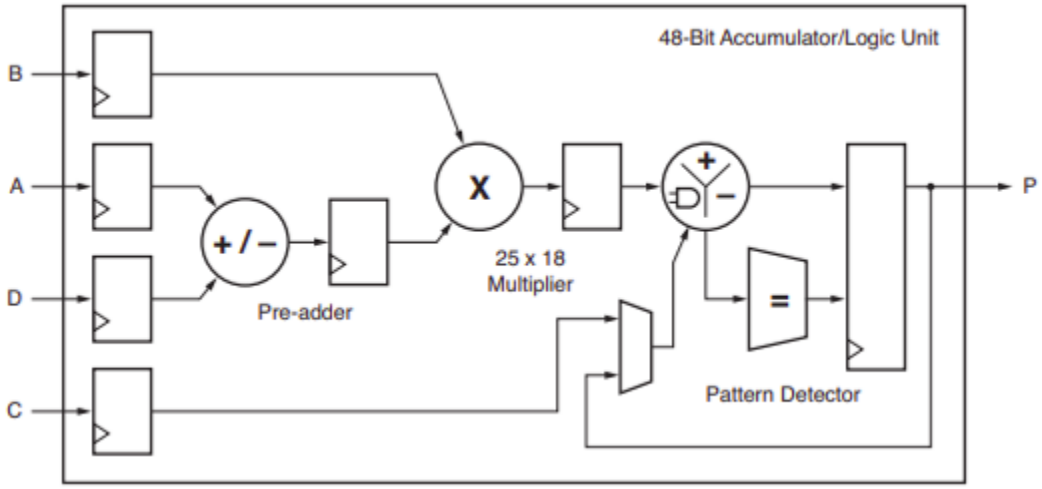
Mapiranje VHDL koda na DSP48E1 ćelije

1. Opis DSP48E1 ćelija

FPGA platforme su efikasne za digitalno procesiranje signala (*eng. Digital signal processing, eg. DSP*) zato što na njima mogu da se implementiraju algoritmi koji mogu u potpunosti da se paralelizuju. Za takvo procesiranje uglavnom se koriste množači, sabirači, oduzimači, akumulatori i oni se najbolje implementiraju pomoću već postojećih DSP ćelija. Sve FPGA platforme, koje pripadaju sedmoj seriji Xilinx platformi (*eng. 7 series*), imaju određeni broj ovih DSP ćelija, koje su *"low-power"*, a imaju visoku frekvenciju rada. Na sledećoj slici su ilustrovane osnovne funkcionalnosti jedne DSP ćelije:



Slika 1. Osnovne funkcionalnosti DSP ćelije

Osnovni blokovi jedne DSP48E1 ćelije su:

- 25 x 18 množač, koji omogućava označeno (*eng. signed*) množenje, u komplementu dvojke, operanada "A" i "B".
- ALU (*eng. Arithmetic logic unit*), koji omogućava 48-bitno sabiranje, oduzimanje i izvršavanje određenih logičkih operacija. Na ovoj jedinici su omogućene i SIMD (*eng. Single Instruction Multiple Data*) operacije, odnosno mogu da se izvrše istovremeno dva 24-bitna ili četiri 12-bitna sabiranja/oduzimanja.
- "Pre-adder" modul. Omogućava sabiranje/oduzimanje pre množenja.
- "Pattern detector" modul. Omogućava proveru da li se rezultat ALU jedinice poklapa sa nekom predefinisanom vrednošću.
- Opcioni registri, koji omogućavaju uvođenje protočne obrade prilikom digitalnog procesiranja signala. Preporuka je koristiti ih kako bi se postigle maksimalne performanse.

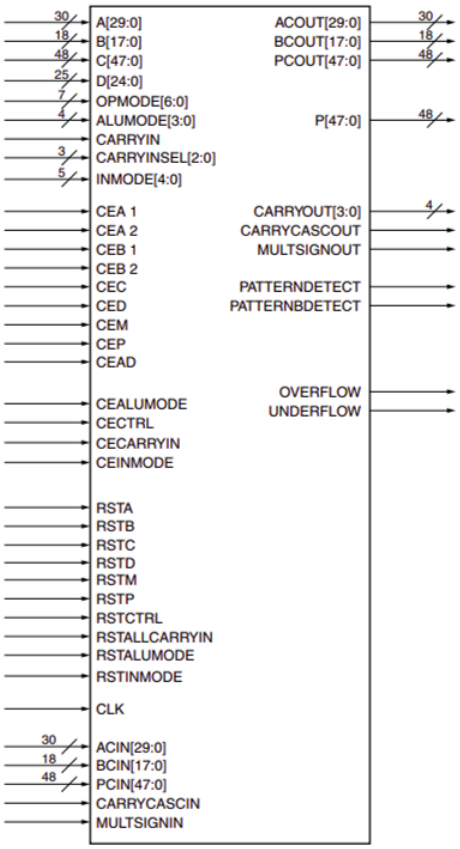
Za detaljnije informacije o DSP48E1 ćelijama pogledati:
https://www.xilinx.com/support/documentation/user_guides/ug479_7Series_DSP48E1.pdf

2. Mapiranje VHDL koda na DSP ćelije

DSP ćelije koje postoje na FPGA platformama mogu da se koriste na dva načina (pri čemu će samo drugi biti objašnjen u ovoj skripti):

1) Direktnim instanciranjem i ručnim konfigurisanjem pojedinačne DSP ćelije kako bi ona obavljala željenu funkcionalnost.
2) Opisom digitalnog sistema na bihevijalnom nivou, pri čemu se prepušta alatu da prepozna koje delove koda treba mapirati na DSP ćelije.

Prvi način omogućava veću kontrolu, jer se ne dopušta alatu koji vrši sintezu digitalnog sistema da pretpostavi šta treba da uradi. No, ta veća kontrola dolazi uz posledice, jer je neophodno utrošiti mnogo vremena na ručno instanciranje i konfigurisanje DSP ćelija. Na sledećoj slici je ilustrovan interfejs jedne DSP48E1 ćelije:



Slika 2. Interfejs DSP48E1 ćelije

Sa slike se može videti da je njen interfejs jako kompleksan, odnosno direktno instanciranje DSP ćelije i njeno konfigurisanje bi oduzimalo previše vremena. Takođe, direktno instanciranje zahteva posedovanje biblioteka koje poseduju hardverski opis DSP ćelija, što može da dovede do toga da u nekim simulatorima nije moguće izvršiti kompajliranje opisanih digitalnih sistema.

U ovoj skripti se neće zalaziti u direktno instanciranje DSP ćelija, te stoga neće biti objašnjavani pojedinačni portovi DSP interfejsa, kao ni način konfigurisanja DSP ćelije.

Drugi način korišćenja DSP ćelija zasniva se na tome da alat sam, na osnovu opisa sistema na bihevijalnom nivou, shvati koje delove treba da mapira na DSP ćelije, a koje ne. U nastavku će biti objašnjene implementacije nekih jednostavnih modula, kao što su: sabirač, oduzimač, množač, akumulator. Da bi se oni implementirali neophodno je razumeti strukturu DSP ćelija prikazanu na slici 1, odnosno treba pisati kod tako da se u njemu koriste samo komponente koje postoje unutar DSP ćelija. Naredni kodni listing je implementacija jednostavnog sabirača i njegovo mapiranje pomoću DSP ćelije:

```
entity adder is
    generic (WIDTHA:natural:=32;
             WIDTHB:natural:=32;
             SIGNED_UNSIGNED: string:= "unsigned");
    Port (a_i: in std_logic_vector (WIDTHA - 1 downto 0);
          b_i: in std_logic_vector (WIDTHB - 1 downto 0);
          res_o: out std_logic_vector(WIDTHA - 1 downto 0));
end adder;

architecture Behavioral of adder is

---------------------------------------------------------------------------------
--------------------
    -- Atributes that need to be defined so Vivado synthesizer maps appropriate
    -- code to DSP cells
    attribute use_dsp : string;
    attribute use_dsp of Behavioral : architecture is "yes";

---------------------------------------------------------------------------------
--------------------
begin
    signed_add: if SIGNED_UNSIGNED = "signed" generate
                res_o <= std_logic_vector(signed(a_i) + signed(b_i));
    end generate;
    unsigned_add: if SIGNED_UNSIGNED = "unsigned" generate
                res_o <= std_logic_vector(unsigned(a_i) + unsigned(b_i));
    end generate;
end Behavioral;
```

Kodni listing 1. Implementacija sabirača

Prethodni kodni listing implementira sabirač čiji su ulazni operandi široki WIDTHA, odnosno WIDTHB. Da bi se implementiralo sabiranje u kodu je korišćen operator plus i da bi se taj operator plus mapirao na ALU jedinicu DSP ćelije neophodno je definisati atribut "*use_dsp*" i postaviti njegovu vrednost na "yes" (kao u prethodnom kodnom listingu). Ukoliko se to ne uradi, prepušta se alatu da sam odluči da li će mapirati određeni operator na DSP ćeliju ili ne (alat uglavnom pokušava da mapira operatore na DSP ćelije).

Sabirač implementiran u prethodnom primeru implementiran je kao kombinaciona logika, ali on može da poseduje i nekoliko faza protočne obrade, odnosno da se sabiranje vrši nekoliko taktova (radi povećanja performansi). Sa slike 1 se može videti da se unutar DSP ćelije nalazi određeni broj registara i prilikom sinteze alat može da konfiguriše DSP ćelije tako da se koriste ti registre. U nastavku je prikazan sabirač opisan na bihevijalnom nivou koji poseduje dve faze protočne obrade:

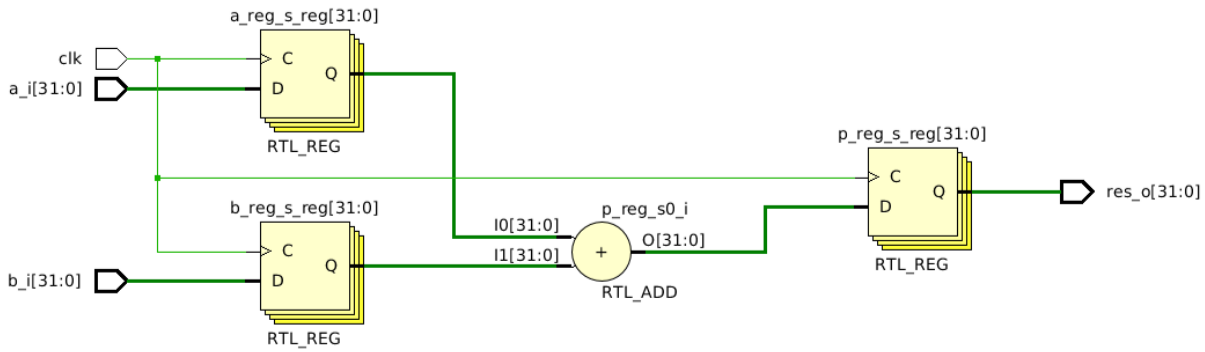
```
entity pipelined_adder is
    generic (WIDTHA:natural:=32;
             WIDTHB:natural:=32;
             SIGNED_UNSIGNED: string:= "unsigned");
    Port ( clk: in std_logic;
           a_i: in std_logic_vector (WIDTHA - 1 downto 0);
           b_i: in std_logic_vector (WIDTHB - 1 downto 0);
           res_o: out std_logic_vector(WIDTHA + WIDTHB - 1 downto 0));
end pipelined_adder;
architecture Behavioral of pipelined_adder is

------------------------------------------------------------------------------------
---------------------
    -- Atributes that need to be defined so Vivado synthesizer maps appropriate
    -- code to DSP cells
    attribute use_dsp : string;
    attribute use_dsp of Behavioral : architecture is "yes";

------------------------------------------------------------------------------------
---------------------

------------------------------------------------------------------------------------
---------------------
    -- Pipeline registers.
    signal a_reg_s: std_logic_vector(WIDTHA - 1 downto 0);
    signal b_reg_s: std_logic_vector(WIDTHB - 1 downto 0);

    signal p_reg_s: std_logic_vector(WIDTHA + WIDTHB downto 0);

------------------------------------------------------------------------------------
```

```
---------------------
begin
     process (clk) is
     begin
         if (rising_edge(clk))then
             a_reg_s <= a_i;
             b_reg_s <= b_i;
             if (SIGNED_UNSIGNED = "signed") then
                 p_reg_s <= std_logic_vector(signed(a_reg_s) + signed(b_reg_s));
             else
                 p_reg_s <= std_logic_vector(unsigned(a_reg_s) +
unsigned(b_reg_s));
             end if;
         end if;
     end process;
     res_o <= p_reg_s;
end Behavioral;
```

Kodni listing 2. Implementacija sabirača sa 2 nivoa protočne obrade

Sa slike 1 može se videti da DSP ćelija poseduje 3 faze registara. Prva faza su registri u koje se upisuju ulazni operandi (A, B, C, D), druga faza je registar u koji se upisuje rezultat množenja i u treća faza je registar u koji se upisuje rezultat ALU jedinice. Ukoliko se implementira sabirač, mogu da se koriste registri iz prve i treće faze, odnosno može da se implementira sabirač sa 3 nivoa protočne obrade (što nije neophodno, dovoljna je jedna faza). 3 nivoa protočne obrade su moguća jer prva faza registara može da se konfiguriše tako da ima 2 nivoa registara.

Da bi alat ispravno konfigurisao DSP ćeliju prilikom sinteze, tako da koristi određeni broj registara, neophodno je na ispravan način opisati sistem na bihevijalnom nivou. U prethodnom primeru, ulazni operandi su prvo upisani u registre *a_reg_s* i *b_reg_s* (kao da se koriste registri iz prve faze DSP ćelije), a u narednom taktu je rezultat sabiranja upisan u p_reg_s registar. Odnosno neophodno je opisati digitalni sistem sa sledeće slike:



Slika 3.Sabirač sa dva nivoa protočne obrade.

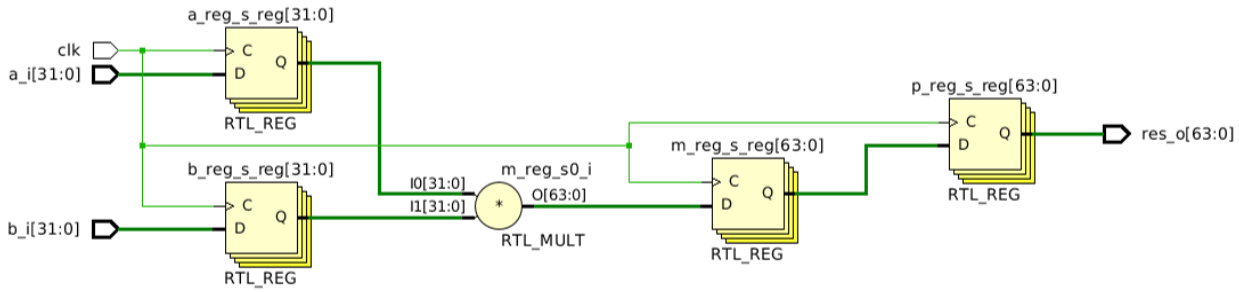
Slično kao što je implementiran sabirač, može da se implementira i množač, samo što se umesto operatora “+”, prilikom opisa dizajna na bihevijalnom nivou, korisisti operator “*”. Takođe, može da se menja broj faza protočne obrade i preporuka kompanije Xilinx, radi maksimalnih performansi je da se koristi množač sa 3 nivoa protočne obrade, ilustrovan u sledećem primeru:

```
entity simple_multiplier is
    generic (WIDTHA:natural:=32;
             WIDTHB:natural:=32;
             SIGNED_UNSIGNED: string:= "unsigned");
    Port ( clk: in std_logic;
           a_i: in std_logic_vector (WIDTHA - 1 downto 0);
           b_i: in std_logic_vector (WIDTHB - 1 downto 0);
           res_o: out std_logic_vector(WIDTHA + WIDTHB - 1 downto 0));

end simple_multiplier;

architecture Behavioral of simple_multiplier is

------------------------------------------------------------------------------------
--------------------
    -- Atributes that need to be defined so Vivado synthesizer maps appropriate
    -- code to DSP cells
    attribute use_dsp : string;
    attribute use_dsp of Behavioral : architecture is "yes";

------------------------------------------------------------------------------------
--------------------



------------------------------------------------------------------------------------
--------------------
    -- Pipeline registers.
    signal a_reg_s: std_logic_vector(WIDTHA - 1 downto 0);
    signal b_reg_s: std_logic_vector(WIDTHB - 1 downto 0);

    signal m_reg_s: std_logic_vector(WIDTHA + WIDTHB - 1 downto 0);
    signal p_reg_s: std_logic_vector(WIDTHA + WIDTHB downto 0);

------------------------------------------------------------------------------------
--------------------
begin
     process (clk) is
     begin
         if (rising_edge(clk))then
```

```
            a_reg_s <= a_i;
            b_reg_s <= b_i;
            if (SIGNED_UNSIGNED = "signed") then
                m_reg_s <= std_logic_vector(signed(a_reg_s) * signed(b_reg_s));
                p_reg_s <= m_reg_s;
            else
                m_reg_s <= std_logic_vector(unsigned(a_reg_s) *
unsigned(b_reg_s));
                p_reg_s <= m_reg_s;
            end if;
        end if;
    end process;
    res_o <= p_reg_s;
end Behavioral;
```

Kodni listing 3. Implementacija množača sa 3 nivoa protočne obrade

Da bi se implementirao množač sa 3 nivoa protočne obrade, prvo se operandi “a_i” i “b_i” upisuju u registre “*a_reg*” i “*b_reg*”, u narednom taktu se vrši množenje vrednosti upisanih u te registre i rezultat se upisuje u “*m_reg*”. U narednom taktu se “*m_reg*” upisuje u “*p_reg*” (to je neophodno uraditi jer se rezultat množenja mora propustiti kroz ALU jedinicu DSP ćelije, što će alat interno da uradi). Prethodni kodni listing mapira sledeći digitalni sistem na DSP ćeliju:

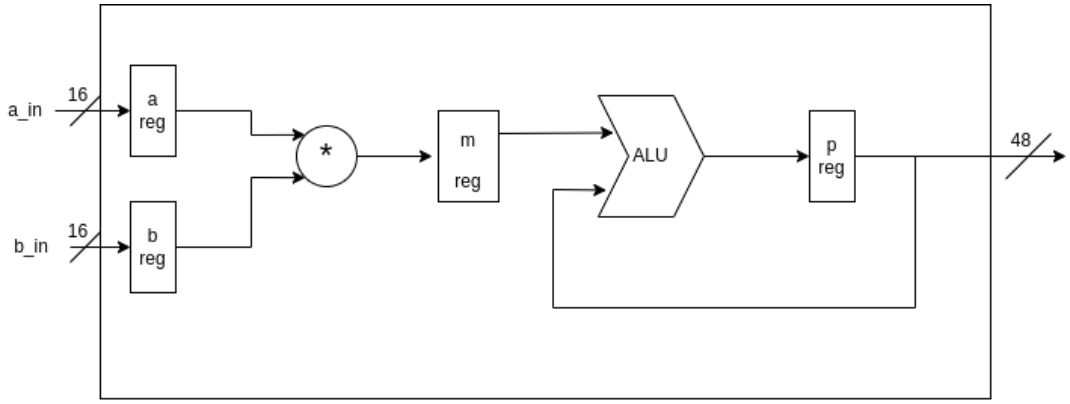


Slika 4. Množač sa 3 nivoa protočne obrade.

Napomena: u zavisnosti od širine operanada koje treba pomnožiiti alat će da mapira operacije na jednu ili više DSP ćelija. Na primer, prilikom implementacije množača, ukoliko je WIDTHA <= 25 i WIDTHB <= 18 alat će mapirati množač na samo jednu DSP ćeliju, u suprotnom će iskoristiti više DSP ćelije kao i dodatnu logiku kako bi implementirao množač.

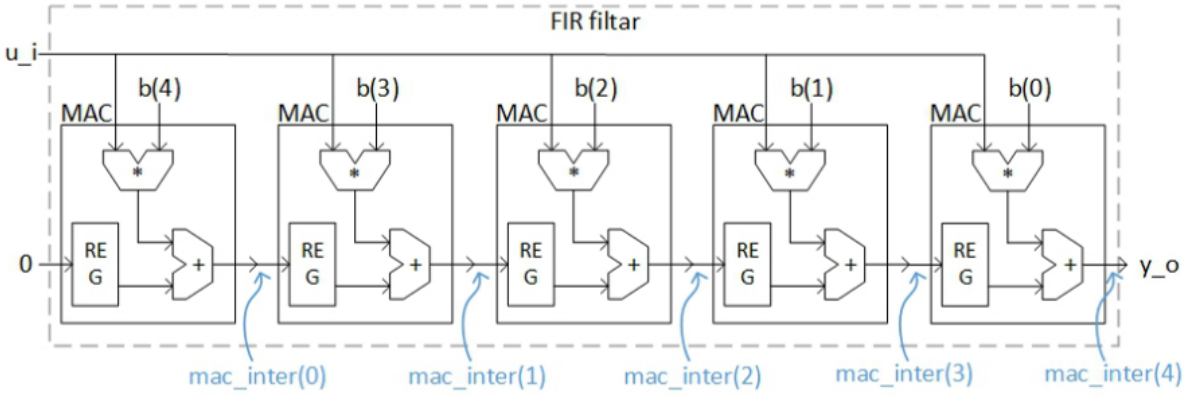
Zadaci:

1. Analizirati utrošenost resursa množača 32 x 32 implementiranog pomoću DSP jedinica. Izvršiti sintezu i analizirati izveštaj alata).
2. Realizovati *multiply_accumulate* (MAC) modul prikazan na sledećoj slici pomoću DSP jedinica i analizirati utrošenost resursa.



Slika 5. MAC modul

3. Realizovati FIR filter opisan u materijalu za laboratorijsku vežbu 8 na predmetu Diskretni sistemi (Hardverska implementacija FIR LVN DS) koristeći DSP ćelije.
    - Pokazati u simulaciji da implementirani sistem funkcioniše.
    - Analizirati utrošenost resursa.
    - Izvršiti statičku vremensku analizu.



Slika 6. FIR filtar.

Hardverska otpornost na otkaz

Zadaci:

1. Koristeći DSP jedinice implementirati MAC modul koji poseduje trostruku modularnu redundansu (Triple modular redundancy). **Pogledati Predavanje 4 - Hardware Fault Tolerance Techniques 1, strana 16.**
    a. Analizirati utrošenost resursa i izvršiti statičku vremensku analizu implementiranog sistema.
    b. Pokazati u simulaciji da implementirani sistem funkcioniše.
2. Koristeći DSP jedinice implementirati MAC modul koji poseduje N modularnu redundansu (N-modular redundancy), gde je N maksimalno 80. **Pogledati Predavanje 4 - Hardware Fault Tolerance Techniques 1, strana 35.**
    a. Analizirati utrošenost resursa i izvršiti statičku vremensku analizu implementiranog sistema za različite vrednosti N. Sistem je neophodno da funkcioniše na barem 150 Mhz, ukoliko statička vremenska analiza pada na toj frekvenciji implementirati optimalniji sistem.
    b. Pokazati u simulaciji da implementirani sistem funkcioniše za različite vrednosti N.
3. Za zadatak 3 (implementacija FIR filtra) iz prethodne sekcije uvesti trostruku modulatnu redundansu za svaki od MAC modula.
    - Analizirati utrošenost resursa i izvršiti statičku vremensku analizu implementiranog sistema.
    - Pokazati u simulaciji da implementirani sistem funkcioniše.