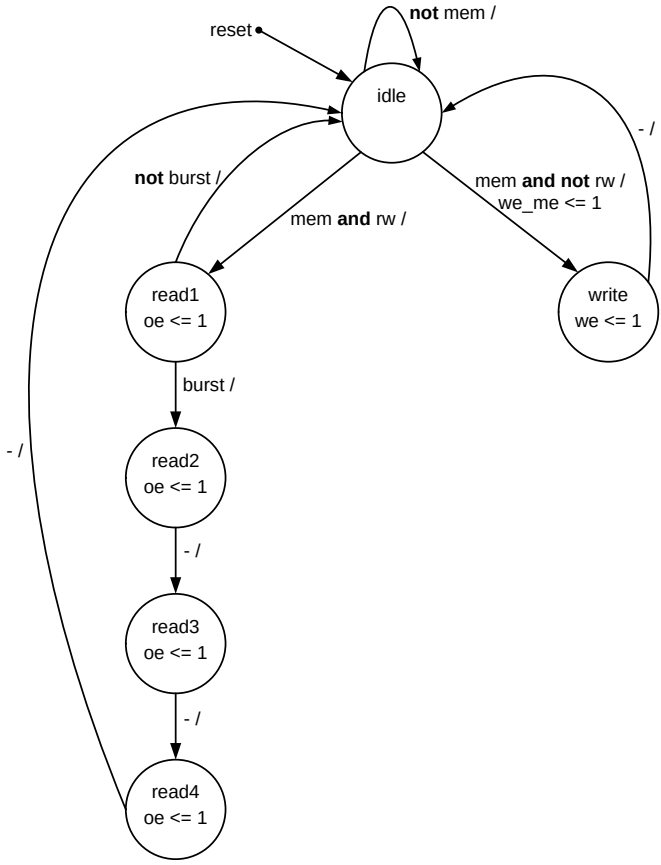
Svaki čvor u dijagramu stanja predstavlja jedno, jedinstveno, stanje u kojem se može naći konačni automat. Svaki čvor mora imati jedinstveno simboličko ime koje mu je pridruženo. Čvor na slici 3.6 ima pridruženo simboličko ime *S0*, koje ujedno predstavlja i identifikaciju stanja u kome se konačni automat nalazi u tom trenutku.

Iz svakog čvora može kretati jedna ili više izlaznih grana. Svaka izlazna grana predstavlja jednu moguću tranziciju iz tekućeg stanja u neko drugo stanje. Pored svake grane može, a ne mora, stajati logički izraz koji definiše uslov pod kojim se vrši posmatrana tranzicija. Logički izraz koji specificira uslov u sebi može da uključuje ulazne portove konačnog automata. Ukoliko iz stanja polazi više od jedne izlazne grane, logički uslovi koji su im pridruženi moraju biti međusobno isključivi. U svakom trenutku mora biti zadovoljen samo jedan od mogućih uslova, jer automat može preći u samo jedno, naredno stanje. Ukoliko iz stanja polazi tačno jedna grana, ona mora biti bezuslovna, pored nje se ne može nalaziti uslov pod kojim će se ova tranzicija obaviti. Ovo je stoga što u svakom trenutku mora biti jednoznačno određeno naredno stanje u kome će se automat naći.

Vrednosti izlaznih portova takođe su specificirane na dijagramu stanja. Nove vrednosti Murovih izlaza navode se unutar oznake čvora (*mo <= vrednost* na slici 3.6), s'obzirom da su oni samo funkcija tekućeg stanja u kome se automat nalazi. S'obzirom da vrednosti Milijevih izlaza zavise ne samo od tekućeg stanja u kome se automat nalazi već i od tekućih vrednosti ulaznih portova automata, nove vrednosti Milijevih izlaza navode se pored svake od izlaznih grana (*me <= vrednost* na slici 3.6).

Da bi se pojednostavio dijagram stanja, konvencija je da se na njemu navode samo dodele izlaznim portovima koji u datom stanju treba da promene svoju vrednost. U slučaju da se izlazni port u datom stanju ne aktivira, on se ne navodi na dijagramu, a podrazumeva se da izlazni port uzima podrazumevanu vrednost. Ovu podrazumevanu vrednost ne treba brkati sa „*don't care*" vrednošću. Uobičajeno je da se za podrazumevanu vrednost izlaznih portova uzima vrednost 0, ukoliko to drugačije nije naglašeno.

Kao ilustracija specifikacije željenog ponašanja konačnog automata pomoću dijagrama stanja, na slici 3.7 prikazan je dijagram stanja konačnog automata hipotetičkog memorijskog kontrolera.



Slika 3.7. Dijagram stanja konačnog automata hipotetičkog memorijskog kontrolera

Hipotetički memorijski kontroler nalazi se između procesora i memorije i zadatak mu je da interpretira komande koje dolaze od procesora i generiše potrebne sekvence upravljačkih signala za memoriju.

Ulazni portovi memorijskog kontrolera su: *mem*, *rw* i *burst*. Svi ulazni portovi su jednobitni i na njih se dovode odgovarajući signali koje generiše procesor. *Mem* ulazni port se aktivira (dovodi na vrednost 1) svaki put kada procesor želi da pristupi memoriji. *Rw* ulazni port predstavlja indikaciju tipa memorijskog ciklusa i ima vrednost 1 ako procesor želi da inicira memorijski ciklus čitanja sadržaja iz memorije, odnosno ima vrednost 0 ukoliko procesor želi da inicira memorijski ciklus upisa u memoriju. *Burst* ulazni port služi za iniciranje specijalnog moda čitanja podataka iz memorije. Kada *burst* ulazni port ima vrednost 1, potrebno je izvršiti četiri uzastopne operacije čitanja podataka iz memorije.

Izlazni portovi memorijskog kontrolera su: *oe* (*output enable*) i *we* (*write enable*). Svi izlazni portovi su takođe jednobitni. Ovi izlazni portovi su povezani sa odgovarajućim ulaznim portovima memorije i služe za kontrolu njenog rada od strane memorijskog kontrolera. Kao što se sa blok dijagrama može zaključiti, oba ova izlazna porta su Murovog tipa. Memorijski kontroler ima još jedan jednobitni izlazni port, *me_we*, koji je Milijevog tipa.

Memorijski kontroler može da se nadje u ukupno šest različitih stanja: *idle*, *read1*, *read2*, *read3*, *read4* i *write*.

Da bi se obezbedila pravilna inicijalizacija memorijskog kontrolera, prisutan je i ulazni port sinhronog reseta, *reset*. Svaki put kada je *reset* port postavljen na 1, memorijski kontroler se vraća u *idle* stanje.

Sam rad memorijskog kontrolera sa slike 3.7 je sledeći. Nakon reseta kontroler se nalazi u *idle* stanju i čeka na aktiviranje *mem* ulaznog porta, što će predstavljati indikaciju da procesor želi da komunicira sa memorijom. Jednom kada se *mem* port aktivira (postane 1), konačni automat analizira trenutnu vrednost *rw* ulaznog porta i u zavisnosti od njegove vrednosti prelazi ili u stanje *read1* ili u stanje *write*. Ovo željeno ponašanje može se modelovati pomoću tri grane koje napuštaju *idle* stanje, kao što je prikazano na slici 3.7:

- Grana uz koju stoji test "**not** *mem*" označava prelaz iz *idle* stanja u *idle* stanje. Dok procesor ne zahteva iniciranje ciklusa pristupa memoriji memorijski kontroler treba da ostane u *idle* stanju i ne generiše nikakve upravljačke sekvence ka memoriji.
- Grana uz koju stoji test "*mem* **and** *rw*" označava prelaz iz *idle* stanja u *read1* stanje. Kada je ovaj uslov ispunjen, procesor zapravo zahteva iniciranje memorijskog ciklusa čitanja iz memorije tako da memorijski kontroler treba da pređe u *read1* stanje i generiše odgovarajuće upravljačke signale za memoriju.
- Grana uz koju stoji test "*mem* **and not** *rw*" označava prelaz iz *idle* stanja u *write* stanje. Kada je ovaj uslov ispunjen, procesor zapravo zahteva iniciranje memorijskog ciklusa upisa u memoriju tako da memorijski kontroler treba da pređe u *write* stanje i generiše odgovarajuće upravljačke signale za memoriju.

Nakon što memorijski kontroler uđe u *read1* stanje proverava se tekuća vrednost *burst* ulaznog porta:

- Ukoliko *burst* ima vrednost 1, procesor zahteva *Burst* pristup memoriji, što u našoj implementaciji podrazumeva čitanje sadržaja četiri sukcesivne memorijske lokacije, tako da memorijski kontroler prelazi u *read2* stanje, a zatim bezuslovno i u *read3* i *read4* stanja. U svim ovim stanjima izlazni port *oe* je aktivan (ima vrednost 1), a port *we* je neaktivan (ima vrednost 0), što predstavlja komandu memoriji da izvrši operaciju čitanja sadržaja iz adresirane memorijske lokacije.

- U slučaju da je *burst* ulazni port neaktivan, kontroler se vraća u *idle* stanje, jer je procesor zahtevao operaciju čitanja sadržaja samo jedne memorijske lokacije.

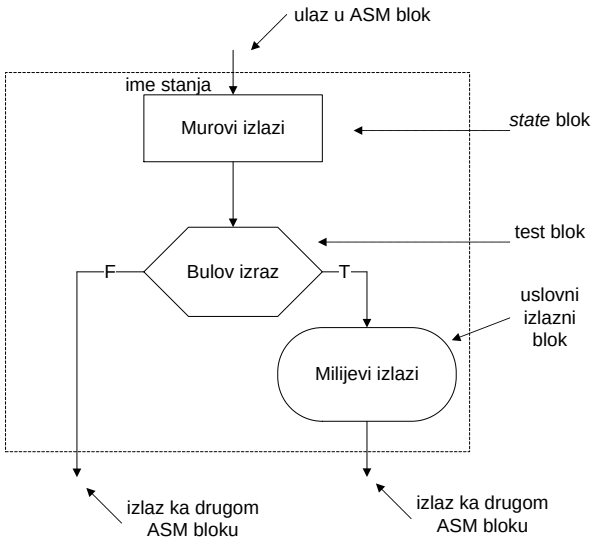
U slučaju da memorijski kontroler uđe u *write* stanje, aktivira izlazni port *we* (postavlja ga na 1) i deaktivira izlazni port *oe* (postavlja ga na nulu), što predstavlja indikaciju memoriji da je potrebno izvršiti operaciju upisa podatka u adresiranu memorijsku lokaciju. Iz *write* stanja memorijski kontroler bezuslovno odlazi u *idle* stanje, jer u našem primeru ne postoji mogućnost *Burst* upisa podataka u memoriju.

We_me izlazni port je zapravo suvišan u radu memorijskog kontrolera, ali postoji u ovom primeru kako bi se ilustrovao rad sa Milijevim izlazima. Ovaj signal biće aktivan samo kada je memorijski kontroler u *idle* stanju, a ulazni portovi *mem* i *rw* imaju vrednost 1 i 0 respektivno. *We_me* izlazni port će biti deaktiviran čim memorijski kontroler pređe iz *idle* stanja u *write* stanje.

Opis pomoću ASM dijagrama

ASM (*Algorithmic State Machine*) dijagram predstavlja alternativni način za reprezenzaciju konačnog automata. Iako ASM dijagram sadrži istu količinu informacija kao i odgovarajući dijagram stanja, način na koji je ona prezentovana u ASM dijagramu čini ga razumljivijim od dijagrama stanja. ASM dijagrame možemo koristiti za specifikaciju složenih sekvenci događaja koje uključuju komande (kao ulaze) i upravljačke akcije (kao izlaze), što ih čini idealnim načinom za modelovanje kontrolnih procesa unutar složenih algoritama koje želimo da implementiramo u hardveru. Ono što ASM dijagrame čini posebno interesantnim je da se oni vrlo lako mogu transformisati u odgovarajući VHDL model kontrolera.

ASM dijagram sastoji se iz odgovarajućeg broja povezanih ASM blokova. Struktura ASM bloka prikazana je na slici 3.8.



Slika 3.8. Struktura ASM bloka

Kao što se sa slike 3.8 može videti, svaki ASM blok sastoji se iz:

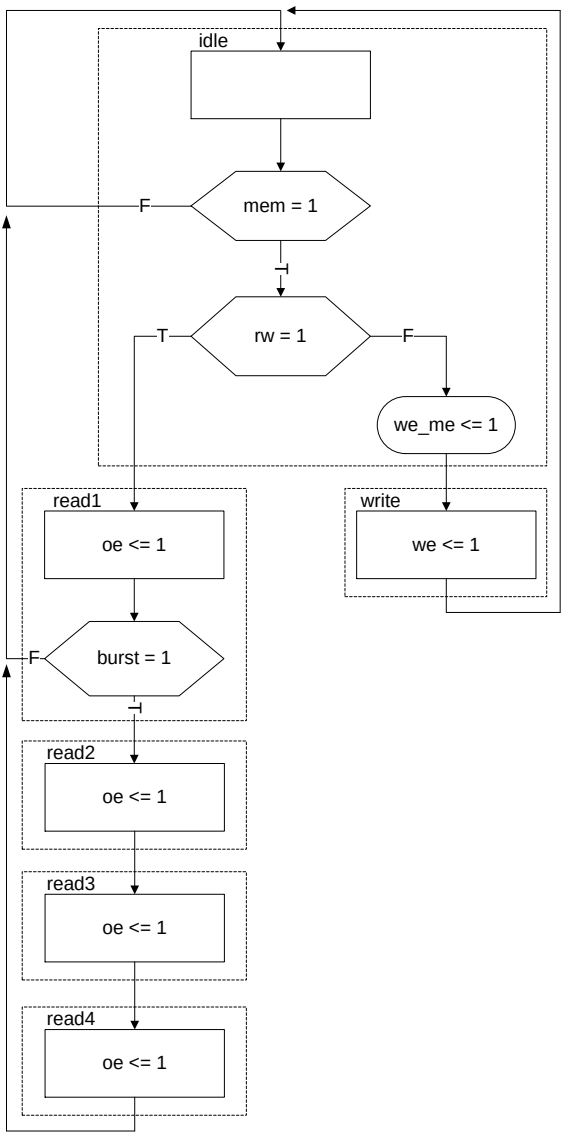
- jednog *state* bloka koji predstavlja stanje pridruženo posmatranom ASM bloku i pomoću kojega se modeluje logika generisanja Murovih izlaza iz konačnog automata,
- opcione mreže test blokova, koji modeluju proces izbora izlazne putanje koja će biti korišćena za napuštanje posmatranog ASM bloka,
- opcionog broja uslovnih izlaznih blokova, pomoću kojih se modeluje logika generisanja Milijevih izlaza konačnog automata.

U svakom trenutku konačni automat može se naći u samo jednom od mogućih ASM blokova. Dok se nalazi u odgovarajućem ASM bloku, konačni automat generiše sve Murove izlaze (ako oni postoje), dodeljujući im vrednosti koje su specificirane unutar *state* bloka aktivnog ASM bloka. Nakon toga, u istom trenutku, konačni automat evaluira testove koji su navedeni u test blokovima (ako oni postoje) i u zavisnosti od njihovog ishoda bira tačno jednu izlaznu granu preko koje napušta tekući ASM blok i u narednom trenutku ulazi u odabrani ASM blok. Pored toga, ukoliko na odabranoj izlaznoj putanji postoje uslovni izlazni blokovi, konačni automat generiše i odgovarajuće Milijeve izlaze, dodeljujući im vrednosti koje su specificirane u odgovarajućim uslovnim izlaznim blokovima.

Iz prethodne analize možemo primetiti da je opis rada konačnog automata pomoću ASM dijagrama vrlo sličan opisu pomoću dijagrama stanja. Suštinska razlika ogleda se u načinu kako se određuje naredno stanje u kojem konačni automat treba da se nađe. U

slučaju dijagrama stanja logika određivanja narednog stanja je distribuirana pored svih izlaznih grana koje napuštaju tekuće stanje i samim tim teže razumljiva. U slučaju ASM dijagrama logika određivanja narednog stanja jasno je vidljiva, u formi mreže test blokova, koja ima strukturu blok dijagrama algoritma (otuda dolazi i naziv za ovaj vid reprezentacije konačnih automata, *Algorithmic State Machine*), te je vrlo razumljiva. Kako je logika ovaj put skoncentrisana na jednom mestu i prikazana u poznatoj formi (blok dijagram algoritma), dizajner je može daleko lakše razumeti, kreirati ili modifikovati, i pretočiti u odgovarajući HDL model, što i predstavlja najveću prednost prilikom korišćenja ASM dijagrama.

Kako je ASM dijagram samo jedan od načina za reprezentaciju konačnih automata, kao i dijagram stanja, svaki ASM dijagram može se prevesti u odgovarajući dijagram stanja, i obrnuto. Svaki ASM blok u ASM dijagramu odgovara jednom čvoru u dijagramu stanja, a njegove izlazne putanje odgovaraju odgovarajućim izlaznim granama unutar dijagrama stanja. Na primer, na slici 3.9 prikazan je ASM dijagram istog hipotetičkog memorijskog kontrolera, čiji dijagram stanja je prikazan ranije, na slici 3.7.



Slika 3.9. ASM dijagram konačnog automata hipotetičkog memorijskog kontrolera

Realizacija konačnih automata

Nakon što se željena funkcionalnost konačnog automata modeluje korišćenjem dijagrama stanja ili ASM dijagrama, može se pristupiti njegovoj realizaciji pomoću digitalnog elektronskog sistema. Prvi korak prilikom realizacije, nezavisno od odabrane arhitekture za realizaciju, jeste da se odredi način kodovanja simboličkih stanja

konačnog automata pomoću binarnih reči konačne dužine. Optimalan izbor kodovanja je NP kompletan problem, tako da za njegovo rešavanje ne postoje efikasni algoritmi, već se koriste različite varijante heurističkih algoritama. Većina alata za automatsku sintezu hardvera ima mogućnost automatskog određivanja „optimalnog“ načina kodovanja stanja konačnog automata, pod odabranim ograničenjima (tipično se može zahtevati da se kodovanje izvrši na takav način da rezultujući automat ima maksimalnu brzinu rada ili zahteva minimalni broj hardverskih resursa), pa je u većini slučajeva najbolje koristiti ovu mogućnost. Manuelni izbor načina kodovanja treba izbegavati, osim u izuzetnim slučajevima.

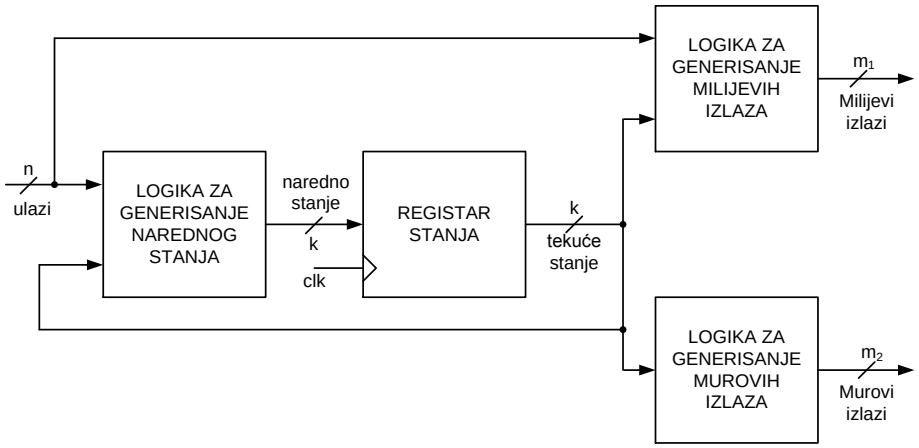
Iako se konačni automati mogu realizovati korišćenjem različitih arhitektura, u praksi se najčešće koriste dve kanoničke arhitekture:

- „*Random Logic*“ arhitektura,
- Mikroprogramirana arhitektura.

Kada se konačni automat mapira u odabranu arhitekturu, ostaje da se napiše njen HDL model, koji se zatim automatski prevodi u digitalni elektronski sistem, korišćenjem nekog od kompajlera za sintezu hardvera.

3.2.1. Realizacija korišćenjem “*Random Logic*” arhitekture

Prilikom realizacije konačnog automata korišćenje „*Random Logic*“ arhitekture, automat se realizuje pomoću sekvencijalne mreže, čija je struktura prikazana na slici 3.10.



Slika 3.10. Realizacija konačnog automata korišćenjem „Random Logic“ arhitekture

„*Random Logic*“ arhitektura sastoji se iz sledećih blokova:

- **Registra stanja** – sekvencijalne mreže u kojoj se čuva binarni kod tekućeg stanja u kojem se automat trenutno nalazi
- **Logike za generisanje narednog stanja** – kombinacione mreže pomoću koje se izračunava binarni kod narednog stanja u kojem automat treba da se nađe
- **Logike za generisanje Milijevih izlaza** – kombinacione mreže pomoću koje se izračunavaju vrednosti Milijevih izlaza automata. Ovaj blok je opcioni i ne postoji ukoliko automat koji se realizuje nema Milijeve izlaze.
- **Logike za generisanje Murovih izlaza** – kombinacione mreže pomoću koje se izračunavaju vrednosti Murovih izlaza automata. Ovaj blok je opcioni i ne postoji ukoliko automat koji se realizuje nema Murove izlaze.

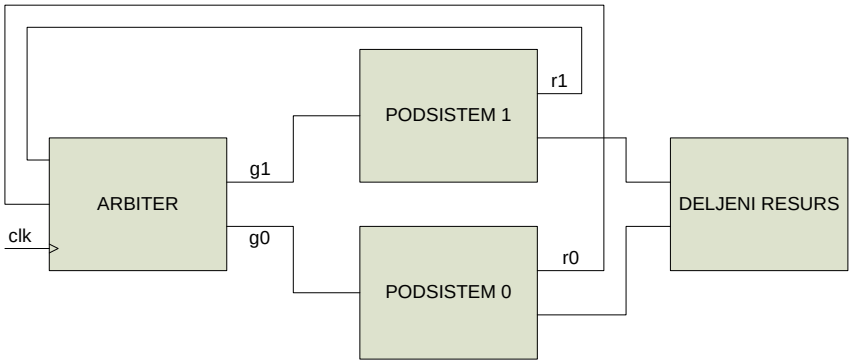
Naziv „*Random Logic*“ dolazi od činjenice da je struktura modula za generisanje narednog stanja u slučaju realizacije proizvoljnog automata potpuno iregularna (samim tim na neki način se može reći da je ona slučajna, ali ne u smislu slučajnog rada, već u smislu njene strukture koja se teško može unapred predvideti), za razliku od ostalih standardnih sekvencijalnih mreža (registri, brojači, itd.), kod kojih je ova logika vrlo regularna.

Kako je svaki od gradivnih blokova „*Random Logic*“ arhitekture standardna kombinaciona ili sekvencijalna mreža, razvoj odgovarajućeg HDL model ne predstavlja naročiti problem.

Zadaci za vežbu

Zadatak 3.2.

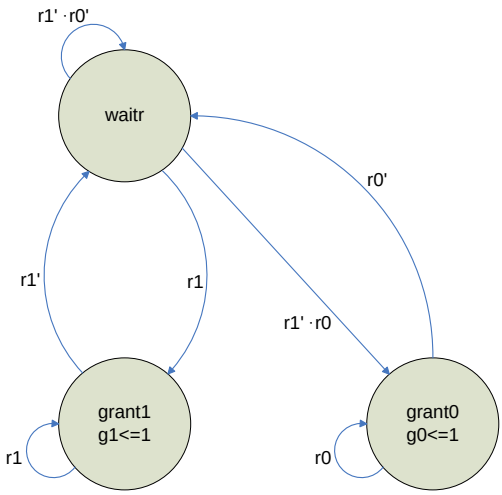
U složenim sistemima, neki od resursa su deljeni između većeg broja podsistema. Na primer, nekoliko procesora mogu deliti istu memoriju, ili, na primer, veći broj periferijskih jedinica može biti priključen na istu, zajedničku magistralu. Arbiter je kolo čija je funkcija da razrešava konflikte koji se mogu pojaviti između podsistema koji žele koristiti deljeni resurs. Arbiter takođe koordinira i kontroliše pristup ka deljenom resursu. Primer koji ćemo mi razmatrati odnosi se na arbiter sa dva podsistema prikazan na slici 3.11.



Slika 3.11. Blok dijagram složenog sistema sa arbiterom

Podsistemi komuniciraju sa arbiterom pomoću *request* i *grant* signala, koji su označeni sa *r(1)* i *g(1)* za podsistem 1, odnosno, sa *r(0)* i *g(0)* za podsistem 0. Kada neki od podsistema želi da pristupi deljenom resursu, aktivira svoj *request* signal. Arbiter nadgleda korišćenje deljenog resursa i *request* signale, i na osnovu njih dodeljuje pristup nekom od podsistema aktiviranjem odgovarajućeg *grant* signala. Kada je njegov *grant* signal aktiviran, podsistem ima dozvolu da pristupi deljenom resursu. Nakon što je završio rad sa deljenim resursom, podsistem deaktivira svoj *request* signal i prestaje sa korišćenjem deljenog resursa. S'obzirom da arbiter odluku o dodeljivanju deljenog resursa delom donosi na osnovu događaja koji su se desili u prošlosti (na osnovu prethodnih vrednosti *request* i *grant* signala), arbiter mora posedovati neke memorijske elemente u kojima će sačuvati informacije o tome šta se desilo sa *request* i *grant* signalima u prošlosti. Konačni automat može u potpunosti zadovoljiti ovaj zahtev. Zbog toga ćemo arbiter i projektovati kao jedan konačni automat.

Jedno od ključnih pitanja prilikom projektovanja arbitera jeste obrada istovremenih zahteva od strane podsistema. Da bi se rešio ovaj problem mora se uvesti neki sistem prioriteta u zahtevima različitih podsistema. Prvi dizajn koji ćemo razmotriti uvek daje prioritet podsistemu 1. Dijagram stanja konačnog automata u ovom slučaju prikazan je na slici 3.12. Obratite pažnju da su na slici negirani signali označeni sa apostrofom (').



Slika 3.12. Dijagram stanja konačnog automata arbitera sa fiksnim prioritetom

a) Napisati VHDL model arbitera realizovanog kao konačni automat koristeći dijagram stanja sa slike 3.12.

b) Napisati jednostavan testbenč koji će izvršiti verifikaciju arbitera projektovanog pod a). Kao polazna tačka u razvoju testbenča može se koristiti blok dijagram sa slike 3.11, pri čemu nije neophodno modelovati deljeni resurs.

Zadatak 3.3.

Dizajn arbitera iz zadatka 3.2 uvek daje prednost podsistemu 1 prilikom pristupa deljenom resursu. Ovakav pristup može predstavljati problem ukoliko podsistem 1 konstantno zahteva pristup deljenom resursu. U tom slučaju podsistem 0 nikad ne bi imao priliku da koristi deljeni resurs. Moguće je prepraviti dijagram stanja konačnog automata sa slike 3.12 tako da arbiter realizuje pošteniju arbitracionu politiku. Nova politika arbitracije vodi računa o tome koji je podsistem dobio dozvolu za korišćenje deljenog resursa poslednji put, tako da u slučaju da oba podsistema istovremeno zahtevaju pristup deljenom resursu on se dodeljuje onom podsistemu koji poslednji put nije dobio dozvolu za korišćenje deljenog resursa. Novi dizajn mora posedovati dva različita *waitr* stanja, *waitr0* i *waitr1*. Kada se arbiter nalazi u *waitr0* stanju to znači da je podsistem 1 poslednji koristio deljeni resurs, te ukoliko dođe do istovremenog zahteva prednost treba dati podsistemu 0. Ukoliko se arbiter nalazi u *waitr1* stanju logika rada je suprotna od one opisane za *waitr0* stanje. Dodatna modifikacija koja je neophodna odnosi se na stanje u koje arbiter prelazi kada napušta *grant0*, odnosno *grant1* stanja. Ukoliko arbiter napušta stanje *grant0* potrebno je da pređe u stanje *waitr1* kako bi prilikom naredne dodele favorizovao podsistem 1. Slično pravilo važi i za prelaz iz stanja *grant1*.

a) Nacrtati dijagram stanja za arbiter koji će realizovati prethodno opisanu arbitracionu politiku.

b) Na osnovu dijagrama stanja nacrtati ASM dijagram.

c) Koristeći ASM dijagram napisati VHDL model arbitera.

d) Modifikovati testbenč iz zadatka 3.2 i izvršiti verifikaciju novog arbitera.

Zadatak 3.4.

Arbiter sa fiksnim prioritetom iz zadatka 3.2, čiji dijagram stanja je prikazan na slici 3.12, svaki put se mora vratiti u *waitr* stanje pre nego što dodeli deljeni resurs na naredni zahtev od strane nekog podsistema. Preraditi dijagram stanja sa slike 3.12 tako da arbiter može da se kreće iz jednog u drugo *grant* stanje bez potrebe za vraćanjem u *waitr* stanje ukoliko postoji aktivni zahtev za korišćenjem deljenog resursa.

a) Nacrtati dijagram stanja novog arbitera.

b) Na osnovu dijagrama stanja nacrtati ASM dijagram.

c) Koristeći ASM dijagram napisati VHDL model arbitera.

d) Modifikovati testbenč iz zadatka 3.2 i izvršiti verifikaciju novog arbitera.

Zadatak 3.5.

Posmatrajmo arbiter sa fer prioritetima definisan u zadatku 3.3. On se bazira na pretpostavci da će podsistem koji trenutno koristi deljeni resurs u jednom trenutku dobrovoljno prestati da ga koristi. Alternativa bi bila da se uvede takozvani *timeout* signal koji onemogućava dogutrajno korišćenje deljenog resursa od strane bilo kog podsistema, kada mu se deljeni resurs dodeli. Kada se *timeout* signal aktivira, arbiter se vraća u stanje čekanja bez obzira da li je odgovarajući *request* signal i dalje aktivan.

a) Nacrtati prerađeni dijagram stanja koji uključuje i postojanje *timeout* signala.

b) Na osnovu dijagrama stanja nacrtati ASM dijagram.

c) Koristeći ASM dijagram napisati VHDL model arbitera.

d) Modifikovati testbenč iz zadatka 3.3 i izvršiti verifikaciju novog arbitera.

Zadatak 3.6.

RAM memorija se često koristi za skladištenje velike količine podataka unutar digitalnog sistema. Najčešće korišćeni tip RAM memorije predstavlja asinhrona statička RAM (SRAM) memorija. Za razliku od registra, kod kojega se sve operacije izvode sinhrono sa klok signalom (tipično na rastuću ivicu), pristup podacima unutar asinhrone SRAM memorije je složeniji. Operacije čitanja i upisa zahtevaju da se podaci, adresa i

kontrolni signali postave u specifičnom redosledu, a zatim drže stabilnim odgovarajući period vremena da bi tekuća operacija bila uspešna.

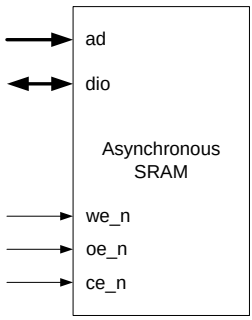
Sinhronom sistemu je teško da direktno pristupa SRAM memoriji. Namena *memorijskog kontrolera* jeste da predstavlja interfejs između sinhronog sistema i SRAM memorije. Memorijski kontroler uzima komande od glavnog sistema sinhrono, a zatim generiše vremenske signale za pristup SRAM memoriji. Kontroler „štiti“ glavni sistem od detalja vezanih za vremenski pristup SRAM memoriji, na taj način čineći da pristup memoriji izgleda kao sinhrona operacija. Performanse memorijskog kontrolera mere se brojem memorijskih pristupa koji se mogu kompletirati u posmatranom vremenskom intervalu. Iako je projektovanje prostog memorijskog kontrolera relativno jednostavan zadatak, ostvarivanje optimalnih performansi može predstavljati ozbiljan zadatak.

Na ovom mestu biće izvršeno projektovanje jednostavnog memorijskog kontrolera za jednu hipotetičku asinhronu SRAM memoriju. S'obzirom da su vremenske karakteristike svake SRAM memorije drugačije, projektovani kontroler bi se morao prilagoditi karakteristikama odabrane SRAM komponente.

Interfejs asinhrone SRAM memorije

Interfejs tipične asinhrone SRAM memorije prikazan je na slici 3.13 i sastoji se od adresne i magistrale podataka, kao i od sledećih kontrolnih signala:

- **ce_n (chip enable)** – signal dozvole pristupa SRAM memoriji
- **we_n (write enable)** – signal dozvole upisa u SRAM memoriju
- **oe_n (output enable)** – signal dozvole izlaza



Slika 3.13. Interfejs tipične asinhrone SRAM memorije

Kao što se na osnovu sufiksa *_n* može primetiti, svi kontrolni signali su aktivni na niskom logičkom nivou. Način pristupa asinhronoj SRAM memoriji korišćenjem ovih kontrolnih signala prikazan je u tabeli 3.1.

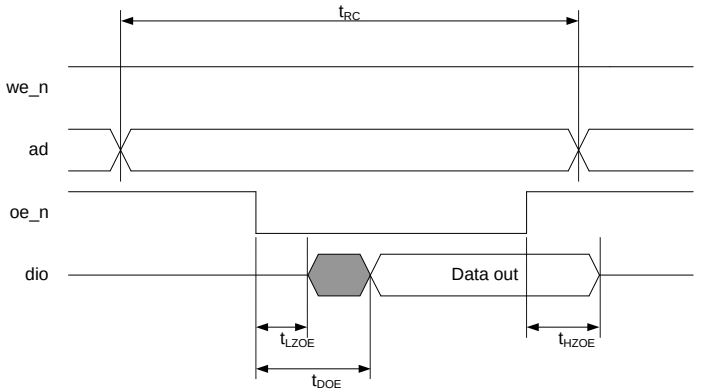
Operacija	ce_n	we_n	oe_n	dio
Neaktivna	1	-	-	Z
	0	1	1	Z
Čitanje	0	1	0	Data out
Upis	0	0	-	Data in

Tabela 3.1. Funkcionalna tabela asinhrone SRAM memorije

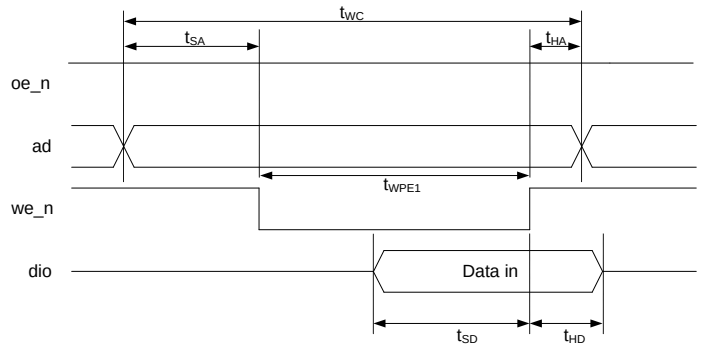
Vremenski parametri asinhrone SRAM memorije

Vremenske karakteristike asinhrone SRAM memorije su prilično složene i uključuju veliki broj različitih parametara. Mi ćemo ovde spomenuti samo one koji su relevantni za naš dizajn.

Uprošćeni vremenski dijagram operacija čitanja i upisa u asinhronu SRAM memoriju prikazani su na slici 3.14.



a) Vremenski dijagram ciklusa čitanja podatka iz SRAM memorije



b) Vremenski dijagram ciklusa upisa podatka u SRAM memoriju

Slika 3.14. Vremenski dijagrami čitanja i upisa u asinhronu SRAM memoriju

Relevantni vremenski parametri su:

- $\mathbf{t_{RC}}$, **read cycle time** – minimalno vreme koje mora da protekne između dva uzastopna ciklusa čitanja.
- $\mathbf{t_{DOE}}$, **output enable access time** – vreme potrebno da se na izlazu asinhrone SRAM memorije pojavi validan podatak nakon što se *oe_n* aktivira.
- $\mathbf{t_{HZOE}}$, **output enable to high-Z time** – vreme potrebno da *tri-state* bafer uđe u stanje visoke impedanse nakon što se *oe_n* deaktivira.
- $\mathbf{t_{LZOE}}$, **output enable to low-Z time** – vreme potrebno da *tri-state* bafer napusti stanje visoke impedanse nakon što se *oe_n* aktivira. Obratite pažnju da čak i kada magistrala podataka izađe iz stanja visoke impedanse, podaci na magistrali podataka tipično još neko vreme nisu validni (označeno sivom bojom na dijagramu).
- $\mathbf{t_{WC}}$, **write cycle time** – minimalno vreme koje mora da protekne između dva uzastopna ciklusa upisa.
- $\mathbf{t_{SA}}$, **address setup time** – minimalno vreme tokom kojega adresa mora biti stabilna pre nego što se *we_n* aktivira.
- $\mathbf{t_{HA}}$, **address hold time** – minimalno vreme tokom kojega adresa mora biti stabilna nakon što se *we_n* deaktivira.
- $\mathbf{t_{WPE1}}$, **we_n pulse width** – minimalno vreme tokom kojega *we_n* signal mora biti aktivan.
- $\mathbf{t_{SD}}$, **data setup time** – minimalno vreme tokom kojega podaci moraju biti stabilni pre njihovog zahvatanja od strane SRAM memorije (podaci se zahvataju od strane SRAM memorije u trenutku nailaska rastuće ivice na *we_n* signalu).
- $\mathbf{t_{HD}}$, **data hold time** – minimalno vreme tokom kojega podaci moraju biti stabilni nakon njihovog zahvatanja od strane SRAM memorije.

Tipične vrednosti ovih parametara prikazane su u tabeli 3.2.

Parametar	min [ns]	max [ns]
t_{RC}, read cycle time	10	-
t_{DOE}, output enable access time	-	4
t_{HZOE}, output enable to high-Z time	-	4
t_{LZOE}, output enable to low-Z time	0	-
t_{WC}, write cycle time	10	-
t_{SA}, address setup time	0	-
t_{HA}, address hold time	0	-
t_{WPE1}, we_n pulse width	8	-
t_{SD}, data setup time	6	-
t_{HD}, data hold time	0	-

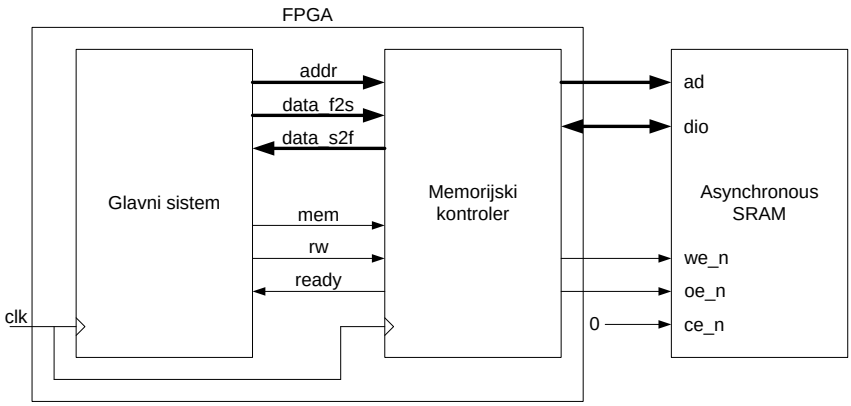
Tabela 3.2. Tipične vrednosti vremenskih parametara asinhrone SRAM memorije

Još jednom treba napomenuti da su ovo samo najvažniji vremenski parametri na osnovu kojih ćemo projektovati jednostavni memorijski kontroler, i da se vrednosti iz tabele 3.2 ne odnose ni na jednu konkretnu komponentu. Kompletne informacije potrebno je potražiti u dokumentaciji proizvođača za konkretnu asinhronu SRAM memoriju koja se planira koristi.

Jednostavan memorijski kontroler

Uloga i način povezivanja memorijskog kontrolera sa ostalim komponentama složenog sistema prikazani su na slici 3.15. Interfejs između asinhrone SRAM memorije i memorijskog kontrolera diskutovani su ranije, tako da ćemo na ovom mestu razmotriti signale koji predstavljaju interfejs između glavnog sistema i memorijskog kontrolera:

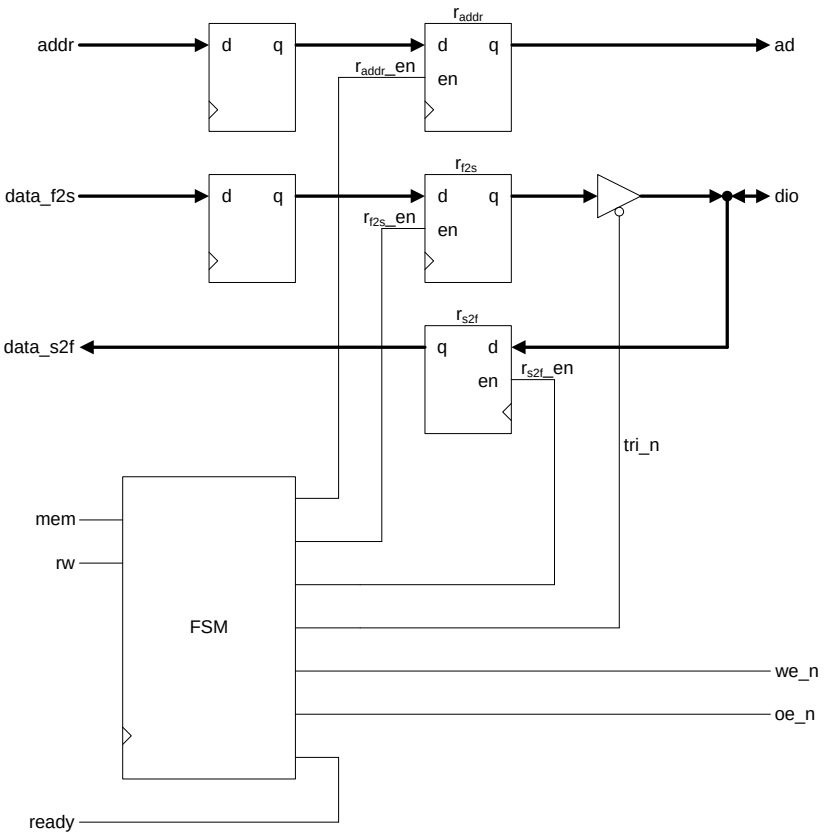
- **mem** – signal pomoću kojega glavni sistem inicira operaciju pristupa SRAM memoriji. Kada je ovaj signal aktivan (ima vrednost ’1’), to je znak da je inicirana operacija pristupa memoriji.
- **rw** – signal koji određuje da li je tekuća operacija pristupa SRAM memoriji operacija čitanja podatka (’1’) ili operacija upisa podatka (’0’).
- **addr** – adresna magistrala.
- **data_f2s** – magistrala podataka preko koje glavni sistem šalje podatak koji želi upisati u SRAM memoriju.
- **data_s2f** – magistrala podataka preko koje glavni sistem prihvata podatak koji je želeo da pročita iz SRAM memorije.
- **ready** – statusni signal koji glavnom sistemu pruža informaciju da li je memorijski kontroler spreman da prihvati novu komandu. Ovaj signal je neophodan jer memorijska operacija može trajati (tipično i traje) duže od jedne periode klok signala.



Slika 3.15. Uloga SRAM memorijskog kontrolera

Memorijski kontroler zapravo predstavlja blok koji transformiše asinhronu SRAM memoriju u sinhroni modul, gledano sa strane glavnog sistema. Kao što je ranije već rečeno, glavni sistem može biti bilo kakav složeni digitalni sistem koji se realizovan pomoću FPGA komponente. Kada glavni sistem želi da pristupi memoriji, on postavlja adresu i podatak (u slučaju operacije upisa) na odgovarajuće magistrale i aktivira komandu upisa (pomoću *mem* i *rw* signala). Na rastuću ivicu klok signala, vrednosti ovih signala se sempluju i izvršava se željena operacija. U slučaju operacije čitanja podatka, pročitani podatak postaje dostupan glavnom sistemu (preko *data_s2f* magistrale) nakon odgovarajućeg broja perioda klok signala (koji zavisi od načina realizacije memorijskog kontrolera).

Blok dijagram jednostavnog memorijskog kontrolera koji ćemo mi razmatrati prikazan je na slici 3.16. Memorijski kontroler se sastoji iz jednog adresnog registra, koji služi za smeštanje adrese memorijske lokacije kojoj je potrebno pristupiti i dva registra koji služe za čuvanje podataka koji se upisuju, odnosno, čitaju iz SRAM memorije. Obzirom da je *dio* magistrala bidirekciona, neophodno je koristiti i *tri-state* bafer. Čitavim sistemom upravlja upravljačka jedinica realizovana kao konačni automat koja je projektovana tako da zadovolji propisane vremenske dijagrame i generiše odgovarajuće kontrolne signale prilikom operacije upisa, odnosno čitanja, iz asinhrone SRAM memorije, prikazane na slici 3.14.



Slika 3.16. Blok dijagram jednostavnog SRAM memorijskog kontrolera

Iako vremenski dijagrami upisa i čitanja podatka iz asinhrone SRAM memorije na prvi pogled deluju komplikovano, upravljačke sekvence su prilično jednostavne. Razmotrimo prvo operaciju čitanja podatka iz SRAM memorije. Kao prvo, tokom čitave operacije *we_n* signal treba da bude neaktivan (postavljen na vrednost '1'). Potrebni koraci prilikom čitanja podatka iz memorije su:

1. Postavi adresu memorijske lokacije kojoj se želi pristupiti na *ad* magistralu i aktiviraj *oe_n* signal. Ova dva signala moraju biti stabilna tokom čitave operacije čitanja podatka iz memorije.
2. Sačekaj najmanje t_{RC} vremena. Nakon isteka ovog vremena željeni podatak pojaviće se na *dio* magistrali SRAM memorije.
3. Pokupi podatak sa *dio* magistrale i deaktiviraj *oe_n* signal.

U slučaju operacije upisa podatka u SRAM memoriju *oe_n* signal treba da bude neaktivan za celokupno vreme trajanja operacije. Potrebni koraci u slučaju operacije upisa su:

1. Postavi adresu memorijske lokacije kojoj se želi pristupiti na *ad* magistralu, podatak koji se želi upisati u memoriju na *dio* magistralu i aktiviraj *we_n* signal. Ova dva signala moraju biti stabilna tokom čitave operacije upisa podatka u memoriju.
2. Sačekaj najmanje t_{PWE1} vremena.
3. Deaktiviraj *we_n* signal. Podatak koji želimo upisati u memoriju biće prihvaćen u trenutku nailaska rastuće ivice na *we_n* signalu.
4. Skloni podatak sa *dio* magistrale.

Poslednji korak u realizaciji memorijskog kontrolera prikazanog na slici 3.16, predstavlja projektovanje upravljačke jedinice. Dizajn koji će ovde biti prikazan predstavlja „sigurni“ dizajn, u smislu da obezbeđuje dovoljno velike vremenske margine prilikom izvođenja operacije čitanja i upisa u memoriju. Upravljački signali direktno se generišu unutar upravljačke jedinice. Ovako projektovani kontroler zahteva tri perioda klok signala za kompletiranje ciklusa čitanja ili upisa u memoriju. Ukoliko pretpostavimo da će memorijski kontroler raditi na učestanosti od 50 MHz, ciklusi čitanja i upisa u SRAM memoriju trajaće po 60 ns, što je znatno duže od minimalnih trajanja koja iznose 10 ns (vidi tabelu 2, parametre t_{RC} i t_{WC}).

ASM dijagram upravljačke jedinice prikazan je na slici 3.17. Upravljačka jedinica ima ukupno 7 stanja i inicijalno se nalazi u *idle* stanju. Memorijska operacija započinje kada se aktivira *mem* signal, a *rw* signal određuje da li je reč o operaciji čitanja ili upisa podatka u asinhronu SRAM memoriju.

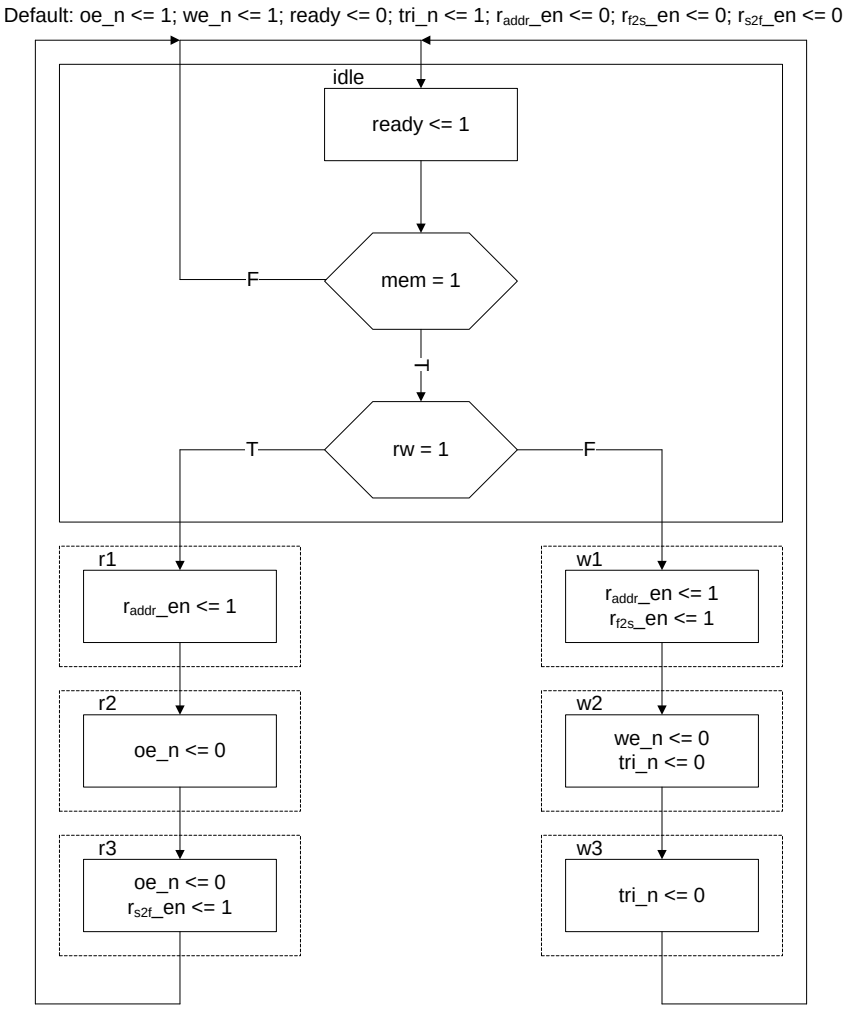
U slučaju operacije čitanja (*rw* = 1), upravljačka jedinica prelazi u *r1* stanje. U ovom stanju se adresa memorijske lokacije kojoj je potrebno pristupiti, *addr*, smešta u r_{addr} registar. Signal *oe_n* je aktivan u stanjima *r2* i *r3*. U *r3* stanju u r_{s2f} registar se smešta podatak koji je pročitan iz SRAM memorije i on postaje dostupan preko *data_s2f* magistrale. Na kraju ciklusa čitanja upravljačka jedinica se vraća u *idle* stanje.

U slučaju operacije upisa (*rw* = 0), upravljačka jedinica prelazi u *w1* stanje, u kojem se adresa memorijske lokacije kojoj se želi pristupiti i podatak koji se želi upisati smeštaju u r_{addr} i r_{f2s} registre. U *w2* stanju aktiviraju se *we_n* i *tri_n* signali. Signal *tri_n* aktivira *tri-state* bafer koji omogućava da se podatak koji se želi upisati u SRAM memoriju pojavi na *dio* magistrali. U w3 stanju deaktivira se *we_n* signal, ali *tri_n* signal i dalje ostaje aktivan. Ovo obezbeđuje pravilno prihvatanje podatka od strane SRAM memorije, prilikom nailaska rastuće ivice na *we_n* signalu. Na kraju ciklusa upisa upravljačka jedinica se vraća u *idle* stanje.

Da bi se obezbedio pravilan rad memorijskog kontrolera, potrebno je proveriti da li predloženi dizajn ispunjava neophodne vremenske zahteve SRAM memorije (vidi sliku 3.14). Pretpostavimo da će upravljačka jedinica raditi na učestanosti od 50 MHz, odnosno da će se u svakom stanju zadržati 20 ns.

Tokom operacije čitanja podatka iz memorije, *oe_n* signal aktivan je tokom dva uzastopna stanja (*r2* i *r3*), odnosno traje ukupno 40 ns što je znatno duže od minimalne zahtevane veličine koja iznosi 10 ns (vidi vremenski parametar t_{RC}).

Tokom operacije upisa podatka u memoriju, *we_n* signal je aktivan samo u *w2* stanju, odnosno traje 20 ns. Ova vrednost je i dalje značajno veća od minimalno zahtevane koja iznosi 8 ns (vidi vremenski parametar t_{WPE1}).



Slika 3.17. ASM dijagram upravljačke jedinice SRAM memorijskog kontrolera

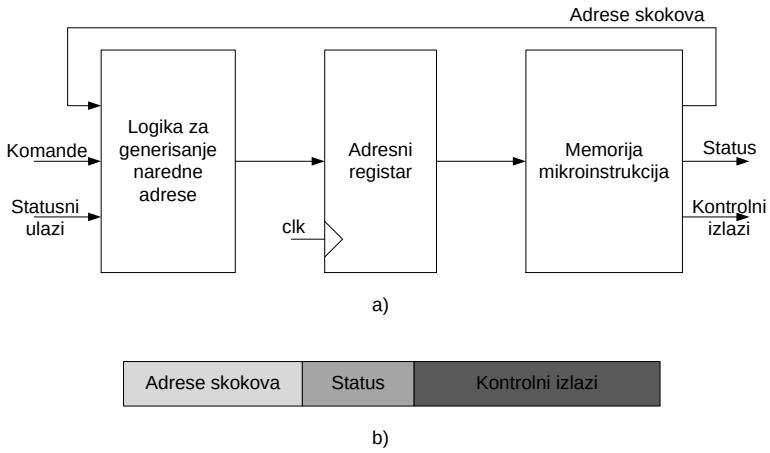
a) Na osnovu ASM dijagrama sa Slike 3.17 napisati VHDL model upravljačke jedinice. Upravljačku jedinicu realizovati koristeći „*Random Logic*“ arhitekturu.

b) Napisati jednostavan testbenč koji će izvršiti verifikaciju upravljačke jedinice projektovane pod a).

3.2.2. Realizacija korišćenjem mikroprogramirane arhitekture

Na slici 3.18a prikazana je generička mikroprogramirana arhitektura za realizaciju konačnog automata. Na ovoj slici pretpostavljeno je da se konačni automat koristi za realizaciju *controlpath* modula, pa su stoga i navedeni specifični ulazni i izlazni signali koji se u tom slučaju koriste.

Osnovna ideja mikroprogramiranja je zamena logike za generisanje narednog stanja iz „*Random Logic*“ arhitekture sa programabilnom memorijom mikroinstrukcija.



Slika 3.18. Mikroprogramirana arhitektura za realizaciju konačnog automata: a) generička struktura mikroprogramirane arhitekture za realizaciju konačnog automata; b) format mikroinstrukcije

Kao što se sa slike 3.18a može videti, mikroprogramirana arhitektura za realizaciju konačnog automata sastoji se iz tri bloka:

- **Logike za generisanje naredne adrese** – ova kombinaciona mreža zadužena je za određivanje adrese naredne mikroinstrukcije koja će biti zahvaćena iz memorije mikroinstrukcija. Na osnovu statusnih ulaza koji dolaze od *datapath* modula, komandi koje generiše okolna logika i skupa adresa skokova koje predstavljaju deo svake mikroinstrukcije, generiše se vrednost naredne adrese

memorijske lokacije unutar memorije mikrooperacija kojoj je potrebno pristupiti u narednom ciklusu.

- **Adresnog registra** – ovaj registar sa paralelnim ulazom i izlazom čuva adresu tekuće mikroinstrukcije koja se pojavljuje na izlazima memorija mikroinstrukcija. Adresni registar ekvivalentan je programskom brojaču koji se nalazi unutar svakog mikroprocesora.
- **Memorije mikroinstrukcija** – programabilna memorija u kojoj se nalaze smeštene mikroinstrukcije. Svakom stanju konačnog automata odgovara tačno jedna mikroinstrukcija. Broj mikroinstrukcija potrebnih za realizaciju konačnog automata stoga je jednak broju stanja posmatranog konačnog automata. Tipičan format mikroinstrukcije prikazan je na slici 3.18b. On se obično sastoji iz tri polja:
    - **Adrese skokova** – ovo polje sadrži adrese narednih mikroinstrukcija, koje odgovaraju stanjima koja su dostižna iz tekućeg stanja u kojem se nalazi konačni automat, u jednom skoku. Broj različitih adresa koje su smeštene unutar ovog polja odgovara maksimalnom broju izlaznih grana iz svakog stanja konačnog automata.
    - **Status** – ovo polje sadrži vrednosti izlaznih statusnih signala pomoću kojih konačni automat svom okruženju saopštava informacije o svom tekućem statusu.
    - **Kontrolnih izlaza** - pomoću ovog polja konačni automat generiše upravljačke signale pomoću kojih upravlja radnom asociranog *datapath* modula.

Navedena tri bloka zajedno obezbeđuju mehanizam koji je ekvivalentan mehanizmu prihvata instrukcija ununar mikroprocesora. Svaka mikroinstrukcija izvršava se tokom jedne periode klok signala. U okviru svake periode klok signala odigravaju se sledeće aktivnosti:

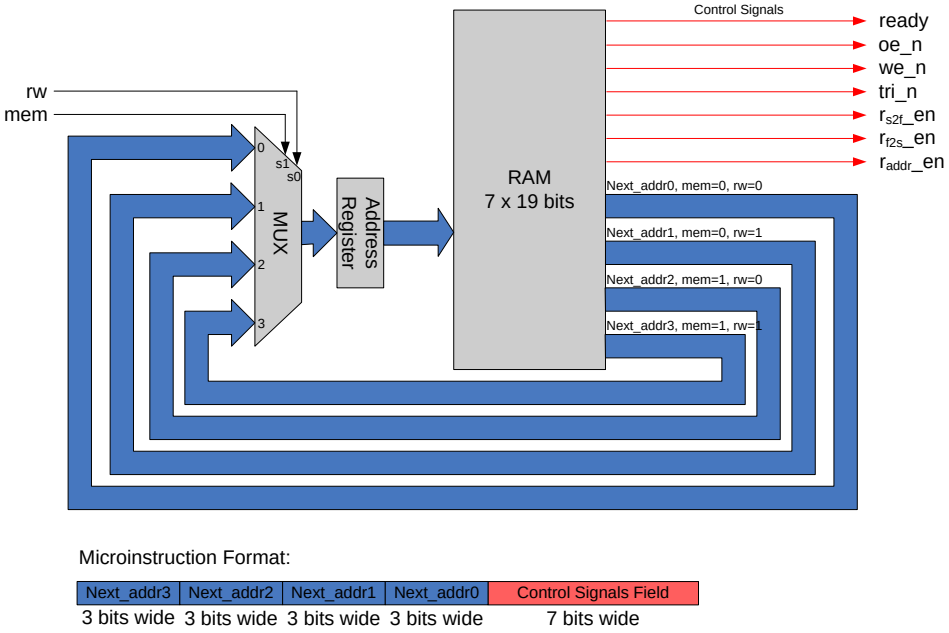
- Adresni registar obezbeđuje adresu za memoriju mikroinstrukcija. Adresirana mikroinstrukcija pojavljuje se na izlazima memorije mikroinstrukcija. Kao što je već rečeno, u opštem slučaju, svaka mikroinstruckija sastoji se iz tri polja: polja koje sadrži adrese narednih mikroinstrukcija koje slede tekuću, statusnog polja i kontrolnog polja. Sadržaj statusnog polja koristi se za prosleđivanje informacije o tekućem statusu sistema, dok se sadržaj kontrolnog polja koristi za upravljanje radom asociranog *datapath* modula. Adresno polje prosleđuje se logici za generisanje naredne adrese, kao bi se na osnovu njega generisala korektna adresa naredne mikroinstrukcije.
- Asocirani *datapath* modul se konfiguriše prema trenutnim vrednostima kontrolnog polja i generiše nove vrednosti ulaznih statusnih signala koje se takođe prosleđuju logici za generisanje naredne adrese.

- Logika za generisanje naredne adrese, na osnovu trenutnih vrednosti komandnog ulaza, statusnog ulaza i adresnog polja generiše vrednost adrese naredne mikroinstrukcije koju treba izvršiti. Ova vrednost se zatim upisuje u adresni registar, čime se i završavaju aktivnosti u okviru jedne periode klok signala.

Zadaci za vežbu

Zadatak 3.7.

Jedna moguća arhitektura mikroprogramirane upravljačke jedinice konačnog automata SRAM memorijskog kontrolera, čiji ASM dijagram je prikazan na slici 3.17 prikazana je na slici 3.19.



Slika 3.19. Arhitektura mikroprogramirane upravljačke jedinice SRAM memorijskog kontrolera

Kao što se sa slike može videti, osnovu mikroprogramirane upravljačke jedinice čini RAM/ROM memorija veličine 7x19 bita. Broj potrebnih memorijskih lokacija određen je brojem stanja u ASM dijagramu sa slike 3.17, a potreban broj bitova u svakoj od mikroinstrukcija određen je na sledeći način. Upravljačka jedinica treba da generiše ukupno 7 različitih upravljačkih signala (vidi sliku 3.16), a s'obzirom da iz *idle* stanja

možemo otići u tri različita naredna stanja (*idle*, *r1*, *w1*) potrebno je imati ukupno 4 adresna polja u kojima će biti smeštene adrese naredne mikroinstrukcije koju treba izvršiti. Kako imamo ukupno 7 različitih mikroinstrukcija, veličina svakog od adresnih polja mora biti najmanje 3 bita. Sadržaj RAM/ROM memorije prikazan je u tabeli 3.3.

Stanje	adr	n_addr3	n_addr2	n_addr1	n_addr0	r_{addr}_en	r_{f2s}_en	r_{s2f}_en	tri_n	we_n	oe_n	ready
idle	0	4	1	0	0	0	0	0	1	1	1	1
w1	1	2	2	2	2	1	1	0	1	1	1	0
w2	2	3	3	3	3	0	0	0	0	0	1	0
w3	3	0	0	0	0	0	0	0	0	1	1	0
r1	4	5	5	5	5	1	0	0	1	1	1	0
r2	5	6	6	6	6	0	0	0	1	1	0	0
r3	6	0	0	0	0	0	0	1	1	1	0	0

Tabela 3.3. Sadržaj RAM/ROM memorije mikroprogramirane upravljačke jedinice

a) Napisati VHDL model mikroprogramirane upravljačke jedinice čija je arhitektura prikazana na slici 3.19. Sadržaj RAM memorije treba da odgovara sadržaju tabele 3.3.

b) Napisati jednostavan testbenč koji će izvršiti verifikaciju upravljačke jedinice projektovane pod a).

Zadatak 3.8.

Projektovani memorijski kontroler koji nakon završetka tekuće memorijske operacije mora da se vrati u *idle* stanje u kojem čeka na zahtev za novom memorijskom operacijom. To znači da glavni sistem mora da sačeka još jednu periodu klok signala pre nego što inicira sledeću memorijsku operaciju. Ovakve uzastopne memorijske operacije u engleskoj literaturi su poznate pod imenom „*back-to-back memory access*“. Projektovani memorijski kontroler ima lošu osobinu da „*back-to-back memory access*“ traje 4 periode klok signala.

a) Projektovati novu upravljačku jedinicu memorijskog kontrolera koja će omogućiti skraćivanje „*back-to-back memory access*“ na 3 periode klok signala.

   Ideja: umesto da se nakon kompletiranja tekuće memorijske operacije kontroler vrati u *idle* stanje u stanjima *r3*, odnosno *w3*, izvršiti proveru da li je glavni sistem inicirao narednu memorijsku operaciju, i ako jeste odmah preći u stanje *r1*, odnosno *w1*, bez vraćanja u *idle* stanje. Vratiti se u *idle* stanje jedino ako glavni sistem nije inicirao novu memorijsku operaciju.

b) Nacrtati ASM dijagram nove upravljačke jedinice.

c) Na osnovu ASM dijagrama realizovati upravljačku jedinicu korišćenjem mikroprogramirane arhitekture.

d) Napisati jednostavan testbenč koji će izvršiti verifikaciju upravljačke jedinice projektovane pod c).