# Laboratorijska vežba 7

## Opis konačnih automata korišćenjem VHDL jezika

### Konačni automati

Konačni automati (*Finite State Machine, FSM*) pripadaju grupi sekvencijalnih mreža. Konačni automat kao model zapravo može da modeluje proizvoljnu sekvencijalnu mežu. Osnovna razlika između konačnih automata i ostalih vrsta sekvencijalnih mreža ogleda se u tome da konačni automati imaju neregularnu strukturu funkcije narednog stanja. Flip flopovi, registri, memorije i brojači su sekvencijalne mreže sa regularnom strukturom funkcije narednog stanja.

Formalno, konačni automat se definiše pomoću pet objekata:

- skupa simboličkih stanja u kojima se može naći automat
- skupa ulaznih signala na koje je automat osetljiv
- 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

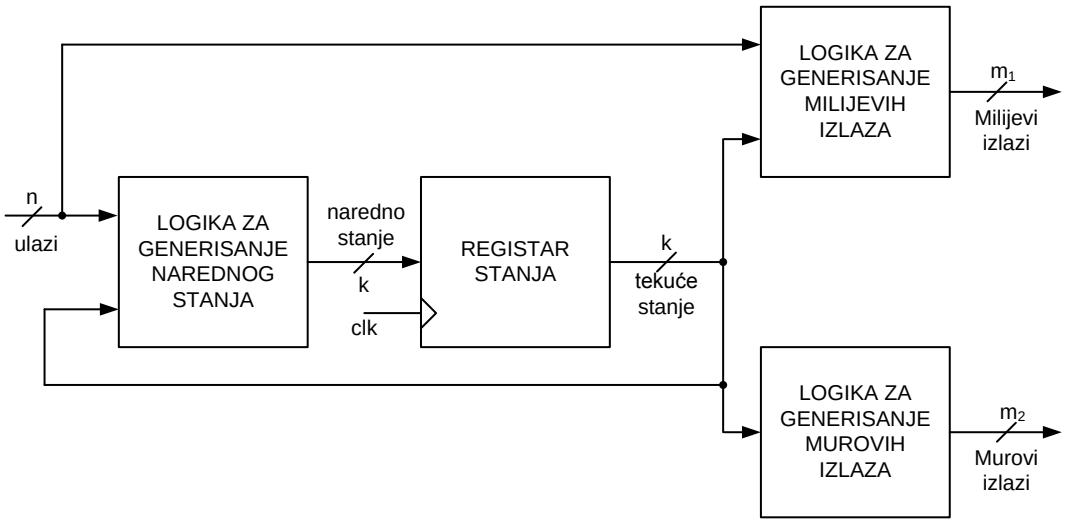
Stanje automata predstavlja jedinstveno unutrašnje stanje u kome se automat može naći. Skup svih dozvoljenih stanja konačnog automata more biti konačan (otuda i naziv *konačni* automat). Kako protiče vreme, konačni automat prelazi iz jednog stanja u drugo. Naredno stanje automata određeno je funkcijom 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, na ovim vežbama ćemo se baviti isključivo sinhronim konačnim automatima.

Izlazna funkcija automata koristi se za određivanje tekućih vrednosti izlaznih signala automata. U zavisnosti od šta predstavlja 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 konačnog automata prikazan je na slici 1.




Slika 1. Generalna struktura konačnog automata

Konačni automat u opštem slučaju poseduje sledeće portove:

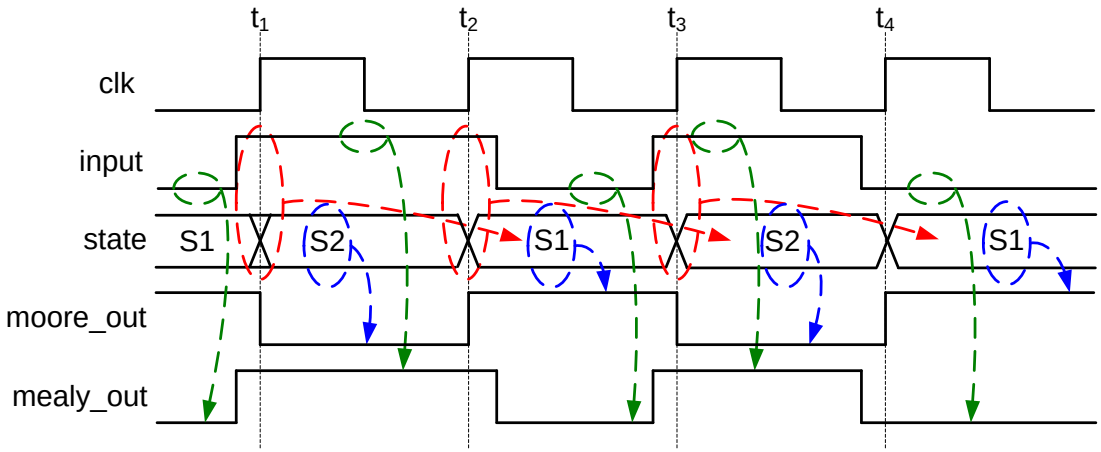
- *N* ulaznih portova,
- $M_1$ izlaznih portova Milijevog tipa,
- $M_2$ izlaznih portova Murovog tipa,
- *clk* ulaz za sinhronizaciju rada konačnog automata

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 ili 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 jednim Murovim i Milijevim izlazom, *moore_out* i *mealy_out*, prikazan je na slici 2. U ovom primeru ulazni port *input* je jednobitni, kao i Murovi i Milijevi izlazni portovi *moore_out* i *mealy_out*.




Slika 2. Vremenski dijagram rada konačnog automata, sa jednim ulazom i dva izlaza

Sa slike 2 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 2 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 2 svaki put kada se nađe u stanju *S2* spušta *moore_out* izlaz na 0. Karakteristične promene do kojih dolazi prilikom rada automata na slici 2 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 promena na Milijevom izlazu.

Sa slike 2 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, 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 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 tri modela se najčešće koriste u praksi:

- tekstualni opis

- opis pomoću tablice prelaza/izlaza
- opis pomoću dijagrama stanja

*Tekstualni opis*

Tekstualni opis kao model za specifikaciju željenog ponašanja konačnog automata često se koristi u praksi iako ima ozbiljne nedostatke. Glavni nedostatak ovog pristupa ogleda se u nedovoljnoj preciznosti specifikacije i mogućim različitim interpretacijama. Ovo je posledica prevelike slobode izražavanja koju nam pružaju prirodni jezici (srpski, engleski). Jedna te ista rečenica može se protumačiti na različite načine od strane različitih ljudi. Ovo postaje još veći problem ako je tekstulna specifikacija napisana na jeziku koji nije maternji za dizajnera koji treba da je pretoči u odgovarajuću implementaciju. Na primer, tekstualni opis željenog ponašanja kontrolera za lift mogao bi biti:

„Lift opslužuje dva sprata. Na svakom spratu ima dugme za poziv, a u kabini postoje dva dugmeta za odabiranje željenog sprata. Opisati u VHDL-u konačni automat koji opisuje lift imajući u vidu da:

a) Pritisak na svako dugme predstavlja poseban ulazni signal, slučaj istovremenog pritiska na više od jednog tastera nije moguć i paziti da postoji i slučaj kada ni jedan ulaz nije aktivan (kodovati sa tri bita)

b) Izlazne signali predstavljaju rad ili mirovanje motora

c) Stanje automata predstavlja mirovanje lifta na nekom spratu.”

Potencijalne nejasnoće u ovoj tekstualnoj specifikaciji mogle bi biti:

1. Koliko različitih stanja mirovanja lifta postoji? U tekstu kaže da lift opslužuje dva sprata. Da li to uključuje i prizemlje ili u zgradi ima dva sprata plus prizemlje? U prvom slučaju postoje dva stanja mirovanja (prizemlje + jedan dodatni sprat), u drugom postoje tri stanja mirovanja (prizemlje + dva sprata).
2. U delu pod a) napominje se da je potrebno koristiti kodovanje sa tri bita. Na šta se ovo tačno odnosi? Ako pretpostavimo da se lift može naći u dva stanja mirovanja, onda ukupno imamo četiri ulazna signala (po jedan taster za poziv lifta na prizemlju i prvom spratu + dva tastera unutar lifta). Ako pretpostavimo da se lift može naći u tri stanja mirovanja, onda imamo ukupno pet ulaznih signala u automat. Ni u jednom slučaju nemamo samo tri ulaza. Na šta se onda odnosi napomena da koristimo kodovanje sa tri bita?
3. Da li je reč o inteligentnom liftu, koji je u stanju da reaguje na nove pozive dok je u pokretu? Da li opaska da „slučaj istovremenog pritiska na više od jednog tastera nije moguć“ znači da će dok je lift u pokretu a pritisne se taster za poziv na nekom drugom spratu, ovaj novi poziv biti ignorisan? Da li lift reaguje na nove pozive samo dok je u stanju mirovanja?

Kao što možemo primetiti tekstualni opis željenog rada konačnog automata je prilično nerazumljiv i podložan različitim interpretacijama.

*Opis pomoću tablice prelaza/izlaza*

Opis pomoću tablice prelaza/izlaza je mnogo formalniji način specifikacije željenog ponašanja konačnog automata koji ostavlja daleko manje prostora za različite interpretacije od strane dizajnera. Ovaj opis obično se sastoji iz jedne tabele u kojoj se

za svako od mogućih stanja u kojima se automat može naći jasno navode svi dozvoljeni prelazi u naredno stanje, kao i potrebne vrednosti izlaznih signala u tom trenutku. Jedan primer tablične specifikacije rada konačnog automata prikazan je u nastavku.

| Trenutno stanje | Naredno stanje/izlaz | | | |
|---|---|---|---|---|
| | Ulazi: (x, y) | | | |
| | (0, 0) | (0, 1) | (1, 0) | (1, 1) |
| *S1* | *S3*/0 | *S3*/0 | *S2*/1 | *S2*/1 |
| *S2* | *S1*/0 | *S1*/0 | *S1*/0 | *S1*/0 |
| *S3* | *S4*/1 | *S4*/1 | *S4*/1 | *S4*/1 |
| *S4* | *S1*/0 | *S2*/0 | *S1*/0 | *S2*/0 |

Na osnovu gornje tabele možemo zaključiti sledeće:

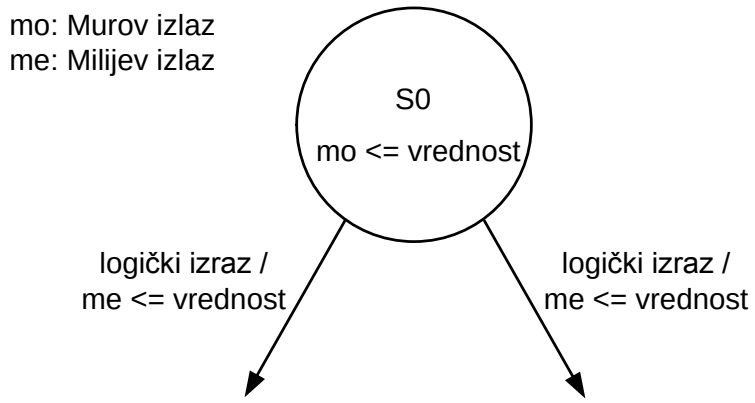
- Konačni automat koji je potrebno projektovati ima ukupno četiri stanja u kojima se može naći (*S1, S2, S3, S4*).
- Konačni automat ima dva jednobitna ulaza, *x* i *y*.
- Konačni automat ima jedan jednobitni Milijev izlaz. Znamo da je reč o Milijevom izlazu jer iz tablice možemo videti da se vrednost izlaza menja ne samo od stanja u kome se automat nalazi već i od trenutnih vrednosti ulaza *x* i *y*. Na primer, kada je automat u stanju *S1*, izlaz treba da ima vrednost 0 kada ulazi *x* i *y* imaju vrednosti "00" ili "01", a treba da ima vrednost 1 kada ulazi *x* i *y* imaje vrednosti "10" ili "11". Daljom analizom možemo zaključiti da Milijev izlaz konačnog automata treba da zapravo prati stanje na ulazu *x*, dok se automat nalazi u stanju *S1*.

Iako je tablični način specifikacije daleko bolji od tekstualnog, ipak nije dovoljno razumljiv i teško se na osnovu njega može „shvatiti logika rada" konačnog automata koji je potrebno projektovati. Takođe, bilo kakve naknadne modifikacije automata moraju se izvršiti vrlo pažljivo jer je vrlo lako u tablici modifikovati pogrešno polje ili zaboraviti da je potrebno modifikovati još neko dodatno polje kako bi se promenio način rada automata u skladu sa našim potrebama.

*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 obzirom da je reč o grafičkom prikazu, opis je daleko razumljivi za dizajnera od prethodna dva tipa.

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.




Slika 3. Generička struktura čvora unutar dijagrama stanja konačnog automata

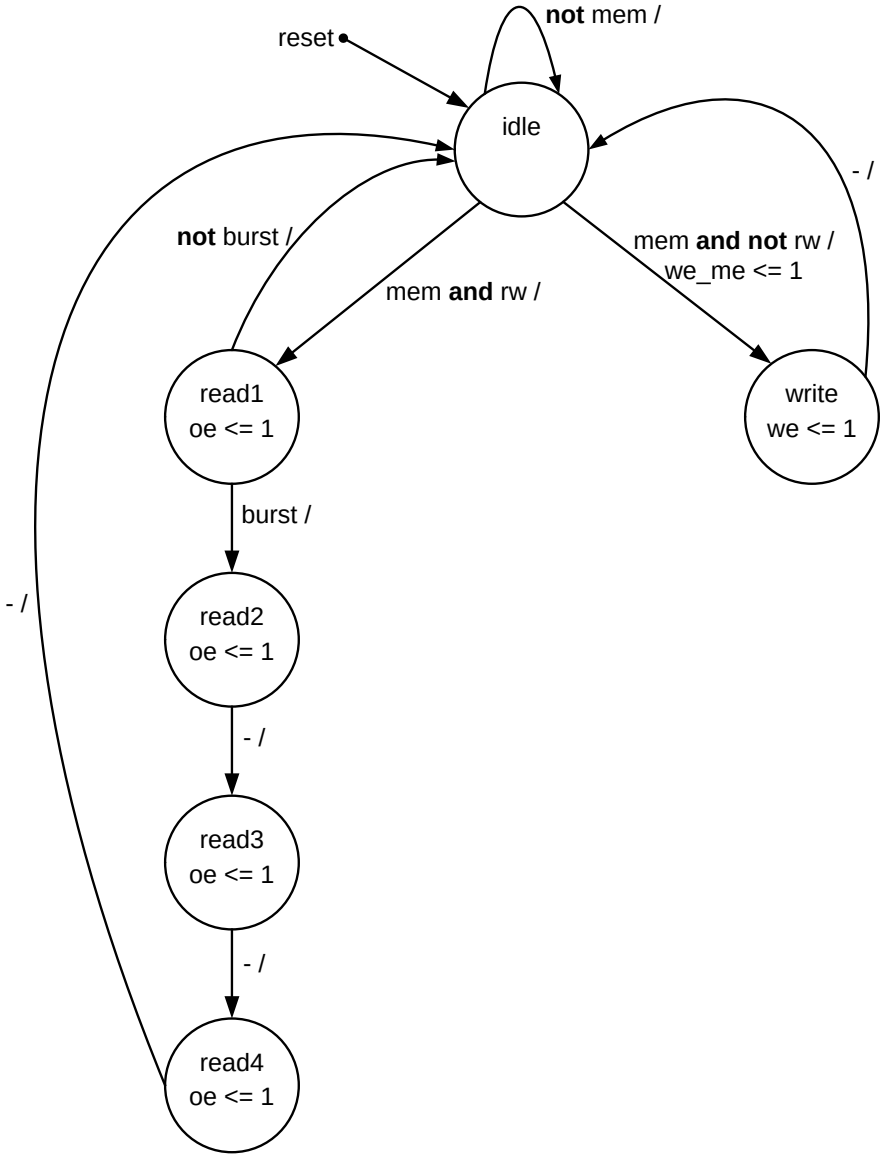
Svaki čvor u dijagramu stanja predstavlja jedno, jedinstveno, stanje u kome se može naći konačni automat. Svaki čvor mora imati jedinstveno simboličko ime koje mu je pridruženo. Čvor na slici 3 ima pridruženo simboličko ime *S0*, koje ujedno predstavlja i identifikaciju stanja u kome se konačni automat 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 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), obzirom da su oni samo funkcija tekućeg stanja u kome se automat nalazi. 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 smeštaju se pored svake od izlaznih grana (*me <= vrednost* na slici 3).

Da bi se pojednostavio dijagram stanja, konvencija je da se na njemu navode dodele izlaznim portovima koji u datom stanju treba da se aktiviraju. 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 4 prikazan je dijagram stanja konačnog automata hipotetičkog memorijskog kontrolera.




Slika 4. 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 signali su jednobitni i generiše ih procesor. *Mem* signal se aktivira (dovodi na vrednost 1) svaki put kada procesor želi da pristupi memoriji. *Rw* signal 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* signal služi za iniciranje specijalnog moda čitanja podataka iz memorije. Kada je *burst* signal 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 postvljen na 1 memorijski kontroler se vraća u *idle* stanje.

Sam rad memorijskog kontrolera sa slike 4 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 4:

- 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.

**VHDL modeli konačnih automata**

Konačni automati se u VHDL-u modeluju korišćenjem odgovarajućih *process* naredbi. Obzirom da opšta struktura konačnog automata, prikazana na slici 1, sadrži nekoliko kombinacionih mreža (za generisanje narednog stanja i vrednosti Murovih i Milijevih izlaza) i jednu sekvencijalnu mrežu (registar stanja) postoje tri različita načina modelovanja konačnih automata pomoću VHDL jezika:

- pomoću četiri process naredbe
- pomoću dve process naredbe
- pomoću jedne process naredbe

Pored ovog izbora, dizajner se mora odlučiti a na koji način će izvršiti reprezentaciju konačnog broja stanja u kojima se automat može naći. U ovom slučaju postoje dve mogućnosti:

- Modelovanje stanja automata korišćenjem nabrojivog tipa podatka
- Modelovanje stanja automata korišćenjem vektorskog tipa podataka

Da bi smo ilustrovali različite načine modelovanja koristićemo konačni automat definisan dijagramom stanja sa slike 4. Za ovaj automat odgovarajuća e*ntity* deklaracija ima sledeći izgled

```
library ieee;
use ieee.std_logic_1164.all;

entity mem_ctrl is
   port (clk:      in std_logic;
         reset:    in std_logic;

         -- Ulazni portovi
         mem:      in std_logic;
         rw:       in std_logic;
         burst:    in std_logic;

         -- Izlazni portovi
         oe:       out std_logic;
         we:       out std_logic;
         we_me:    out std_logic
        );
end entity mem_ctrl;
```

Nakon što smo opisali interfejs memorijskog kontrolera, potrebno je modelovati njegovu funkcionalnost. Za ovo je potrebno napisati odgovarajuće arhitekturno telo.

***Bihevijani model***

***Varijanta 1***: Model memorijskog kontrolera korišćenjem četiri *process* naredbe i nabrojivog tipa podataka za modelovanje stanja u kome se kontroler nalazi

```
architecture mult_seg_arch of mem_ctrl is
   type mc_state_type is (idle, read1, read2, read3, read4, write);
   signal state_reg, state_next: mc_state_type;
begin
   -- registar stanja
   process (clk, reset) is
   begin
      if (reset = '1') then
         state_reg <= idle;
      elsif (clk'event and clk = '1') then
         state_reg <= state_next;
      end if;
   end process;

   -- logika za generisanje narednog stanja
   process (state_reg, mem, rw, burst)
   begin
      case state_reg is
         when idle =>
            if mem = '1' then
               if rw = '1' then
                  state_next <= read1;
               else
                  state_next <= write;
               end if;
            else
               state_next <= idle;
            endif;
         when write =>
            state_next <= idle;
         when read1 =>
            if (burst = '1') then
               state_next <= read2;
            else
               state_next <= idle;
            end if;
         when read2 =>
            state_next <= read3;
         when read3 =>
            state_next <= read4;
         when read4 =>
            state_next <= idle;
      end case;
   end process;

    -- Murovi izlazi
   process (state_reg)
   begin
```

```
        we <= '0'; -- prodrazumevana vrednost
        oe <= '0'; -- podrazumevana vrednost
        case state_reg is
          when idle =>
          when write =>
            we <= '1';
          when read1 =>
            oe <= '1';
          when read2 =>
            oe <= '1';
          when read3 =>
            oe <= '1';
          when read4 =>
            oe <= '1';
        end case;
      end process;

      -- Milijev izlaz
      process (state_reg, mem, rw)
      begin
        we_me <= '0'; -- podrazumevana vrednost
        case state_reg is
          when idle =>
            if (mem = '1') and (rw = '0') then
              we_me <= '1';
            end if;
          when others =>
        end case;
      end process;
    end mult_seg_arch;
```

U prikazanom modelu konačnog automata imamo ukupno četiri konkurentna procesa. Prvi proces je sekvencijalni (jer je osetljiv na *clk* signal) i modeluje registar stanja konačnog automata. Kao što se može videti analizom tela procesa reč je o standardnom modelu registra sa sinhronim reset ulazom.

Drugi proces modeluje kombinacionu mrežu koja je zadužena za generisanje narednog stanja konačnog automata. Ovaj proces je osetljiv na trenutno stanje u kome se automat nalazi (*state_reg* signal), kao i na ulazne portove konačnog automata (*mem*, *rw* i *burst*). U okviru tela procesa nalazi se jedna *case* naredba koja prolazi kroz sva dozvoljena stanja konačnog automata. U svakom od stanja, pomoću dodatnih *if* naredbi, vrši se izračunavanje narednog stanja u kome automat treba da se nađe, prateći specifikaciju automata zadatu dijagramom stanja sa slike 4. Na primer, ako se automat nalazi u *idle* stanju tada se u odgovarajućoj *when* grani *case* naredbe koja je asocirana ovom stanju prvo vrši provera *mem* ulaznog porta. Ukoliko je mem jednak 1, na osnovu rezulata testa *rw* porta naredno stanje automata (signal *state_next*) postaje ili *read1* (kada je *rw* port jednak 1) ili *write* (kada je *rw* port jednak 0). U slučaju da je mem port jednak 0, automat ostaje u *idle* stanju (*state_next* signalu se dodeljuje vrednost idle).

Treći proces takođe modeluje kombinacinu mrežu, ali ovaj put je to mreža koja je zadužena za generisanje Murovih izlaza. Imajući ovo u vidu, proces je osetljiv samo na tekuće stanje konačnog automata (*state_reg* signal). U zavisnosti od tekućeg stanja

automata, korišćenjem jedne *case* naredbe, vrši se dodela odgovarajućih vrednosti Murovim izlazima *oe* i *we*.

Konačno, četvrti proces modeluje kombinacionu mrežu koja generiše Milijeve izlaze. Ovaj proces osetljiv je na tekuće stanje automata (*state_reg* signal) kao i na odgovarajuće ulazne portove automata (u našem prmeru su to portovi *mem* i *rw*) koji su od značaja za generisanje Milijevih izlaza. U zavisnosti od tekućeg stanja u kome se automat nalazi, kao i od tekućih vrednosti ulaznih portova, proces generiše potrebne vrednosti na Milijevom izlazu *we_me*.

Prikazani način modelovanja rezultuje u najdužem modelu od svih ostalih modela koji će biti prikazani. Međutim, ovaj način modelovanja i pruža najveći stepen kontrole dizajneru, što ponekad može biti od interesa.

**NAPOMENA**: U ovom primeru modelovanja konačnog automata pomoću VHDL-a, za reprezentaciju tekućeg i narednog stanja konačnog automata korišćen je nabrojivi tip podataka, *mc_state_type*. Korišćenjem nabrojivog tipa podataka za reprezentaciju stanja konačnog automata izbegnuta je potreba za eksplicitnim kodovanjem skupa stanja u kojima se automat može naći odgovarajućim binarnim rečima. Postupak optimalnog izbora načina kodovanja simboličkih stanja binarnim rečima u opštem slučaju je algoritamski težak i nepraktično ga je rešavati ručno. Sa druge strane, izborom neoptimalnog načina kodovanja stanja automata prilikom njegove hardverske implementacije rezultujući elektronski sistem takođe neće biti optimalan, u smislu korišćenja minimalnog broja hardverskih resursa (flip flopova i logičkih kapija) i maksimalne brzine rada (maksimalne učestanosti *clk* signala). Savremeni alati za automatsku sintezu hardvera sadrže implementacije različitih algoritama za optimalan izbor načina kodovanja stanja automata. Ukoliko se prilikom modelovanja konačnog automata koristi nabrojivi tip za reprezentaciju stanja automata, nalaženje najboljeg načina kodovanja prepušta se alatu za automatsku sintezu hardvera. U većini slučajeva ovo je preporučeni način modelovanja konačnog automata. Slučaju kada to nije poželjno biće diskutovan prilikom predstavljanja četvrte varijante modelovanja konačnih automata.

***Varijanta 2***: Model memorijskog kontrolera korišćenjem dve *process* naredbe i nabrojivog tipa podataka za modelovanje stanja u kome se kontroler nalazi

```
architecture two_seg_arch of mem_ctrl is
   type mc_state_type is (idle, read1, read2, read3, read4, write);
   signal state_reg, state_next: mc_state_type;
begin
   -- registar stanja
   process (clk, reset) is
   begin
      if (reset = '1') then
         state_reg <= idle;
      elsif (clk'event and clk = '1') then
         state_reg <= state_next;
      end if;
   end process;

   -- logika za generisanje narednog stanja i izlazna logika
   process (state_reg, mem, rw, burst)
   begin
```

```
        oe <= '0'; -- podrazumevane vrednosti
        we <= '0';
        we_me <= '0';
        case state_reg is
          when idle =>
            if mem = '1' then
              if rw = '1' then
                state_next <= read1;
              else
                state_next <= write;
                we_me <= '1';
              end if;
            else
              state_next <= idle;
            end if;
          when write =>
            state_next <= idle;
            we <= '1';
          when read1 =>
            if (burst = 'l') then
              state_next <= read2;
            else
              state_next <= idle;
            end if;
            oe <= '1';
          when read2 =>
            state_next <= read3;
            oe <= '1';
          when read3 =>
            state_next <= read4;
            oe <= '1';
          when read4 =>
            state_next <= idle;
            oe <= '1';
        end case;
      end process;
    end two_seg_arch;
```

**NAPOMENA**: U ovoj varijanti sve kombinacione mreže koje mogu biti prisutne unutar jednog konačnog automata modeluju se pomoću jednog, zajedničkog procesa. Drugi proces služi za modelovanje registra stanja, kao i ranije. Ovaj način modelovanja se najčešće koristi u praksi jer ga odlikuje dobar kompromis između dužine modela i mogućnosti kontrole.

***Varijanta 3***: Model memorijskog kontrolera korišćenjem jedne *process* naredbe i nabrojivog tipa podataka za modelovanje stanja u kome se kontroler nalazi

```
architecture one_seg_arch of mem_ctrl is
  type mc_state_type is (idle, read1, read2, read3, read4, write);
  signal state_reg: mc_state_type;
```

```
begin
   process (clk, reset)
   begin
      if (reset = '1') then
         state_reg <= idle;
      elsif (clk'event and clk = '1') then
         oe <= '0'; -- podrazumevane vrednosti
         we <= '0';
         we_me <= '0';
         case state_reg is
            when idle =>
               if mem = '1' then
                  if rw = '1' then
                     state_reg <= read1;
                  else
                     state_reg <= write;
                     we_me <= '1';
                  end if;
               else
                  state_reg <= idle;
               end if;
            when write =>
               state_reg <= idle;
               we <= '1';
            when read1 =>
               if (burst = '1') then
                  state_reg <= read2;
               else
                  state_reg <= idle;
               end if;
               oe <= '1';
            when read2 =>
               state_reg <= read3;
               oe <= '1';
            when read3 =>
               state_reg <= read4;
               oe <= '1';
            when read4 =>
               state_reg <= idle;
               oe <= '1';
         end case;
      end if;
   end process;
end one_seg_arch;
```

**NAPOMENA**: Ovaj način modelovanja rada konačnih automata rezultuje u najkompaktijem modelu, te je zbog toga jako privlačan. Međutim, korišćenje ovog načina modelovanj nosi sa sobom jednu opasnost. Obzirom da su svi moduli sa slike 1 sada modelovani pomoću jednog, zajedničkog, procesa ovaj proces mora da napisan na takav način da modeluje sekvencijalnu mrežu (zbog toga što unutar njega moramo modelovati i registar stanja, koji je sekvencijalna mreža). Zbog toga će i sve ostale mreže (za računanje narednog stanja, Murovih i Milijevih izlaza) takođe biti

modelovane kao sekvencijalne mreže, što one zapravo nisu i ne treba da budu! Modelovanje kombinacionih mreža kao da su sekvencijalne rezultovaće u implicitnom ubacivanju flip flopova na svaki od izlaznih portova automata i kašnjenju izlaznih signala u trajanju od jedne periode *clk* signala u odnosu na željeno ponašanje! Ovo nužno ne mora da bude loše rešenje, ukoliko se prilikom projektovanja konačnog automata uzme u obzir (na primer jedno rešenje bi bilo da se svi izlazni signali generišu jedan takt ranije kako bi se kompenzovalo ovo dodatno kašnjenje). Zbog svega rečenog, ovaj način modelovanja konačnih automata nije preporučljiv, naročito početnicima.

***Varijanta 4***: Model memorijskog kontrolera korišćenjem dve *process* naredbe i vektorskog tipa podataka za modelovanje stanja u kome se kontroler nalazi

```
architecture two_seg_arch_ver2 of mem_ctrl is
   constant idle:    std_logic_vector(2 downto 0) := “000”;
   constant read1: std_logic_vector(2 downto 0) := “001”;
   constant read2: std_logic_vector(2 downto 0) := “010”;
   constant read3: std_logic_vector(2 downto 0) := “011”;
   constant read4: std_logic_vector(2 downto 0) := “100”;
   constant write:  std_logic_vector(2 downto 0) := “101”;

   signal state_reg, state_next: std_logic_vector(2 downto 0);
begin
   -- registar stanja
   process (clk, reset) is
   begin
      if (reset = ‘1’) then
         state_reg <= idle;
      elsif (clk’event and clk = ‘1’) then
         state_reg <= state_next;
      end if;
   end process;

   -- logika za generisanje narednog stanja i izlazna logika
   process (state_reg, mem, rw, burst)
   begin
      oe <= '0'; -- podrazumevane vrednosti
      we <= '0';
      we_me <= '0';
      case state_reg is
         when idle =>
            if mem = ‘1’ then
               if rw = ‘1’ then
                  state_next <= read1;
               else
                  state_next <= write;
                  we_me <= '1';
               end if;
            else
               state_next <= idle;
            end if;
```

```
            when write =>
               state_next <= idle;
               we <= '1';
            when read1 =>
               if (burst = '1') then
                  state_next <= read2;
               else
                  state_next <= idle;
               end if;
               oe <= '1';
            when read2 =>
               state_next <= read3;
               oe <= '1';
            when read3 =>
               state_next <= read4;
               oe <= '1';
            when read4 =>
               state_next <= idle;
               oe <= '1';
            when others =>
         end case;
      end process;
   end two_seg_arch_ver2;
```

**NAPOMENA**: Prikazani model razlikuje se od modela iz varijante 2 samo u jednom detalju. Umesto nabrojivog tipa podataka (*mc_state_type* tip) za reprezentaciju stanja konačnog automata ovde je iskorišćen vektorski tip (*std_logic_vector* tip). Korišćenjem vektorskog tipa, dizajner ima mogućnost potpune kontrole nad procesom hardverske implementacije konačnog automata, jer je sada moguće eksplicitno specificirati način kodovanja svakog od mogućih stanja u kome se konačni automat može naći. Ovo je u prikazanom modelu postignuto definisanjem odgovarajućeg broja konstanti unutar deklarativnog dela arhitekture. Prilikom deklaracije svake od konstanti, navedena je i njega vrednost, u terminima binarnog vektora. Prilikom automatske sinteze hardvera pomoću kojega će biti implementiran ovaj model konačnog automata, alat će biti „primoran“ da koristi specificiran način kodovanja. Iako ovo u većini slučajeva nije dobra ideja, jer po pravilu rezultuje u neoptimalnoj implementaciji, postoji jedan izuzetak. Prilikom hardverskog testiranja rada projektovanog sistema, u procesu otklanjanja grešaka potrebno je znati način kako su kodirana stanja automata. Ukoliko se način kodovanja automatski određuje unutar alata za sintezu hardvera, onda on može biti nepoznat dizajneru, što će u velikoj meri otežati proces korelisanja observiranog stanja (prikazanog u formi binarnog vektora) i simboličke oznake na dijagramu stanja. Ukoliko pak dizajner eksplicitno navede način pridruživanja kodnih reči svakoj simboličkoj oznaci stanja, ovaj problem će biti izbegnut. U praksi se često pribegava ovom načinu modelovanja u ranoj fazi razvoja, kada se vrši testiranje ispravnog načina rada projektovanog sistema. Nakon što se utvrdi da sistem ispravno funkcioniše, model konačnog automata se prerađuje tako da koristi nabrojivi tip podataka, kako bi se generisao najoptimalniji hardver u finalnom uređaju.

**Verifikaciono okruženje**

Primer jednog mogućeg testbenča koji se može iskoristiti za verifikaciju memorijskog kontrolera prikazan je u nastavku.

```
library ieee;
use ieee.std_logic_1164.all;

entity mem_ctrl_tb is
end entity mem_ctrl_tb;


architecture beh of mem_ctrl_tb is
   -- Deklaracija unutrašnjih signala potrebnih za povezivanje stimulus
   -- generatora sa ulazima DUV-a
   signal clk_s: std_logic;
   signal reset_s: std_logic;
   signal mem_s: std_logic;
   signal rw_s: std_logic;
   signal burst_s: std_logic;
   signal oe_s: std_logic;
   signal we_s: std_logic;
   signal we_me_s: std_logic;

begin
        -- Komponenta koja se verifikuje
        duv: entity work. mem_ctrl(two_seg_arch)
           port map (
              clk       =>  clk_s,
              reset     =>  reset_s,
              mem       =>  mem_s,
              rw        =>  rw_s,
              burst     =>  burst_s,
              oe        =>  oe_s,
              we        =>  we_s,
              we_me =>  we_me_s);

        -- Klok generator koji generise periodicni clk_s signal koji
        -- ce se koristiti za aktiviranje memorijskog kontrolera
        clk_gen: process
        begin
                clk_s <= '0', '1' after 100 ns;
                wait for 200 ns;
        end process;

        -- Stimulus generator koji generise potrebne vrednosti na
        -- ulaznim portovima DUV-a na osnovu kojih ce biti moguce
        -- proveriti da li DUV implementira potrebnu funkcionalnost
        stim_gen: process
        begin
           -- Inicijalizacija ulaznih signala memorijskog kontrolera
           mem_s <= '0';
           rw_s <= '0';
```

```
    burst_s <= '0';

    -- Inicijalizacija memorijskog kontrolera
    reset_s <= '1';
    wait until falling_edge(clk_s);
    reset_s <= '0';

    -- Nema nikakve komande, kontroler treba da ostane u idle
    -- stanju
    for i in 1 to 2 loop
       wait until falling_edge(clk_s);
    end loop;

    -- Zadavanje komande upisa sadrzaja u memoriju, kontroler
    -- bi trebalo da ode u write stanje
    mem_s <= '1';
    wait until falling_edge(clk_s);
    mem_s <= '0';
    wait until falling_edge(clk_s);

    -- Zadavanje komande citanja sadrzaja iz memorije,
    -- kontroler bi trebalo da ode u read1 stanje
    -- Posto ce burst_s signal ostati '0', kontroler treba da izvrsi
    -- samo jedan ciklus citanja i da se nakon toga vrati u
    -- idle stanje
    mem_s <= '1';
    rw_s <= '1';
    wait until falling_edge(clk_s);
    mem_s <= '0';
    rw_s <= '0';
    wait until falling_edge(clk_s);

    -- Zadavanje "burst" komande citanja sadrzaja iz memorije,
    -- kontroler bi trebalo da ode u read1 stanje
    -- Posto je trazen "burst" rezim citanja, kontroler bi trebalo
    -- da generise cetiri ciklusa citanja i tek onda se vrati u
    -- idle stanje
    mem_s <= '1';
    rw_s <= '1';
    burst_s <= '1';
    wait until falling_edge(clk_s);
    mem_s <= '0';
    rw_s <= '0';
    burst_s <= '1';
    wait until falling_edge(clk_s);
    burst_s <= '0';
    for i in 1 to 3 loop
       wait until falling_edge(clk_s);
    end loop;

    -- Kraj testiranja
    wait;
end process;
```

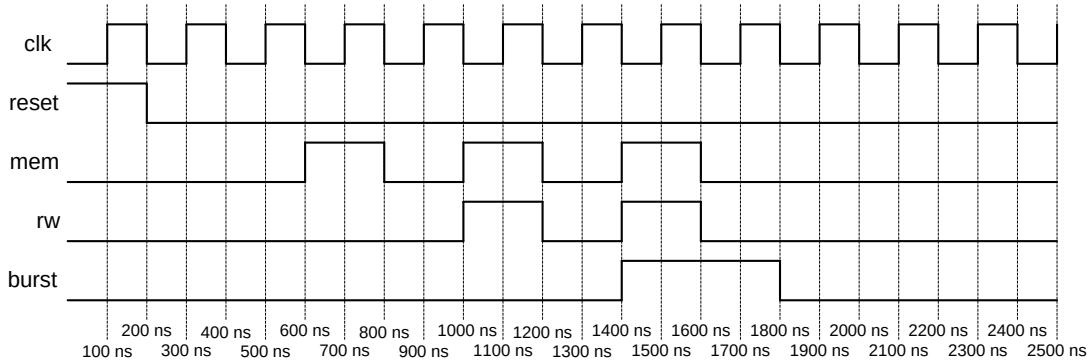
**end architecture** beh;

Prikazani testbenč instancionira komponentu memorijskog kontrolera i koristeći *clk_gen* i *stim_gen* procese generiše povorku ulaznih signala koja može da se iskoristi za proveru ispravnog rada napisanog modela kontrolera.

*Clk_gen* proces generiše periodičnu povorku impulsa na signalu *clk_s* koji se dovodi na *clk* ulaz memorijskog kontrolera.

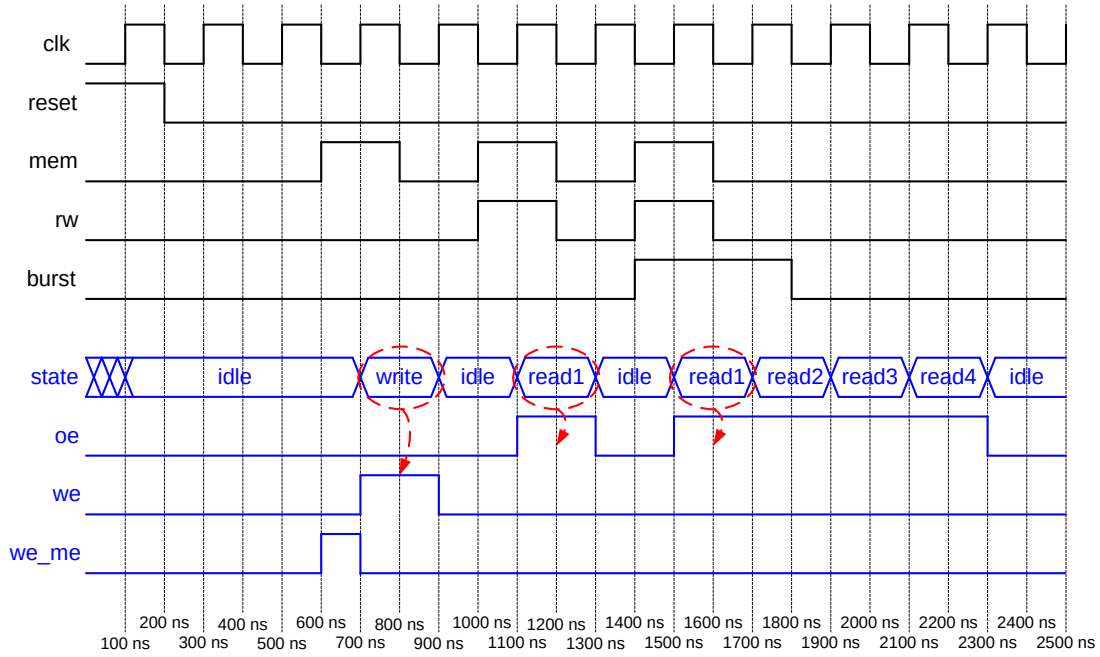
*Stim_gen* proces generiše vremenske oblike za tri ulazna signala memorijskog kontrolera (*mem_s, rw_s* i *burst_s* signali). Talasni oblici koji će biti generisani odabrani su tako da izvrše verifikaciju ispravnog rada memorijskog kontrolera. Na početku se aktivira reset signal kako bi se memorijski kontroler doveo u poznato početno stanje. Nakon toga stimulus generator čeka nekoliko taktova kako bi se proverilo da li će memorijski kontroler ostati u *idle* stanju. Zatim sledi zadavanje komande za upis podataka u memoriju. U ovom slučaju memorijski kontroler bi trebao da pređe u *write* stanje tokom kojega bi trebalo da se generišu upravljački signali memorije koji će aktivirati proces upisa u memoriju. Ovaj proces trebalo bi da traje jednu periodu klok signala. U sledećoj fazi verifikacije stimulus generator zadaje komandu čitanja jednog podatka iz memorije. Na kraju, generator zadaje komandu čitanja podataka u „burst“ režimu. U ovom slučaju memorijski kontroler trebalo bi da tokom četiri periode *clk* signala generiše zahtev za čitanjem.

Talasni oblici koji će biti generisani pomoću ove dve *process* naredbe prikazani su na slici 5.



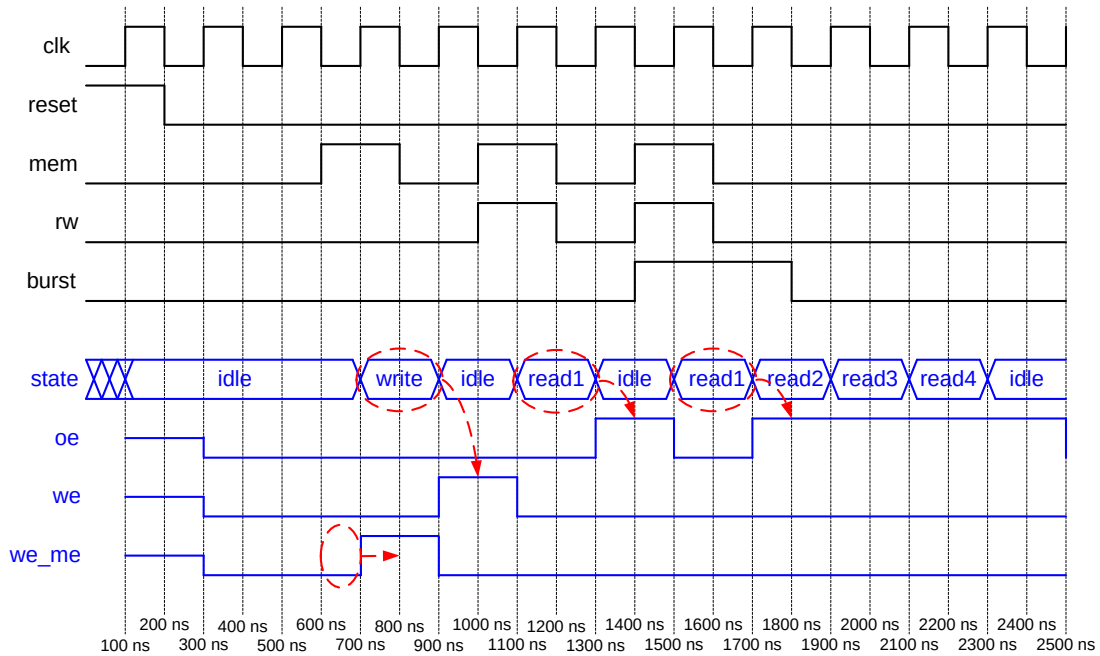

Slika 5. Generisani talasni oblici na ulaznaim signalima memorijskog kontrolera

Nakon što se izvrši simulacija kombinovanog rada testbenča i modela memorijskog kontrolera talasni oblici izlaznih signala kontrolera (*oe_s, we_s* i *we_me_s*), kao i unutrašnjeg stanja u kome se nalazi kontroler (*state* signal) trebalo bi da izgledaju kao na slici 6.




Slika 6. Rezultat simulacije modela memorijskog kontrolera baziranog na korišćenju dve *process* naredbe i nabrojivog tipa podataka za reprezentacije stanja automata

U slučaju da smo prilikom verifikacije koristili *one_seg_arch* arhitekturu, koja rad memorijskog kontrolera modeluje korišćenjem samo jedne *process* naredbe, vremenski oblici izlaznih signala, kao i unutrašnjeg stanja memorijskog kontrolera trebalo bi da imaju izgled prikazan na slici 7.

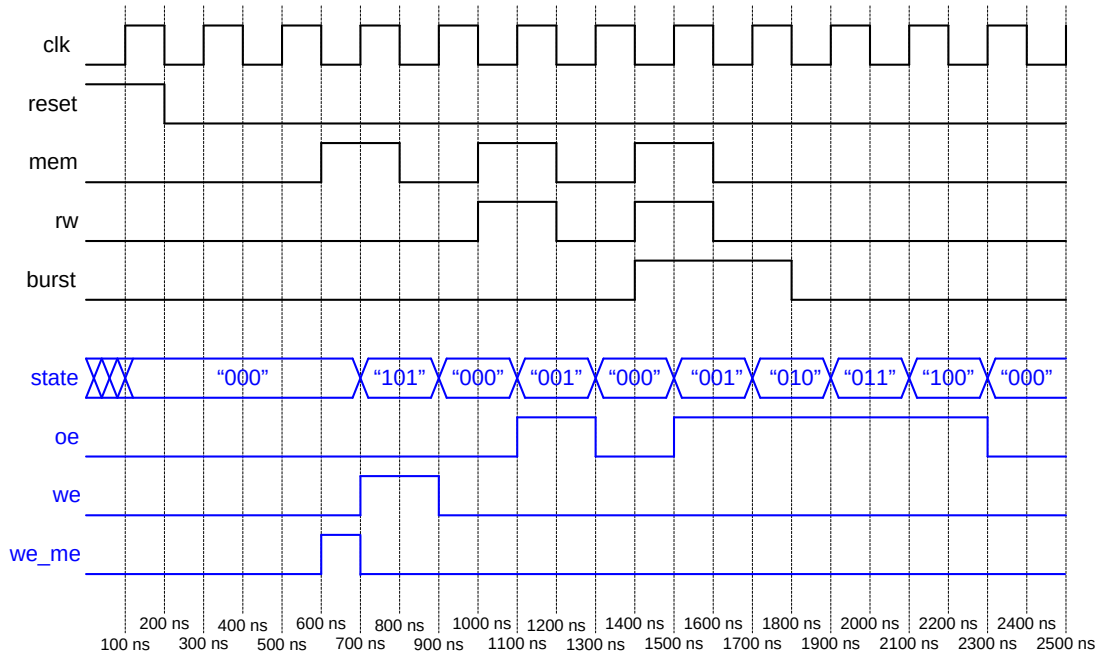



Slika 7. Rezultat simulacije modela memorijskog kontrolera baziranog na korišćenju jedne *process* naredbe i nabrojivog tipa podataka za reprezentacije stanja automata

Upoređivanjem talasnih oblika izlaznih signala memorijskog kontrolera (*oe*, *we*, *we_me*) sa slika 6 i 7, možemo videti da su izlazni signali sa slike 7 zakašnjena verzija

odgovarajućih signala sa slike 6 sa kašnjenjem od jedne periode *clk* signala. Ovo je upravo ponašanje koje se moglo očekivati u slučaju kada je konačni automat modelovan samo pomoću jednog procesa, jer se u ovom slučaju na svaki od izlaznih signala implicitno dodaje po jedan flip flop. U slučaju memorijskog kontrolera, ovako generisani izlazni kontrolni signali bili bi neupotrebljivi jer bi aktivirali memoriju u pogrešnim trenucima.

Konačno, ukoliko bi smo prilikom simualcije koristili *two_seg_arch_ver2* arhitekturu, koja koristi vektorski tip podataka za reprezentaciju stanja u kome se konačni automat nalazi, rezultati simulacije trebalo bi de odgovaraju talasnim oblicima prikazanim na slici 8.




Slika 8. Rezultat simulacije modela memorijskog kontrolera baziranog na korišćenju dve *process* naredbe i vektorskog tipa podataka za reprezentacije stanja automata

Na slici 8 možemo videti da signal *state*, koji predstavlja unutrašnje stanje memorijskog kontrolera, sada uzima konkretne binarne vrednosti. Vrednosti koje uzima signal *state* tačno odgovaraju vrednostima odgovarajućih konstanti iz modela prikazanog u varijanti 4. U slučaju hardverske verifikacije ovo bi nam omogućilo tačnu korelaciju obzerviranih vrednosti *state* signala sa odgovarajućim stanjima konačnog automata. Na primer, svaki put kada *state* signal ima vrednost "001" možemo zaključiti da se tada konačni automat nalazi u *read1* stanju. Ovo je zbog toga što smo u modelu prikazanom u varijanti 4 *read1* stanje kodovali sa kodnom rečju "001" (vidi vrednost konstante read1 unutar modela iz varijante 4). Upoređivanjem vremenskih oblika *state* signala sa slike 6 i slike 8 takođe možemo izvesti ovaj isti zaključak.

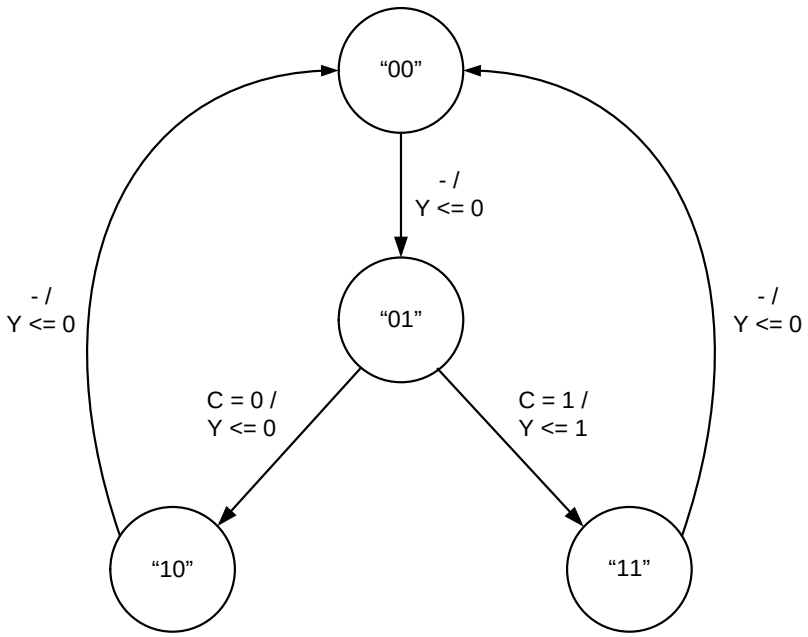
**Zadaci za vežbu**

Zadatak 1:
Napisati VHDL model konačnog automata koji je zadat sledećom tablicom prelaza/izlaza

| Trenutno stanje | Naredno stanje/izlaz | | | |
|---|---|---|---|---|
| | Ulazi: (x, y) | | | |
| | (0, 0) | (0, 1) | (1, 0) | (1, 1) |
| *S1* | *S3*/0 | *S3*/0 | *S2*/1 | *S2*/1 |
| *S2* | *S1*/0 | *S1*/0 | *S1*/0 | *S1*/0 |
| *S3* | *S4*/1 | *S4*/1 | *S4*/1 | *S4*/1 |
| *S4* | *S1*/0 | *S2*/0 | *S1*/0 | *S2*/0 |

Za razvijeni VHDL model napisati odgovarajući testbenč pomoću kojega će biti moguće izvršiti funkcionalnu verifikaciju modela.

Zadatak 2:
Konačni automat zadat je pomoću sledećeg dijagrama stanja



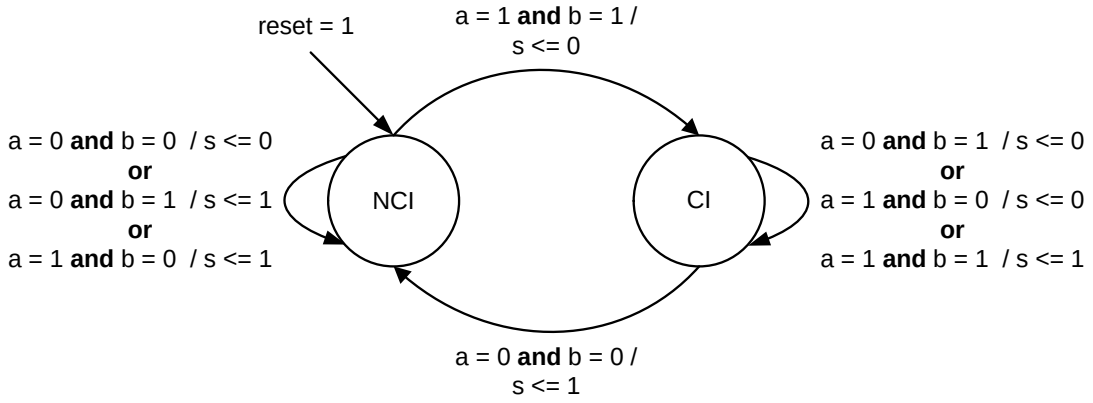

Konačni automat ima jedan jednobitni ulazni port, *C*, i jedan jednobitni izlazni port, *Y*. Napisati odgovarajući VHDL model ovako definisanog konačnog automata. Za razvijeni VHDL model napisai odgovarajući testbenč pomoću kojega će biti moguće izvršiti funkcionalnu verifikaciju modela.

Zadatak 3:

Nacrtati dijagram stanja koji modeluje rad 2-bitnog binarnog brojača koji ima mogućnost obostranog brojanja. Na osnovu dijagrama stanja napisati VHDL model konačnog automata koji će zapravo predstavljati 2-bitni obostrani binarni brojač. Za razvijeni VHDL model napisai odgovarajući testbenč pomoću kojega će biti moguće izvršiti funkcionalnu verifikaciju modela.

Zadatak 4:

Konačni automat serijskog sabirača zadat je pomoću sledećeg dijagrama stanja




Konačni automat ima jedan dva jednobitna ulazna porta, *a i b*, i jedan jednobitni izlazni port, *s*. Princip rada serijskog sabirača je sledeći. Nakon resetovanja serijskog sabirača, pojedinačni biti dva *n*-bitna binarna broja koja želimo da saberemo serijski se dovode na ulaze *a* i *b* konačnog automata tokom *n* taktova počevši od bitova najmanje težine (LSB bitovi). U svakom taktu na izlazu *s* pojavljuje se vrednost sume na tekućoj poziciji, počevši od LSB bita. Nakon *n* taktova proces sabiranja dva *n*-bitna je završen.

Napisati odgovarajući VHDL model ovako definisanog konačnog automata. Za razvijeni VHDL model napisai odgovarajući testbenč pomoću kojega će biti moguće izvršiti funkcionalnu verifikaciju modela.

Zadatak 5:

Nacrtati dijagram stanja konačnog automata koji će biti u stanju da detektuje prisustvo ili odsustvo odgovarajuće sekvence na svom ulazu. Konačni automat koji je potrebno projektovati ima jedan jednobitni ulaz, *w*, i jedan jednobitni izlaz, *z*. U slučaju da su poslednje dve vrednosti ulaza *w* bile “00” ili “11” izlaz *z* treba da bude postavljen na 1, a u slučaju da su poslednje dve vrednosti ulaza *w* bile “00” ili “11” izlaz *z* treba da bude postavljen na 0.

Na osnovu dijagrama stanja napisati VHDL model konačnog automata. Za razvijeni VHDL model napisai odgovarajući testbenč pomoću kojega će biti moguće izvršiti funkcionalnu verifikaciju modela.