POGLAVLJE 7.
PROJEKTOVANJE MEMORIJE

Ovo poglavlje ilustruje metode implementacije RAM, FIFO i LIFO memorija u okviru programabilnih sekvencijalnih mreža. Svi zadaci su rešeni pomoću VHDL jezika za opis fizičke arhitekture.

7.1 ZADATAK:

Isprojektovati RAM memoriju 16×8 bita. Blok dijagram entiteta koji treba realizovati prikazuje Slika 7.1.

Slika 7.1: Blok dijagram RAM memorije

Ulazni signali `RD` i `WR` predstavljaju kontrolne signale za čitanje i upis podataka respektivno. Oba signala su aktivna na visokom nivou. Ulazni vektor `DATA` sadrži osmobitni podatak koji se upisuje u memoriju signalom `WR`. Adresu memorijske lokacije kojoj se pristupa, sadrži ulazni četvorobitni vektor `ADDRESS`.

Tokom operacije čitanja, izlazni vektor `DATA` sadrži sadržaj memorijske lokacije kojoj se pristupa. Inače, vektor je u stanju visoke impedanse.

Memoriju isprojektovati pomoću VHDL jezika za opis fizičke arhitekture.

REŠENJE:

Sa ciljem realizacije tražene memorije, moraju se obezbediti tri preduslova:

1. niz od 16 osmobitnih lokacija
2. mehanizam upisa u adresiranu lokaciju
3. mehanizam čitanja adresirane lokacije

Prvi uslov se realizuje formiranjem novog tipa podataka koji će predstavljati niz osmobitnih lokacija. Ovaj tip je u realizovanom kodu označen sa `tRAM`. Potom se sa ovim tipom formira signal, `sRAM`, koji će predstavljati niz od 16 osmobitnih lokacija, tj. RAM memoriju. Predstavljeno VHDL sintoksom to ima sledeći oblik:

```
TYPE tRAM IS ARRAY (15 DOWNTO 0) OF std_logic_vector(7 DOWNTO 0);
SIGNAL sRAM: tRAM;
```

Upis u RAM memoriju se izvršava kada je aktivan ulazni signal `iWR`. U tom slučaju se na adresiranu lokaciju vektorom `iADDRESS` upisuje sadržaj ulazne magistrale `iDATA`. To se realizuje jednim VHDL procesom sledećeg oblika:

```
PROCESS (iWR, iADDRESS, iDATA) BEGIN
  IF (iWR = '1') THEN
    sRAM(iADDRESS) <= iDATA;
  END IF;
END PROCESS;
```

Na ovaj način se realizuju memorijski elementi tipa LATCH, koji se aktiviraju na određeni nivo ulaznog signala (u ovom slučaju `iWR=1`), a ne na ivicu signala (obično takt signala) kao što je slučaj sa flip-flopovima.

Čitanje sadržaja adresirane lokacije RAM memorije vektorom `iADDRESS` se izvršava kada je aktivan ulazni signal `iRD`. Tada izlazni vektor podataka `oDATA` sadrži sadržaj adresirane lokacije. U suprotnom slučaju ovaj vektor je u stanju visoke impedanse. Opisana operacija se realizuje sa jednim multiplekserom 2×1:

```
PROCESS (iRD, iADDRESS, sRAM) BEGIN
  IF (iRD = '1') THEN
    oDATA <= sRAM(iADDRESS);
  ELSE
    oDATA <= (OTHERS => 'Z');
  END IF;
END PROCESS;
```

Na ovaj način su obezbeđeni svi uslovi za realizaciju tražene RAM memorije. U nastavku je prikazan VHDL kod jednog načina realizacije RAM memorije 16×8 bita.

```
LIBRARY ieee;
  USE ieee.std_logic_1164.all;
  USE ieee.numeric_std.all;

ENTITY RAM IS
  GENERIC (
                    -- pretpostavljena sirina reci je 8 bita
   pDATA_WIDTH:    integer := 8;
                    -- pretpostavljena sirina adresne reci je 4 bita
   pADDRESS_WIDTH: integer := 4;
                    -- pretpostavljen broj reci je 16
   pNO_OF_WORDS:   integer := 16);

  PORT (
   iRD, iWR: IN  std_logic;
   iDATA:    IN  std_logic_vector(pDATA_WIDTH-1 DOWNTO 0);
   iADDRESS: IN  std_logic_vector(pADDRESS_WIDTH-1 DOWNTO 0);
   oDATA:    OUT std_logic_vector(pDATA_WIDTH-1 DOWNTO 0) );
END RAM;

ARCHITECTURE ARH_RAM OF RAM IS
 -- funkcija za konverziju vektora tipa
 -- STD_LOGIC_VECTOR u vektor tipa INTEGER
 -- ulazni parametar je oznacen sa ARG
 FUNCTION CONV_TO_INTEGER (ARG: STD_LOGIC_VECTOR) RETURN INTEGER IS
   -- potrebno je odredsiti broj bita ulaznog vektora
   -- indeks krajnjeg levog bita ulaznog parametra
   CONSTANT cARG_LEFT: INTEGER := ARG'LENGTH-1;
   -- alias na ulazni vektor sa tipom koji ima ograniceni broj bita
   ALIAS    XXARG: STD_LOGIC_VECTOR(cARG_LEFT DOWNTO 0) is ARG;
   -- promenljiva koja ce primiti vrednost ulaznog vektora
   -- na osnovu koje ce se vrsiti konverzija u tip INTEGER
   VARIABLE vXARG:  STD_LOGIC_VECTOR(cARG_LEFT DOWNTO 0);
```

```
    -- rezultat konverzije
    VARIABLE vRESULT: INTEGER := 0;
  BEGIN
    -- telo funkcije za konverziju
    vXARG := XXARG;
    FOR I IN vXARG'RANGE LOOP
      vRESULT := vRESULT + vRESULT;
      IF vXARG(I) = '1' THEN
        vRESULT := vRESULT + 1;
      END IF;
    END LOOP;
    RETURN vRESULT;
  END CONV_TO_INTEGER;

  -- tip koji opisuje niz lokacija koje predstavljaju RAM memoriju
  TYPE tRAM IS ARRAY (pNO_OF_WORDS-1 DOWNTO 0) OF
               std_logic_vector(pDATA_WIDTH-1 DOWNTO 0);
  -- RAM memorija
  SIGNAL sRAM: tRAM;
  -- adresa lokacije kojoj se pristupa
  SIGNAL sADDRESS: integer RANGE 0 TO pNO_OF_WORDS-1;
BEGIN

  -- pristup elementu niza se moze realizovati
  -- samo signalom tipa INTEGER
  sADDRESS <= CONV_TO_INTEGER(iADDRESS);

  -- upis u RAM memoriju
  PROCESS (iWR, sADDRESS, iDATA) BEGIN
    IF (iWR = '1') THEN
      sRAM(sADDRESS) <= iDATA;
    END IF;
  END PROCESS;

  -- citanje iz RAM memorije
  PROCESS (iRD, sADDRESS, sRAM) BEGIN
    IF (iRD = '1') THEN
      oDATA <= sRAM(sADDRESS);
    ELSE
      oDATA <= (OTHERS => 'Z');
    END IF;
  END PROCESS;

END ARH_RAM;
```

Pošto su svi ulazno/izlazni signali realizovani tipom `STD_LOGIC`, javlja se problem pristupa adresiranoj lokaciji niza osmobitnih vektora `sRAM`. Problem se izražava u tome što je prema VHDL standardu pristup lokaciji niza moguće realizovati samo tipom `INTEGER`. Zbog toga je urađena konverzija ulaznog adresnog vektora `iADDRESS` tipa `STD_LOGIC_VECTOR` u signal `sADDRESS` tipa `INTEGER`.

Za potrebe konverzije formirana je funkcija `CONV_TO_INTEGER` koja preuzima parametar tipa `STD_LOGIC_VECTOR` proizvoljne širine i konvertuje ga u odgovarajuću `INTEGER` vrednost koja je i izlazna vrednost funkcije.

Formiranjem konstante cARG_LEFT određuje se širina ulaznog parametra ARG. To se jednostavno realizuje korišćenjem VHDL atributa LENGTH. Nakon toga se formira alias XXARG na ulazni parametar. Time se ulaznom parametru dodeljuje oznaka vektora poznate širine. Na kraju se formira promenljiva vARG tipa STD_LOGIC_VECTOR poznate širine. Ovoj promenljivoj se dodeljuje vrednost ulaznog parametra i celokupna konverzija se izvršava nad datom promenljivom.

Nakon određivanja širine ulaznog parametra u telu funkcije se izvršava konverzija ulazne vrednosti u tip INTEGER određivanjem vrednosti svakog pojedinačnog bita ulaznog parametra i izvršavanjem određenih matematičkih operacija u zavisnosti od vrednosti bita koji se analizira.

Realizovani VHDL kod je potpuno modularan pošto ima generičke parametre koji omogućuju realizaciju RAM memorije sa proizvoljnim brojem lokacija sa proizvoljnim brojem bita. To je realizovano formiranjem generičke liste od tri parametra:

```
GENERIC (
                  -- pretpostavljena sirina reci je 8 bita
 pDATA_WIDTH:     integer := 8;
                  -- pretpostavljena sirina adresne reci je 4 bita
 pADDRESS_WIDTH: integer := 4;
                  -- pretpostavljen broj reci je 16
 pNO_OF_WORDS:    integer := 16);
```

Pretpostavljene vrednosti generičkih parametara su u skladu sa postavkom zadatka koja traži realizaciju RAM memorije 16×8 bita. U slučaju kada se instancira realizovani modul RAM memorije i ne navede se generička lista parametara, biće realizovana RAM memorija sa 16 osmobitnih lokacija. Navođenjem generičke liste parametara može se formirati bilo koji oblik RAM memorije. Naravno, treba voditi računa da broj memorijskih reči ne sme da bude veći od maksimalnog mogućeg koji dozvoljuje širina adresne magistrale, odnosno mora da važi sledeća relacija:

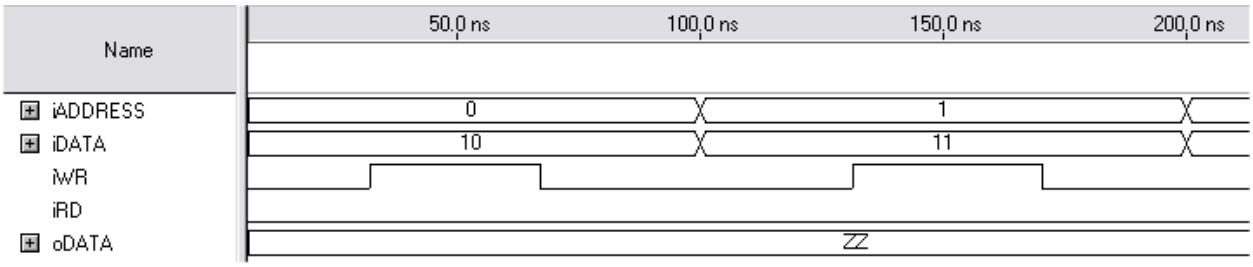
$$pNO\_OF\_WORDS \leq 2^{pADDRESS\_WIDTH}$$

Na primer za realizaciju RAM memorije 100×16 bita potrebno je navesti sledeću generičku listu parametara prilikom instanciranja realizovanog modula RAM memorije:

```
GENERIC MAP(pDATA_WIDTH    => 16;
            pADDRESS_WIDTH => 7;   -- 2^7=128≥100
            pNO_OF_WORDS   => 100)
```

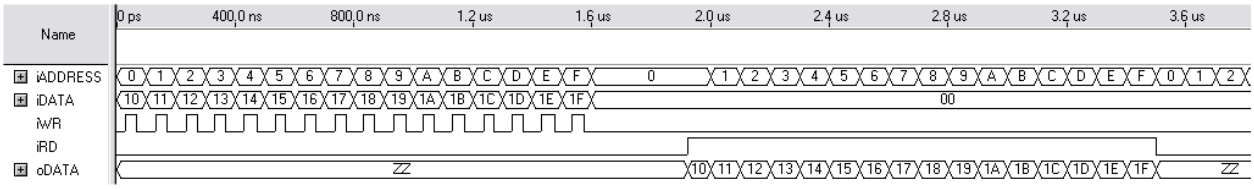
Slika 7.2 prikazuje vremenski dijagram upisa u RAM memoriju dva podatka. Prvi podatak, $iDATA=10_{HEX}$, se upisuje kada se signal iWR postavi na visoki logički nivo, t=30ns. U tom trenutku adresna magistrala iADDRESS sadrži

vrednost 0_{HEX}, te se dati podatak upisuje na adresu nula. Drugi podatak, `iDATA=10`$_{HEX}$, se upisuje na adresu 1 u vremenskom trenutku t=130ns.



Slika 7.2: Vremenski dijagram upisa u RAM memoriju

Simulacioni dijagram upisa u svih 16 memorijskih lokacija i čitanje upisanog sadržaja iz svih lokacija prikazuje Slika 7.3. Sa slike se vidi da su upisani podaci 10_{HEX} do $1F_{HEX}$ na memorijskim lokacijama na adresama 0_{HEX} do F_{HEX}. Upisani podaci su i pročitani tokom ciklusa čitanja, što ukazuje na dobru realizaciju RAM memorije.



Slika 7.3: Vremenski dijagram simulacije upisa i čitanja RAM memorije

7.2 ZADATAK:

Isprojektovati RAM memoriju 16×8 bita pomoću ALTERA megafunkcije `LPM_RAM_DQ`. Izvršiti baferovanje svih ulaznih i izlaznih signal `LPM_RAM_DQ` memorijskog modula. Memoriju isprojektovati pomoću VHDL jezika za opis fizičke arhitekture.

REŠENJE:

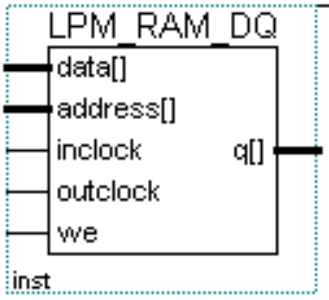
Kada se projektuje memorija u okviru programibilnih sekvencijalnih mreža na način koji je prikazan u prethodnom zadatku, za implementaciju jednog memorijskog bita iskoristi se jedan memorijski element (flip-flop). To za traženu RAM memoriju 16×8 bita iznosi ukupno 128 flip-flopova, što nije zanemarljiv broj. Kod većih memorijskih zahteva broj utrošenih flip-flopova je još veći.

Iz tih razloga proizvođači programibilnih sekvencijalnih mreža omogućuju druge metode realizacije memorijskih modula. Tako na primer, proizvođač ALTERA u okviru svoje serije programibilnih sekvencijalnih mreža FLEX isporučuje namenske matrice memorijskih elemenata. Ove matrice se nazivaju EAB – *Embedded Array Block*. Pristup ovim blokovima je moguć preko takozvanih LPM megafunkcija. Tako postoje megafunkcije za implementaciju

RAM memorije (LPM_RAM_DQ), dvopristupne RAM memorije (LPM_RAM_DP), ROM memorije (LPM_ROM), FIFO memorije (LPM_FIFO) i slično.

Svaka od ovih memorijskih megafunkcija sadrži listu portova na koje se povezuju odgovarajući signali, kao i listu generičkih parametara pomoću kojih se konfiguriše širina adresne magistrale, broj reči u memoriji, širina magistrale podataka, omogućuje baferovanje signala, postavlja početna vrednost memorijskih lokacija i ostali parametri u zavisnosti od korišćene megafunkcije.

U ovom zadatku se ilustruje projektovanje RAM memorije sa LPM_RAM_DQ megafunkcijom. Slika 7.4 prikazuje blok dijagram ove komponente, gde se vidi lista portova i grupa karakterističnih parametara.



Parameter	Value
LPM_ADDRESS_CONTROL	"REGISTERED"
LPM_INDATA	"REGISTERED"
LPM_NUMWORDS	16
LPM_OUTDATA	"REGISTERED"
LPM_WIDTH	8
LPM_WIDTHAD	4

Slika 7.4: Blok dijagram LPM_RAM_DQ megafunkcije

Ulazni podaci se upisuju u memoriju preko data porta, na adresu koja postoji na address magistrali ako je aktivan we signal. U slučaju baferovanja ulaznih signala koristi se takt signal prisutan na inclock portu. Izlazna magistrala podataka q se baferuje sa takt signalom prisutnim na outclock portu.

Širina magistrale podataka se određuje parametrom LPM_WIDTH, dok se širina adresne magistrale definiše LPM_WIDTHAD parametrom. Broj reči u memoriji se može ograničiti sa parametrom LPM_NUMWORDS. Baferovanje adresne magistrale i we signala se realizuje LPM_ADDRESS_CONTROL parametrom, dok se za baferovanje ulazne i izlazne magistrale podataka koriste parametri LPM_INDATA i LPM_OUTDATA respektivno.

Celokupna VHDL deklaracija ove komponente sa inicijalnim vrednostima generičkih parametara je sledeća:

```
COMPONENT lpm_ram_dq
   GENERIC (
      LPM_WIDTH: POSITIVE;
      LPM_WIDTHAD: POSITIVE;
      LPM_NUMWORDS: NATURAL := 0;
      LPM_INDATA: STRING := "REGISTERED";
      LPM_ADDRESS_CONTROL: STRING := "REGISTERED";
      LPM_OUTDATA: STRING := "REGISTERED";
      LPM_FILE: STRING := "UNUSED";
      LPM_TYPE: STRING := "LPM_RAM_DQ";
      LPM_HINT: STRING := "UNUSED");
```

```
    PORT (
      data: IN STD_LOGIC_VECTOR(LPM_WIDTH-1 DOWNTO 0);
      address: IN STD_LOGIC_VECTOR(LPM_WIDTHAD-1 DOWNTO 0);
      inclock, outclock: IN STD_LOGIC := '0';
      we: IN STD_LOGIC;
      q: OUT STD_LOGIC_VECTOR(LPM_WIDTH-1 DOWNTO 0));
END COMPONENT;
```

Opis preostalih parametara, kao i opis i primeri korišćenja svih LPM megafunkcija mogu se pronaći u dokumentaciji koja se isporučuje sa *Quartus II* programskim paketom (*Quartus II Help*).

Da bi se u VHDL kodu mogle koristiti LPM megafunkcije, mora se u proces prevođenja VHDL koda uključiti biblioteka sa opisom LPM komponeti:

```
LIBRARY lpm;
  USE lpm.lpm_components.all;
```

Nakon toga, preostaje jedino da se u VHDL kodu na odgovarajućem mestu postavi instanca željene komponente sa listom povezanosti portova (`PORT MAP`) i sa listom generičkih parametara (`GENERIC MAP`).

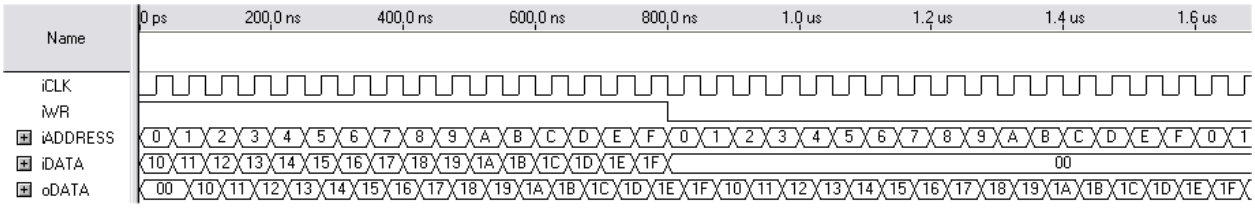
U slučaju implementacije memorije koja se traži u ovom zadatku izgled VHDL koda je sledeći:

```
LIBRARY ieee;
  USE ieee.std_logic_1164.all;
-- ukljucivanje biblioteke sa LPM megafunkcijama
LIBRARY lpm;
  USE lpm.lpm_components.all;

ENTITY RAM IS
  GENERIC (
                    -- pretpostavljena sirina reci je 8 bita
    pDATA_WIDTH:    integer := 8;
                    -- pretpostavljena sirina adresne reci je 4 bita
    pADDRESS_WIDTH: integer := 4;
                    -- pretpostavljen broj reci je 16
    pNO_OF_WORDS:   integer := 16);
  PORT (
    iCLK, iWR: IN  std_logic;
    iDATA:     IN  std_logic_vector(pDATA_WIDTH-1 DOWNTO 0);
    iADDRESS:  IN  std_logic_vector(pADDRESS_WIDTH-1 DOWNTO 0);
    oDATA:     OUT std_logic_vector(pDATA_WIDTH-1 DOWNTO 0) );
END RAM;

ARCHITECTURE ARH_RAM OF RAM IS BEGIN

  -- instanciranje RAM megafunkcije LPM_RAM_DQ
  eLPM_RAM: LPM_RAM_DQ
    -- postavljanje generickih parametara LPM_RAM_DQ megafunkcije
    GENERIC MAP (
                          -- sirina adresne reci
      LPM_WIDTHAD         => pADDRESS_WIDTH,
                          -- broj reci u memoriji
      LPM_NUMWORDS        => pNO_OF_WORDS,
```

```
                              -- sirina reci
      LPM_WIDTH               => pDATA_WIDTH,
                              -- baferovanje ulaznih podataka
      LPM_INDATA              => "REGISTERED",
                              -- baferovanje adresne magistrale
      LPM_ADDRESS_CONTROL => "REGISTERED",
                              -- baferovanje izlaznih podataka
      LPM_OUTDATA             => "REGISTERED" )
    -- povezivanje portova
    PORT MAP (
      data     => iDATA,     -- ulazni vektor podataka
      address  => iADDRESS,  -- adresa lokacije kojoj se pristupa
      we       => iWR,       -- dozvola upisa
      inclock  => iCLK,      -- takt za baferovanje ulaznih podataka
      outclock => iCLK,      -- takt za baferovanje izlaznih podataka
      q        => oDATA);    -- izlazni vektor podataka

END ARH_RAM;
```

Radi jednostavnijeg čitanja VHDL koda, zadržane su oznake ulazno/izlaznih signala iste kao i u prethodom zadatku. Jedina razlika je uvođenje takt signala `iCLK` koji se koristi za baferovanje svih ulaznih i izlaznih signala `LPM_RAM_DQ` komponente, i izostavljanje `iRD` signala. Signal dozvole čitanja je izostavljen jer `LPM_RAM_DQ` komponenta uvek na izlazu postavlja sadržaj adresirane lokacije.

Vremenski dijagram ovakve realizacije RAM memorije prikazuje Slika 7.5. Kašnjenje vrednosti adresirane lokacije za jedan takt postoji zbog baferovanja ulaznih i izlaznih signala.



Slika 7.5: Simulacija rada realizovane memorije

7.3 ZADATAK:

Isprojektovati magazinsku memoriju tipa FIFO. Memorija maksimalno može da sadrži 10 osmobitnih reči. Blok dijagram entiteta koji treba realizovati prikazuje Slika 7.6.

Slika 7.6: Blok dijagram FIFO memorije

Ulazni signali RD i WR predstavljaju kontrolne signale za čitanje i upis podataka respektivno. Oba signala su aktivna na visokom nivou. Ulazni vektor DATA sadrži osmobitni podatak koji se upisuje u memoriju signalom WR. Sve operacije se obavljaju na rastuću ivicu takt signala CLK. Ulazni signal RST, aktivan na viskom nivou, postavlja memoriju u inicijalno stanje. Inicijalno stanje memorije je kada je memorija potpuno ispražnjena.

Tokom operacije čitanja, izlazni vektor DATA sadrži sadržaj memorijske lokacije kojoj se pristupa. Inače, vektor je u stanju visoke impedanse. Izlazni signali EMPTY i FULL ukazuju na stanje ispražnjenosti memorije i na potpunu popunjenost memorije.

Memoriju isprojektovati pomoću VHDL jezika za opis fizičke arhitekture.

REŠENJE:

FIFO (*First In First Out*) memorija predstavlja vrstu magazinske memorije pomoću koje se implementiraju redovi čekanja. Niz bita, organizovanih kao reč, se smeštaju u memoriju po redosledu nailazka, komandom upisa u memoriju. Operacijom čitanja se najstarija upisana reč prenosi na izlaz i odstranjuje iz memorije. Mehanizam pristupa u ovoj organizaciji ne zahteva prisustvo spoljnje adrese.

FIFO memorija se karakteriše sa dva parametra:

- broj bita u reči koji se smeštaju u memoriju
- broj reči koje memorija sadrži

Stoga je zgodno ova dva parametra postaviti kao generička i omogućiti njihovo proizvoljno postavljanje prilikom instanciranja entiteta FIFO memorije koja se realizuje u ovom zadatku. U VHDL kodu koji sledi ova dva parametra su označena identifikatorima pWIDE i pDEEP.

```
LIBRARY ieee;
  USE ieee.std_logic_1164.all;
  USE ieee.numeric_std.all;

ENTITY FIFO IS
  GENERIC (
                  -- pretpostavljena sirina reci je 8 bita
          pWIDE: integer := 8;
                  -- pretpostavljen broj reci u memoriji je 10
          pDEEP: integer := 10);
  PORT (
    iCLK, iRST: IN  std_logic;
    iRD, iWR:   IN  std_logic;
    iDATA:      IN  std_logic_vector(pWIDE-1 DOWNTO 0);
    oDATA:      OUT std_logic_vector(pWIDE-1 DOWNTO 0);
    oEMPTY, oFULL: OUT std_logic );
END FIFO;
```

```
ARCHITECTURE ARH_FIFO OF FIFO IS
   -- tip koji opisuje niz lokacija koje predstavljaju FIFO memoriju
  TYPE tFIFO IS ARRAY (pDEEP-1 DOWNTO 0) OF
                      std_logic_vector(pWIDE-1 DOWNTO 0);

  -- FIFO memorija
  SIGNAL sFIFO: tFIFO;
  -- pokazivac na lokaciju u koju ce se upisati novi sadrzaj
  -- poslednja lokacija ukazuje na popunjenost steka
  SIGNAL sWRPTR: integer RANGE 0 TO pDEEP;
  -- signali koji ukazuju da li je memorija prazna ili puna
  SIGNAL sEMPTY, sFULL: std_logic;
BEGIN

  -- memorija je prazna ako je pokazivac na nuli
  sEMPTY <= '1' WHEN (sWRPTR = 0)     ELSE '0';
  -- memorija je puna ako je pokazivac na maksimalnoj vrednosti
  sFULL  <= '1' WHEN (sWRPTR = pDEEP) ELSE '0';

  -- proces koji opisuje FIFO memoriju
  PROCESS (iCLK) BEGIN
    IF (iCLK'EVENT AND iCLK='1') THEN
      IF (iRST = '1') THEN          -- sinhroni reset
        FOR i IN sFIFO'RANGE LOOP  -- postavi sve lokacije na nulu
          sFIFO(i) <= (OTHERS => '0');
        END LOOP;
      ELSE
        IF ((iRD = '1') AND (sEMPTY = '0')) THEN
          -- citanje podataka ako memorija nije prazna
          -- pomeri sve lokacije za jednu blize prvoj lokaciji
          FOR i IN 0 TO (pDEEP - 2) LOOP
            sFIFO(i) <= sFIFO(i+1);
          END LOOP;
        ELSIF ((iWR = '1') AND (sFULL = '0')) THEN
          -- upis podataka ako memorija nije puna
          -- upisi na lokaciju na koju ukazuje pokazivac
          sFIFO(sWRPTR) <= iDATA;
        END IF;
      END IF;
    END IF;
  END PROCESS;

  -- kontrola pokazivaca lokacije upisa
  PROCESS (iCLK) BEGIN
    IF (iCLK'EVENT AND iCLK='1') THEN
      IF (iRST = '1') THEN -- sinhroni reset
        sWRPTR <= 0;
      ELSE
        IF ((iWR = '1') AND (sFULL = '0')) THEN
          -- ako je podatak upisan -> uvecaj za jedan
          sWRPTR <= sWRPTR + 1;
        ELSIF ((iRD = '1') AND (sEMPTY = '0')) THEN
          -- ako je podatak procitan -> smanji za jedan
          sWRPTR <= sWRPTR - 1;
        END IF;
      END IF;
    END IF;
  END PROCESS;
```

```
    -- odredjivanje izlaznog vektora
  PROCESS (iRD, sEMPTY, sFIFO) BEGIN
    IF ((iRD = '1') AND (sEMPTY = '0')) THEN
      -- ako je citanje dozvoljeno ->
      -- procitaj sadrzaj prve lokacije u memoriji
      oDATA <= sFIFO(0);
    ELSE
      -- inace postavi izlaz u stanje visoke impedanse
      oDATA <= (OTHERS => 'Z');
    END IF;
  END PROCESS;

  -- prosledjivanje signala na izlazne portove
  oEMPTY <= sEMPTY;
  oFULL  <= sFULL;

END ARH_FIFO;
```

Memorija je implementirana kao niz od `pDEEP` lokacija širine određene parametrom `pWIDE`. Dati niz je predstavljen tipom `tFIFO`, a sama memorija je predstavljena signalom `sFIFO`.

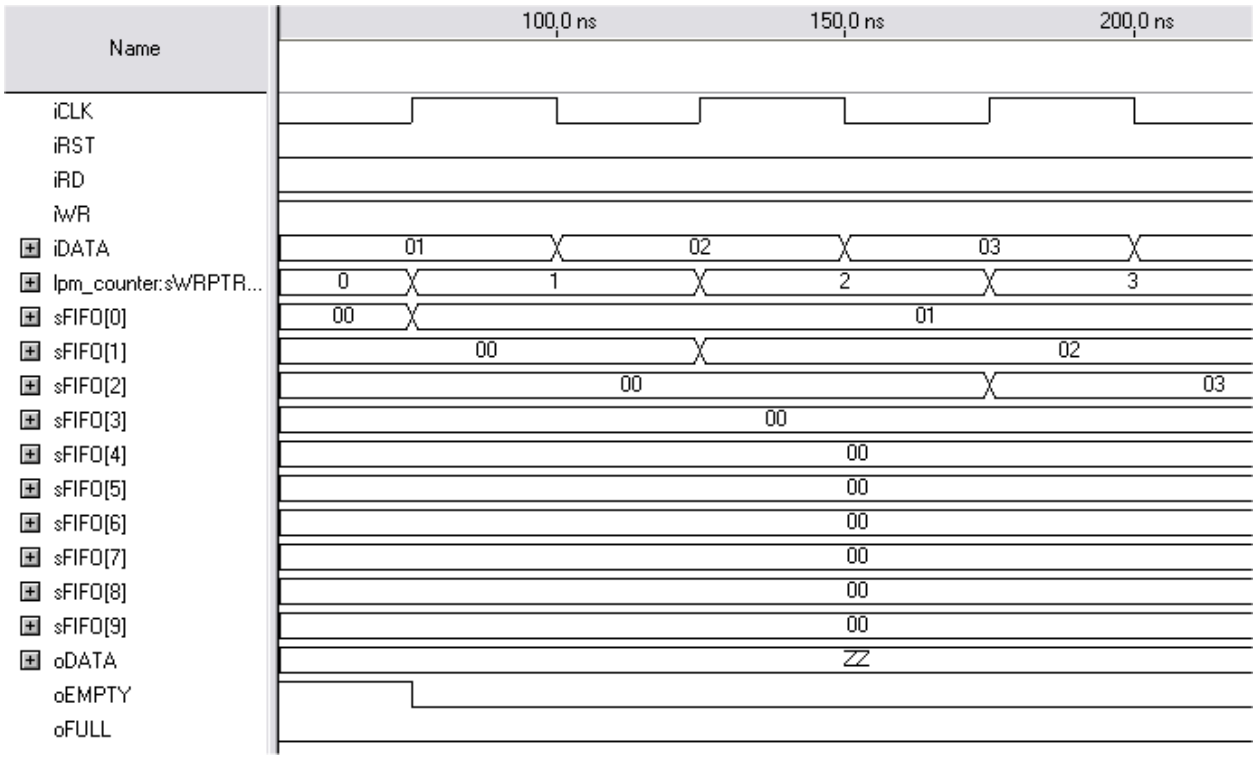
Pristup memoriji je realizovan na sledeći način. Čitanje najstarijeg upisanog podatka se uvek izvršava sa lokacije sa adresom 0 (nula). Prilikom čitanja, aktiviranjem signala `iRD`, se podatak sa ove lokacije postavi na izlaznu magistralu podataka `oDATA`. Sa sledećom rastućom ivicom takt signala `iCLK`, sadržaj svih ostalih lokacija se pomeri na lokaciju sa adresom manjom za jedan, tj, za jedno mesto bliže nultoj lokaciji. Pri tome sadržaj poslednje lokacije ostaje nepromenjen.

Upis podatka u memoriju se izvršava na lokaciji na koju ukazuje pokazivač `sWRPTR`. Stanje ovog pokazivača se automatski uvećava za jedan nakon izvršenog upisa. Takođe, stanje pokazivača se ažurira i nakon operacije čitanja smanjenjem njegove vrednosti za jedan. Dati pokazivač je realizovan tipom `INTEGER` radi direktnog omogućavanja adresiranja lokacije u implementiranom nizu memorijskih lokacija.

Potrebno je primetiti da pokazivač `sWRPTR` može primiti vrednost koja ne adresira nijednu lokaciju (`sWRPTR=pDEEP`). Ovakva implementacija omogućava jednostavno utvrđivanje popunjenosti svih memorijskih lokacija. Naime, pošto pokazivač uvek pokazuje na prvu slobodnu lokaciju upis podatka se uvek izvršava na lokaciju na koju ukazuje pokazivač. Nakon upisa se vrednost pokazivača uvećava za jedan. Time se prilikom upisa poslednjeg podatka u memoriju (`sWRPTR=pDEEP-1`), vrednost pokazivača postavlja na nepostojeću memorijsku lokaciju i automatski ukazuje na popunjenost memorije. Sa druge strane, memorija je prazna kada je `sWRPTR=0`. Na ovaj način se jednostavno realizuju signali koji govore o popunjenosti (`oFULL`) i ispražnjenosti (`oEMPTY`) memorije.

Ažuriranje stanja pokazivača `sWRPTR`, kao i upis i čitanje iz memorije je kontrolisano sa ulaznim signalima `iWR` i `iRD`, kao i sa signalima `sFULL` i `sEMPTY` koji ukazuju na nemogućnost upisa u memoriju (sve memorijske lokacije su pune) i nemogućnost čitanja iz memorije (memorija je prazna).

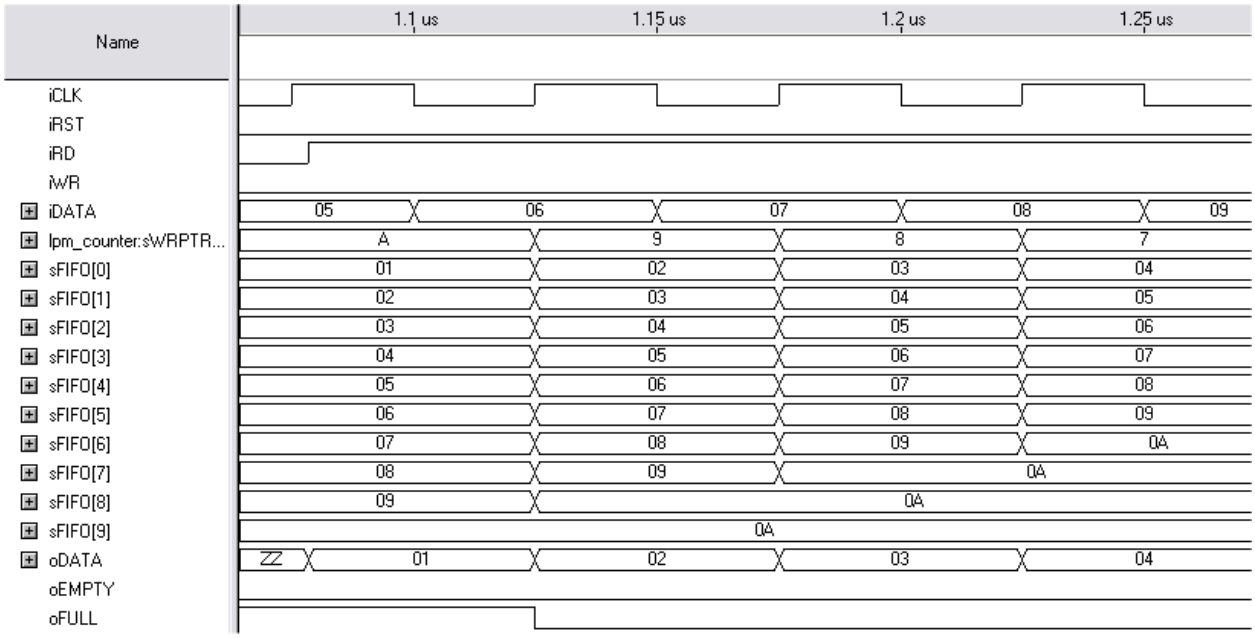
Slika 7.7 prikazuje vremenski dijagram upisa u FIFO memoriju tri podatka. Prvi podatak, `iDATA=01`$_{HEX}$, se upisuje na rastuću ivicu takt signala `iCLK` u vremenskom trenutku t=75ns, pošto je signal `iWR` postavljen na visokom logičkom nivou. U tom trenutku se prednost pokazivača `sWRPTR` uvećava za jedan i ukazuje na prvu sledeću slobodnu lokaciju. Upis prvog podatka se vidi i promenom stanja vektora `sFIFO`. Drugi podatak, `iDATA=02`$_{HEX}$, se upisuje na adresu 1 na rastuću ivicu takt signala `iCLK` u vremenskom trenutku t=125ns, dok se treći podatak `iDATA=03`$_{HEX}$ upisuje na lokaciju sa adresom 2 na rastuću ivicu takt signala `iCLK` u vremenskom trenutku t=175ns.



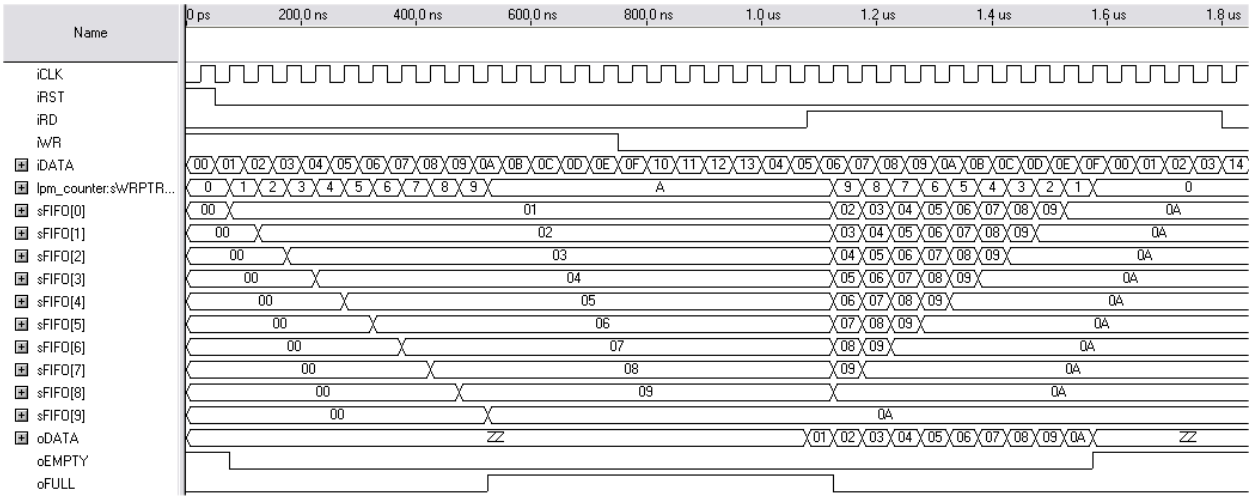
Slika 7.7: Ciklus upisa u FIFO memoriju

Slika 7.8 prikazuje vremenski ciklus čitanja podataka iz FIFO memorije. Vidi se da je prvo pročitan prvi upisan podatak sa vrednošću 01_{HEX}. Nakon toga se redom sa sledeće tri rastuće ivice takt signala čitaju podaci 02_{HEX}, 03_{HEX} i 04_{HEX}. Sa svakim čitanjem podatka iz FIFO memorije, izvršava se pomeranje za jedan sadržaja niza lokacija FIFO memorije prema lokaciji sa najmanjom adresom. To se može videtim promenom sadržaja vektora `sFIFO` sa svakom rastućom ivicom takt signala.

Kompletan ciklus upisa podataka u svih 10 lokacija FIFO memorije i čitanje istih prikazuje Slika 7.9. Sa slike se vidi da upis nije realizovan kada je FIFO memorija popunjena (`oFULL=1`). Isto tako čitanje memorije nije omogućeno kada je `oEMPTY=1`, tj. kada je memorija prazna.



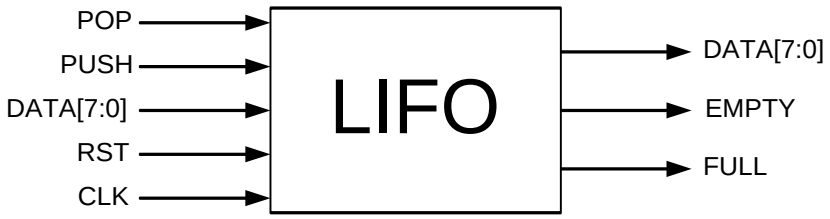
Slika 7.8: Ciklus čitanja iz FIFO memorije



Slika 7.9: Vremenski dijagram simulacije rada FIFO memorije

7.4 ZADATAK:

Isprojektovati magazinsku memoriju tipa LIFO (STEK memorija). Memorija maksimalno može da sadrži 10 osmobitnih reći. Blok dijagram entiteta koji treba realizovati prikazuje Slika 7.10.



Slika 7.10: Blok dijagram LIFO memorije

Ulazni signali POP i PUSH predstavljaju kontrolne signale za čitanje i upis podataka respektivno. Oba signala su aktivna na visokom nivou. Ulazni vektor DATA sadrži osmobitni podatak koji se upisuje u memoriju signalom WR. Sve operacije se obavljaju na rastuću ivicu takt signala CLK. Ulazni signal RST, aktivan na viskom nivou, postavlja memoriju u inicijalno stanje. Inicijalno stanje memorije je kada je memorija potpuno ispražnjena.

Tokom operacije čitanja, izlazni vektor DATA sadrži sadržaj memorijske lokacije kojoj se pristupa. Inače, vektor je u stanju visoke impedanse. Izlazni signali EMPTY i FULL ukazuju na stanje ispražnjenosti memorije i na potpunu popunjenost memorije.

Memoriju isprojektovati pomoću VHDL jezika za opis fizičke arhitekture.

REŠENJE:

LIFO (*Last In First Out*) memorija, ili STEK (*Stack*) memorija, takođe predstavlja vrstu magazinske memorije kao i FIFO memorija. Niz bita, organizovanih kao reč, smeštaju se u memoriju po redosledu nailazka, komandom upisa u memoriju. Suprotno od FIFO memorije, operacijom čitanja se poslednja upisana reč prenosi na izlaz i odstranjuje iz memorije. Na ovaj način se uvek pristupa samo vrhu steka. Zbog toga su operacije upisa i čitanja dobile posebne nazive: uvlačenje (*push*) i izvlačenje (*pop*). Mehanizam pristupa u ovoj organizaciji takođe ne zahteva prisustvo spoljnje adrese.

LIFO memorija se karakteriše sa dva parametra:

- broj bita u reči koji se smeštaju u memoriju
- broj reči koje memorija sadrži

Stoga je zgodno ova dva parametra postaviti kao generička i omogućiti njihovo proizvoljno postavljanje prilikom instanciranja entiteta LIFO memorije koja se realizuje u ovom zadatku. U VHDL kodu koji sledi ova dva parametra su označena identifikatorima pWIDE i pDEEP.

```
LIBRARY ieee;
  USE ieee.std_logic_1164.all;
  USE ieee.numeric_std.all;

ENTITY LIFO IS
  GENERIC (
                  -- pretpostavljena sirina reci je 8 bita
           pWIDE: integer := 8;
                  -- pretpostavljen broj reci u memoriji je 10
           pDEEP: integer := 10);
  PORT (
    iCLK, iRST:    IN  std_logic;
    iPOP, iPUSH:   IN  std_logic;
    iDATA:         IN  std_logic_vector(pWIDE-1 DOWNTO 0);
    oDATA:         OUT std_logic_vector(pWIDE-1 DOWNTO 0);
    oEMPTY, oFULL: OUT std_logic );
END LIFO;
```

```
ARCHITECTURE ARH_LIFO OF LIFO IS
  -- tip koji opisuje niz lokacija koje predstavljaju LIFO memoriju
  TYPE tLIFO IS ARRAY (pDEEP-1 DOWNTO 0) OF
                      std_logic_vector(pWIDE-1 DOWNTO 0);
  -- stek memorija
  SIGNAL sLIFO: tLIFO;
  -- Stack Pointer - Pokazivac vrha steka
  -- poslednje stanje pokazivaca vrha steka ukazuje
  -- da je memorija puna
  SIGNAL sSP: integer RANGE 0 TO pDEEP;
  -- signali koji ukazuju da li je memorija prazna ili puna
  SIGNAL sEMPTY, sFULL: std_logic;
BEGIN

  -- memorija je prazna ako je pokazivac na nuli
  sEMPTY <= '1' WHEN (sSP = 0)     ELSE '0';
  -- memorija je puna ako je pokazivac na maksimalnoj vrednosti
  sFULL  <= '1' WHEN (sSP = pDEEP) ELSE '0';

  -- proces koji opisuje LIFO memoriju
  PROCESS (iCLK) BEGIN
    IF (iCLK'EVENT AND iCLK='1') THEN
      IF (iRST = '1') THEN          -- sinhroni reset
        FOR i IN sLIFO'RANGE LOOP   -- postavi sve lokacije na nulu
          sLIFO(i) <= (OTHERS => '0');
        END LOOP;
      ELSE
        IF ((iPUSH = '1') AND (sFULL = '0')) THEN
          -- upis podataka ako memorija nije puna
          -- na lokaciju gde ukazuje SP
          sLIFO(sSP) <= iDATA;
        END IF;
      END IF;
    END IF;
  END PROCESS;

  -- kontrola pokazivaca steka
  PROCESS (iCLK) BEGIN
    IF (iCLK'EVENT AND iCLK='1') THEN
      IF (iRST = '1') THEN  -- sinhroni reset
        sSP <= 0;
      ELSE
        IF ((iPOP = '1') AND (sEMPTY = '0'))  THEN
          -- citanje iz memorije -> umanjiti vrednost za jedan
          sSP <= sSP - 1;
        ELSIF ((iPUSH = '1') AND (sFULL = '0')) THEN
          -- upis u memoriju -> uvecati vrednost za jedan
          sSP <= sSP + 1;
        END IF;
      END IF;
    END IF;
  END PROCESS;

  -- odredjivanje izlaznog vektora
  PROCESS (iPOP, sEMPTY, sLIFO, sSP) BEGIN
    IF ((iPOP = '1') AND (sEMPTY = '0')) THEN
      -- citanje podataka ako memorija nije prazna
      -- podaci se citaju sa lokacije SP-1
      -- (poslednji upisan podatak)
```

```
        oDATA <= sLIFO(sSP-1);
      ELSE
        oDATA <= (OTHERS => 'Z');
      END IF;
    END PROCESS;

    -- prosledjivanje signala na izlazne portove
    oEMPTY <= sEMPTY;
    oFULL  <= sFULL;

  END ARH_LIFO;
```

Memorija je implementirana kao niz od `pDEEP` lokacija širine određene paramentrom `pWIDE`. Dati niz je predstavljen tipom `tLIFO`, a sama memorija je predstavljena signalom `sLIFO`.

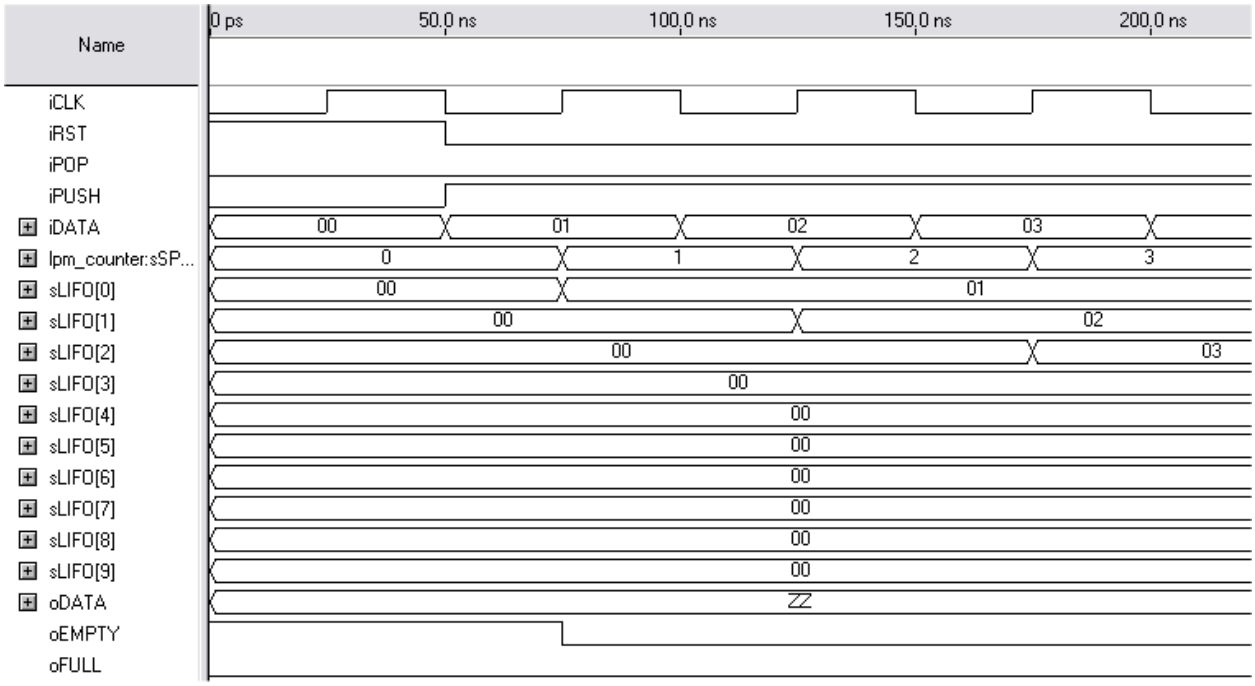
Upis podatka u memoriju se izvršava na lokaciji na koju ukazuje pokazivač `sSP` (*Stack Ponter* – pokazivač vrha steka). Stanje ovog pokazivača se automatski uvećava za jedan nakon izvršenog upisa. Takođe, stanje pokazivača se ažurira i nakon operacije čitanja smanjenjem njegove vrednosti za jedan. Dati pokazivač je realizovan tipom `INTEGER` radi direktnog omogućavanja adresiranja lokacije u implementiranom nizu memorijskih lokacija.

Čitanje poslednjeg upisanog podatka se uvek izvršava sa lokacije sa adresom za jedan manjom od vrednosti pokazivača vrha steka (`sSP-1`). Prilikom čitanja, aktiviranjem signala `iPOP`, se podatak sa ove lokacije postavlja na izlaznu magistralu podataka `oDATA`.

Potrebno je primetiti da i u ovom slučaju pokazivač `sSP` može primiti vrednost koja ne adresira nijednu lokaciju (`sSP=pDEEP`). Na ovaj način je realizovano određivanje popunjenosti stek memorije isto kao i u slučaju FIFO memorije. Sa druge strane, memorija je prazna kada je `sSP=0`. Ovakvom implementacijom se jednostavno realizuju signali koji govore o popunjenosti (`oFULL`) i ispražnjenosti (`oEMPTY`) memorije.

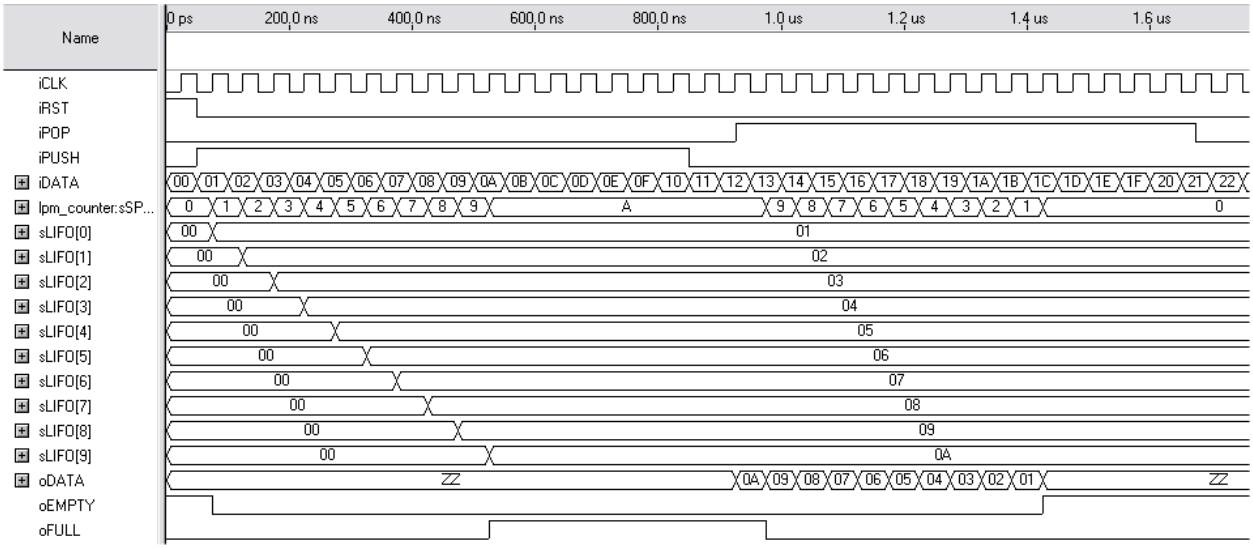
Ažuriranje stanja pokazivača `sSP`, kao i upis i čitanje iz memorije je kontrolisano sa ulaznim signalima `iPUSH` i `iPOP`, kao i sa signalima `sFULL` i `sEMPTY` koji ukazuju na nemogućnost upisa u memoriju (sve memorijske lokacije su pune) i nemogućnost čitanja iz memorije (memorija je prazna).

Slika 7.11 prikazuje vremenski dijagram upisa u LIFO memoriju tri podatka. Prvi podatak, `iDATA=01`$_{HEX}$, se upisuje na rastuću ivicu takt signala `iCLK` u vremenskom trenutku t=75ns, pošto je signal `iPUSH` postavljen na visokom logičkom nivou. U tom trenutku se vrednost pokazivača `sSP` uvećava za jedan i ukazuje na prvu sledeću slobodnu lokaciju. Upis prvog podatka se vidi i promenom stanja vektora `sLIFO`. Drugi podatak, `iDATA=02`$_{HEX}$, se upisuje na adresu 1 na rastuću ivicu takt signala `iCLK` u vremenskom trenutku t=125ns, dok se treći podatak `iDATA=03`$_{HEX}$ upisuje na lokaciju sa adresom 2 na rastuću ivicu takt signala `iCLK` u vremenskom trenutku t=175ns.



Slika 7.11: Ciklus upisa u LIFO memoriju

Simulacioni dijagram upisa u svih 10 memorijskih lokacija i čitanje upisanog sadržaja iz svih lokacija prikazuje Slika 7.12. Sa slike se vidi da su u LIFO memoriju redom upisani podaci 01_{HEX} do $0A_{HEX}$ na memorijskim lokacijama na adresama 0_{HEX} do 9_{HEX}. Upisani podaci su pročitani tokom ciklusa čitanja u obrnutom redosledu (od $0A_{HEX}$ do 01_{HEX}), što ukazuje na dobru realizaciju LIFO memorije. Tokom ciklusa čitanja se sadržaj LIFO memorije ne menja, već se samo ažurira vrednost pokazivača vrha steka `sSP`.



Slika 7.12: Vremenski dijagram simulacije rada LIFO memorije