LIFO struktura (last in - first out buffer)

V VHDL programirajte arhitekturo splošne strukture strojnega sklada - LIFO (ang. last in - first out) strukture.

Entiteta izdelane strukture `lifo` ima priključke:

```
entity lifo is
           generic(  lifo_width : natural := 4;  -- velikost podatka sklada
                     lifo_size  : natural := 8);  -- velikost sklada
           PORT (    clk,       -- signal ure
                     nCLR,      -- signal asinhronega brisanja vsebine sklada (aktiven '0')
                     nEnable,   -- signal omogočanja sklada (aktiven '0', sicer drži vsebino)
                     PUSH,      -- operacija vpisa na sklad (aktiven '1')
                     POP        : IN std_logic; -- operacija branja s sklada (aktiven '1')
                     data       : inout std_logic_vector(lifo_width - 1 downto 0);    --tristanjski izhod sklada
                     FULL,                          -- izhod, ki postane '1', ko je sklad poln
                     EMPTY     : OUT std_logic      -- izhod, ki postane '0', ko je sklad prazen
                );
end lifo;
```

Naloge:

1. Ustvarite datoteko **tff.vhd** v kateri boste programirali entiteto in arhitekturo T flip-flopa.
2. V ustvarjeno datoteko kopirajte entiteto in arhitekturo D flip-flopa (**dff**) iz prejšnje domače naloge. Ime entitete mora biti: **tff**. Podana je entiteta strukture:

```
ENTITY tff IS
PORT (     T,                          -- vhod flip-flopa
           clk,                        -- signal ure
           nPRESET,                    -- asinhrono postavljanje izhoda (aktiven '0')
           nCLEAR : IN STD_LOGIC;      -- asinhrono brisanje izhoda (aktiven '0')
           Q : OUT STD_LOGIC);         -- izhod flip-flopa
END tff;
```

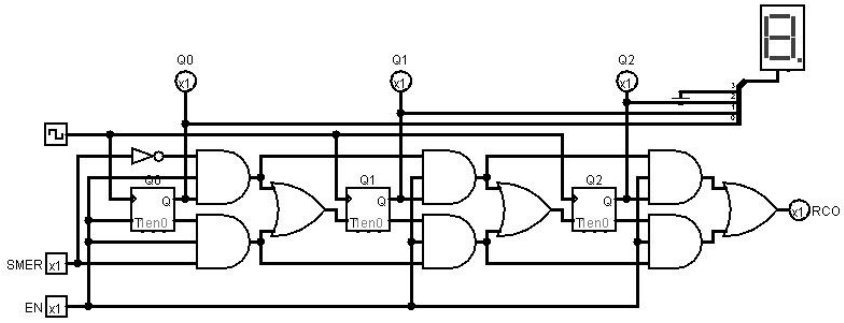
   Delovanje D flip flopa predelajte tako, da bo deloval kot T flip flop - izhod zamenja stanje (**Q <= not Q**), ko je T='1'.

   Poleg osnovnega delovanja mora imeti še signala za asinhrono postavljanje (**nPRESET**) in brisanje (**nCLEAR**) izhoda flip-flopa (**Q**).

   Asinhrono pomeni, da se izhod postavlja in briše neodvisno od signala ure (**clk**). Za realizacijo asinhronih signalov uporabite (**if...elsif...else**) stavek.
3. Izdelajte datoteko testnih vrednosti (**tff_tb.tbw**) in s simulacijo preverite pravilnost delovanja za vpis podatka '0' in '1' ter asinhrono postavljanje in brisanje izhoda (izhod se mora postaviti/brisati asinhrono - recimo ob spremembi na zadnji rob signala ure).
4. Ustvarite datoteko **ud_counter.vhd** v kateri programirajte entiteto in arhitekturo dvosmernega dvojiškega števca. Ime entitete mora biti: **ud_counter**. Podana je entiteta strukture:

```
entity ud_counter is
           generic( ctr_size: natural := 6);
           PORT (    clk,                              -- signal ure
                     nCLR,                             -- signal za brisanje števca (aktiven '0')
                     D_nU,                             -- signal za smer štetja števca (narašèajoèe -> '0')
                     EN   : IN std_logic;      -- signal za omogoèanje štetja (aktiven '1')
                     RCO : out std_logic; -- izhodni prenos na naslednjo stopnjo (Ripple carry out)
                     Q : out std_logic_vector(ctr_size - 1 downto 0) -- izhodno štetje števca
                );
end ud_counter;
```

Velikost števca je (**ctr_size**). Števec se *asinhrono* zbriše (**Q => (others = '0')**), ko postane signal (**nCLR='0'**). Smer štetja definira signal (**D_nU**). Če je signal (**D_nU = '0'**), števec šteje naraščajoče, sicer padajoče. Slika 1 prikazuje izvedbo 3-bitnega dvosmernega števca. Na sliki signal SMER določa smer štetja. Ko je (SMER = '0') je štetje naraščajoče. Podano strukturo 3-bitnega števca opišite v VHDL z uporabo prej izdelanih T flip-flopov (**tff.vhd**), ki jih povežete z uporabo povezovalnega stavka znotraj **for ... generate** zanke. Povezave verige zgornjih in spodnjih AND vrat izvedite znotraj **for ... generate** zanke. Izhodni signal števca RCO postane '1', ko je vrednost števca maksimalna in smer štetja naraščajoča. V primeru na sliki postane signal enak '1', ko je stanje števca "111" in smer štetja naraščajoča. Podobno postane RCO enak '1', ko je vrednost števca minimalna "000" in smer štetja padajoča. Števec ima signal za omogočanje štetja (EN). Ko je štetje omogočeno (EN='1'), bo števec štel, sicer ohranja trenutno stanje. Asinhrono brisanje števca je izvedeno s signalom za asinhrono brisanje T flip flopov (ni narisano). V VHDL ima ta signal ime (**nCLR**).



Slika 1: Dvosmerni števec s T flip−flopi.

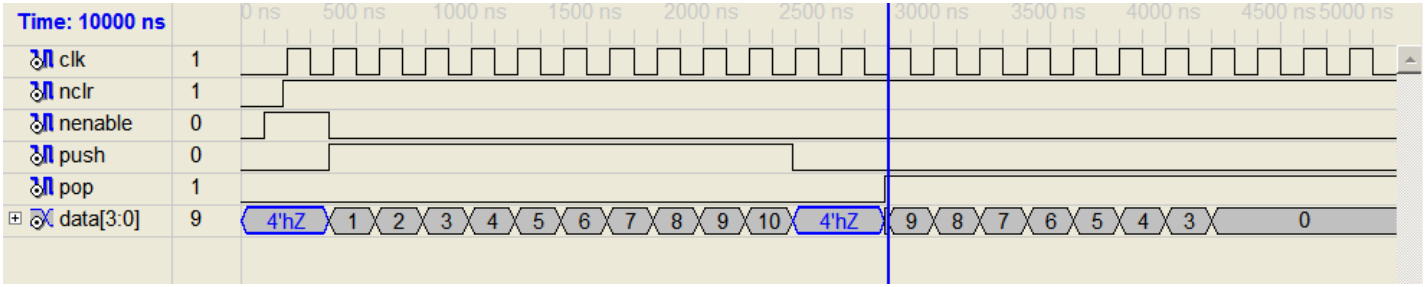
5. Izdelajte datoteko testnih vrednosti (**ud_counter_tb.tbw**) in s simulacijo preverite pravilnost delovanja števca za zgoraj opisane signale.
6. Datoteke **muxnto1.vhd**, **muxdff.vhd** in **shift_reg.vhd** kopirajte iz prejšnje domače naloge (FIFO) v imenik naloge.
7. Podobno kot pri prejšnji domači nalogi (FIFO) povežite **lifo_size** komponent pomikalnega registra z uporabo povezovalnega stavka (**PORT MAP**).
   Vhode pomikalnih registrov povežite podobno, kot je prikazano na prosojnicah predavanj (prosojnica 288). Pri povezovanju s **PORT MAP** definirajte vmesni signal, katerega *tip* je dvodimenzionalno polje:
   ```
   type shift_reg_array_type is array (lifo_width - 1 downto 0) of std_logic_vector(lifo_size - 1 downto 0);
   ```
   Opisani *tip signala* predstavlja dvodimenzionalno polje velikosti **lifo_width** **lifo_size** bitnih registrov.
   Za krmiljenje načina delovanja pomikalnih registrov vpeljite interni vektorski signal (**s**), ki mu z **WITH ...SELECT** prekodirnikom določate vrednost. Vhodni

signal tega prekodirnika (**lifo_mode**) sestavite v vektor s pomočjo operatorja lepljenja (**&**).

Sklad deluje tako, da operacija vpisa (**PUSH='1'**) pomeni pomikanje vsebine pomikalnih registrov **LEVO**, operacija branja podatka s sklada (**POP='1'**) pomeni pomikanje **DESNO**, sicer sklad vsebino ohranja.

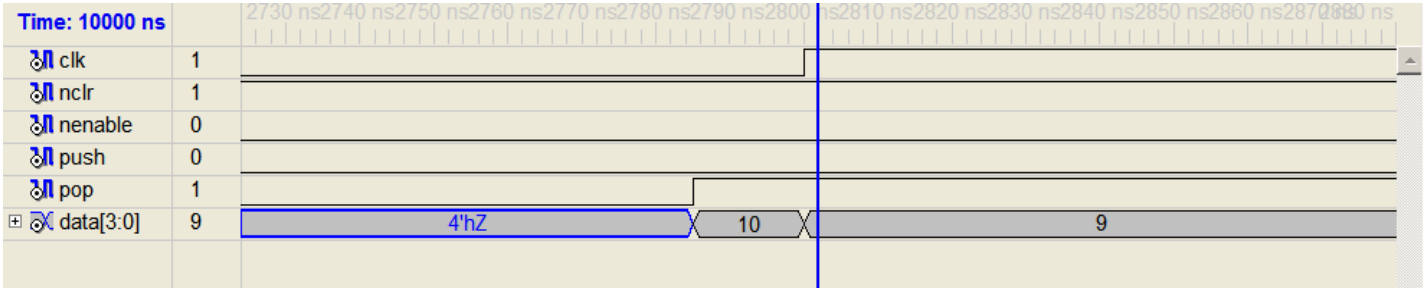
Vmesni izhod sklada (**lifo_out**) povežite na LSB mesta pomikalnih registrov. Tristanjski izhod sklada izdelajte z uporabo **WHEN ...ELSE** izbiralnika. Tristanjski izhod (**data**) postane vmesni izhod (**lifo_out**), če gre za operacijo branja sklada, sicer je njegova vrednost splošni vektor visoke impedance (**others => 'Z'**).

8. Izdelajte datoteko testnih vrednosti (**lifo_tb.tbw**) in s simulacijo preverite pravilnost delovanja vseh operacij sklada. Slika 2 prikazuje potek krmilnih signalov datoteke testnih vrednosti in rezultat simulacije delovanja LIFO strukture. Preveriti morate delovanje brisanja, (prvi cikel ure), delovanje onemogočanja (drugi cikel ure), delovanje vpisovanja (PUSH) več vrednosti, kot je velikost sklada in branje (POP) več vrednosti kot je velikost sklada.



Slika 2: Rezultat simulacije osnovnega delovanja sklada.

Iz rezultata simulacije sledi, da se ob branju *zadnji* vpisan podatek *takoj* pojavi na izhodu sklada (**data**). Torej bo ob prvem ciklu signala ure po aktiviranju operacije branja (**POP => _/¯**) na izhodu že *predzadnji* vpisan podatek. Slika 3, ki je povečava slike 2 na mestu modrega kurzorja, prikazuje potek branja prvega podatka in srž problema.



Slika 3: Izsek branja prvega podatka s sklada po prehodu signala branja v aktivno stanje.

Enota, ki uporablja izdelani sklad, tako ne more prebrati nazadnje vpisanega podatka, saj pričakuje prvi podatek ob aktivnem robu signala ure. Zato je potrebno zadnji vpisan podatek *zadržati do prvega prehoda signala ure po aktiviranju POP signala*. Temu strokovno pravimo *dvojno pomnjenje* (ang. "*double buffering*"). Za izvedbo zadrževanja (pomnjenja) vsebine zadnje vpisanega podatka potrebujemo shranjevalni register, sestavljen iz komponent D flip flopov (**`dff.vhd`**).

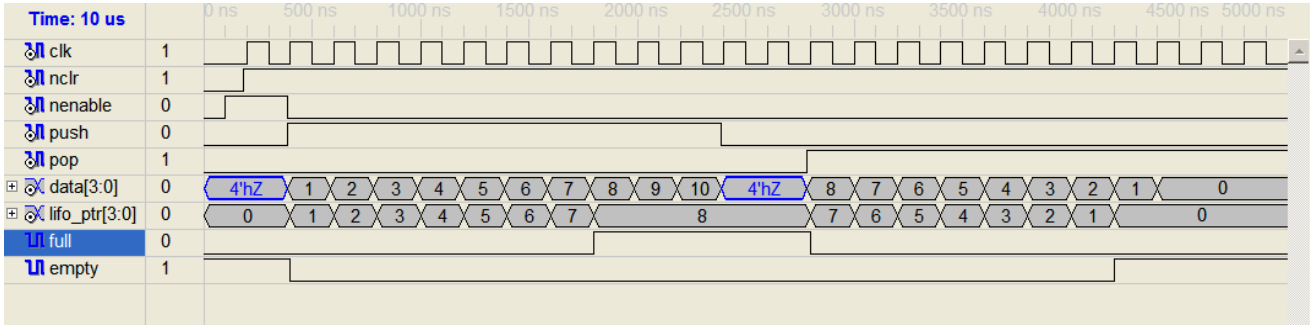
```
BUFi: dff port map ( Q(i)(0), clk, '1', nCLR, lifo_out(i));
```

Deklariramo ga znotraj `for ... generate` zanke, s katero smo izvedli povezovanje pomikalnih registrov sklada.

9. Izdelani strukturi sklada dodajte skladovni števec kot komponento prej izdelanega dvosmernega dvojiškega števca (**`ud_counter.vhd`**). Skladovni števec šteje število vpisanih podatkov na sklad. Za povezovanje moramo število bitov števca izraziti z velikostjo sklada (**`lifo_size`**), zato moramo določiti koliko bitov zahteva dvojiški zapis števila (**`lifo_size + 1`**):

   ```
   constant ud_ctr_size : integer := integer(ceil(log2(real(lifo_size + 1))));
   ```

   Število stanj skladovnega števca je namreč (**`lifo_size + 1`**), saj stanje "sklad prazen" zahteva svojo kodo (vpisanih nič podatkov). Za uporabo funkcij (**`ceil`**, **`log2`**) moramo v knjižnice vključiti realna števila (`use IEEE.MATH_REAL.ALL`). Pri povezovanju števca definirajte 2-bitni vmesni vektorski signal (**`ud_ctr_mode`**), ki mu z `WITH ...SELECT` prekodirnikom določate vrednost. Vhodni signal tega prekodirnika naj bo enak (**`lifo_mode`**).
10. Dodajte izhodna signala detekcije stanja sklada FULL in EMPTY. Če je izhodna vrednost skladovnega števca enaka (**`lifo_size`**), postane FULL = '1'. Dokler je FULL = '1' je vpisovanje na sklad onemogočeno, da ne prepišemo vsebine. Signal EMPTY postane '1' ko je sklad prazen - stanje skladovnega števca je enako nič. Branje praznega sklada ni mogoče.
11. Izdelajte datoteko testnih vrednosti (**`lifo_tb.tbw`**) in s simulacijo preverite pravilnost delovanja 4-bitne LIFO strukture velikosti 8 zlogov (ang. byte).



Slika 4: Delovanje dokončane LIFO strukture.

Slika 4 prikazuje potek krmilnih signalov datoteke testnih vrednosti in rezultat simulacije delovanja 4-bitne LIFO strukture velikosti 8 zlogov. Preveriti morate:

- delovanje brisanja (prvi cikel ure),
- delovanje onemogočanja (drugi cikel ure),
- vpisovanje (PUSH) več vrednosti, kot je velikost sklada in
- branje (POP) več vrednosti kot je velikost sklada.

Sklad postavi signal FULL='1', ko se vpiše osmi podatek. Ob vpisu devetega in desetega podatka (9_{10} in 10_{10} na sliki) sklad ti vrednosti ignorira. Signal FULL postane '0', čim s sklada preberemo en podatek. Signal EMPTY postane '1', ko s sklada preberemo osmi podatek. Če je sklad prazen, POP pa ostaja na '1', se vsebina skladovnega števca ne spreminja več.