Izrada demonstracijskog 8-bitnog procesora

Studenti: Marija Mijić, Vedran Vukotić

Nastavnik: prof. dr.sc. Siniša Šegvić

Predmet: Arhitektura računala 2

Sadržaj

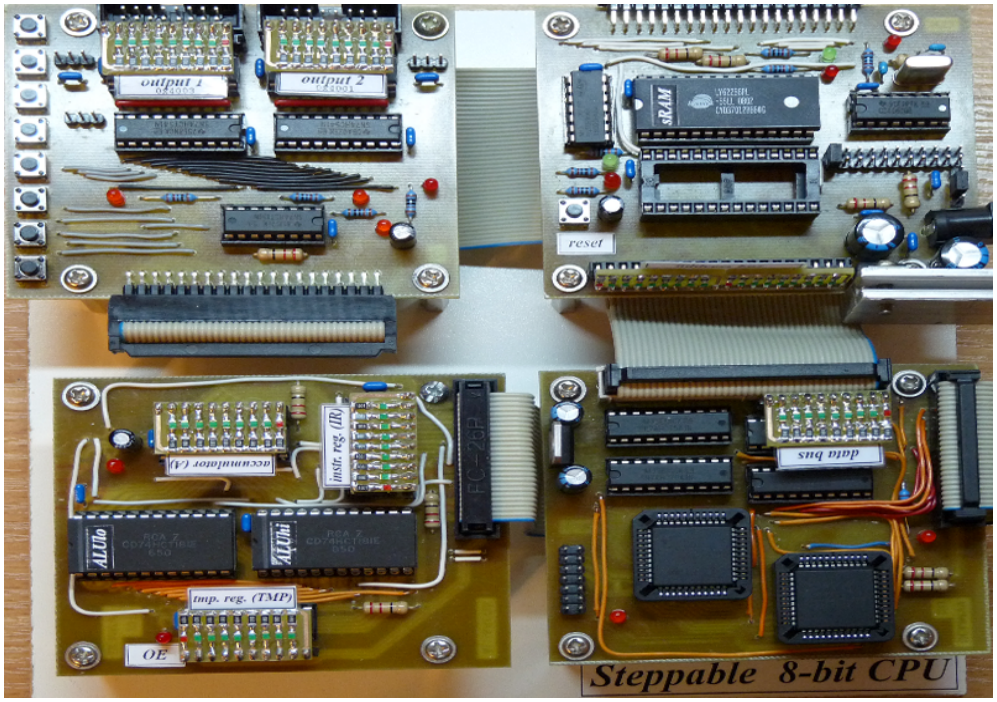
Uvod

Cilj ovog projekta bio je razviti jednostavan 8-bitni procesor od diskretnih elektroničkih elemenata koji bi služio u ilustrativne svrhe. Procesor je zamišljen tako da ima indikatore (LED) koji pokazuju stanje sabirnica (adresne i podatkovne), bitnih registara (instrukcijski registar, akumulator, privremeni registar aritmetičko-logičke jedinice te izlazni registri I/O pločice) te važnijih kontrolnih signala. Također, u svrsi što boljeg predočavanja samog rada procesora omogućeno je ručno generiranje *clock* signala čime se ostvaruje mogućnost izvršavanja svakog pojedinog koraka i proučavanje pojedinih faza procesora. Naravno, osim ručno kontroliranog *clock* signala, tu je i mogućnost da se procesor pogoni automatski generiranim taktom (*quartz* oscilator uz djelitelj takta) raznih frekvencija koje se mogu odabrati prikladnim *jumperom*. Pojedini elementi procesora su uglavnom iz 74xx serije: ALU je sastavljen od dva 4-bitna ALU-a (74HCT181), pojedini registri su 74HC574, a adresni dekoderi 74HCT139. Obzirom da bi izrada procesora od isključivo takvih elemenata bila prezahtjevna, korištena su i dva CPLD-a (XC9572XL). U prvom CPLD-u su sintetizirani programsko brojilo (PC), registar pokazivač (PT) te sva potrebna logika koja upravlja adresnom sabirnicom. Svaki je element modeliran zasebno te povezan signalima unutar CPLD-a kao da se radi o zasebnim implementacijskim elementima. Na drugom CPLD-u se nalazi isključivo logika upravljačke jedinice procesora.

Po pitanju memorije, adresni prostor je podijeljen u tri dijela: ROM, RAM i I/O prostor te se ukupno prostire kroz 64kB. Smisao ROM-a i RAM-a je sam po sebi jasan, a I/O se sastoji od dvaju ulaznih međuspremnika i dvaju izlaznih registara, svaki od po osam bita. Na izlazne registre je moguće spojiti indikatorske ledice ili drugu periferiju (primjerice LCD).

Sama fizička izrada procesora i periferije se sastoji od četiriju pločica (PCB). Na jednoj su smještena dva CPLD-a (kontrolna jedinica te logika koja upravlja adresnom sabirnicom), na drugoj su smješteni ALU i registri (IR, A i TMP), na trećoj su memorija (RAM i ROM), dio adresnih dekodera te generator takta. Konačno, na četvrtoj pločici se nalaze ulazni međuspremnici i izlazni registri, drugi dio adresnih dekodera te priključci za moguću periferiju. Izgled gotovog procesora prikazan je na slici 1.

Za potrebe simulacije izrađeni su i pomoćni VHDL modeli pojedinih bitnih elemenata koji su korišteni. Cjelokupni procesor je najprije simuliran u VHDL-u korištenjem Xilinx ISE alata te je zatim napravljena njegova fizička realizacija istoga.

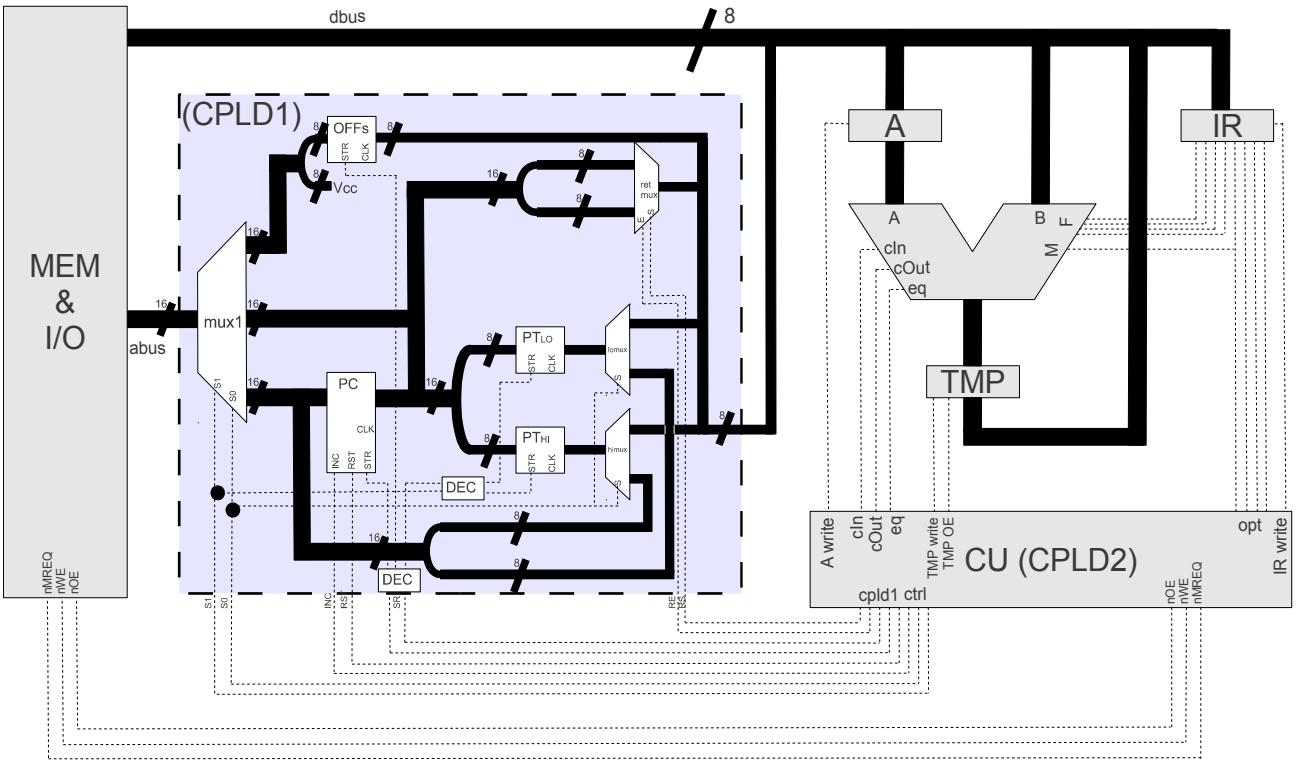


Slika 1: Izgled procesora i periferije

1. Organizacija demonstracijskog procesora

Procesor je akumulatorske strukture. Prilikom izvođenja ALU operacija, jedan se podatak dovodi u akumulator (čiji je izlaz spojen na jedan ulaz ALU-a) dok se drugi dovodi direktno iz podatkovne sabirnice (dbus) u drugi ulaz ALU-a. Akumulator je jedini fizički(pravi) registar koji je dostupan programeru. (Programer ima na raspolaganje 256 memorijskih lokacija u sRAM-u na najvišoj stranici kojima može brzo pristupiti. Detaljnije o ovome kasnije.) Na izlazu ALU-a se nalazi TMP registar. Naziv TMP (*temporary*) slijedi iz činjenice da se u njemu privremeno sačuva rezultat izvođenja ALU operacije. Naime, ne bi bilo moguće direktno rezultat poslati natrag podatkovnom sabirnicom, jer se ona u to vrijeme koristi sa dovođenje drugog operanda na ALU. Za vrijeme izvođenja ALU operacije jedan operand dolazi iz akumulatora (registar A) dok drugi iz memorije dolazi preko sabirnice podataka. Nakon što se rezultat operacije zapiše u TMP registar, memorija oslobodi sabirnicu (OE = 0, WE = 0) te se uključuje izlaz TMP registra, čime se rezultat ALU operacije postavlja na sabirnicu podataka (dbus). Zatim se memorija prebacuje u pisanje te se podatak sa sabirnice zapiše na predviđenu lokaciju u memoriji.

Treći važan registar je instrukcijski registar IR. U njemu se upisuje operacijski kod trenutne instrukcije. Podatak do njega također dolazi podatkovnom sabirnicom, a hoće li se upisati u akumulator (ukoliko je konstanta) ili u instrukcijski registar (ukoliko je operacijski kod) ovisi o tome na koji će registar biti poslan `write` signal. O generiranju `write` (i mnogo drugih) signala brine se upravljačka jedinica.



Slika 2: Organizacija demonstracijskog procesora

Podatak iz instrukcijskog registra putuje u kontrolnu jedinicu i u aritmetičko-logičku jedinicu. Nižih tri bita zajedno s četvrtim bitom određuju operaciju koju treba izvesti. Ukoliko se radi o ALU operaciji, tada četvrti bit određuje da li se radi o aritmetičkoj ili logičkoj operaciji dok preostali bitovi (najviši) određuju operaciju koju će ALU izvršiti. Kodna je riječ 8-bitna te je njezin format sljedeći:

7 6 5 4	3	2 1 0
ALU operacija	ALU mode/ kod naredbe	kod naredbe

Bitovi 3-0 predstavljaju signale koji stižu u upravljačku jedinicu te specificiraju instrukciju. Bitovi 7-4, zajedno sa bitom 3 određuju koju će operaciju ALU izvršiti. Detaljnije o tome će biti riječ u poglavlju koji opisuje rad aritmetičko-logičke jedinice. Upravljačka jedinica i ALU također razmjenjuju signale

`carry` *(ulazni i izlazni)* te `equal`.

Unutar CPLD1 su modelirani programsko brojilo (PC), registar pokazivač koji služi adresiranju 16--bitnog adresnog prostora (inače bi sa 8 bita mogli adresirati samo 256 memorijskih lokacija), o*ffset* registar (OFF) koji služi za brzo pristupanje segmentu s 256 memorijskih lokacija na najvišoj stranici te pripadna logika koja svime time upravlja. U nastavku će svaki od ovih elemenata biti detaljnije obrađen.

2. Aritmetičko-logička jedinica

Aritmetičko-logička jedinica je napravljena od dva 74HCT181 4-bitna ALU-a. Svaki takav 4-bitni ALU ima slijedeće pinove:

- **S0-S3** - 4 bita koji određuju operaciju koju će obaviti.
- **M** - bit koji određuje da li se obavlja aritmetička ili logička operacija.
- **A0-A3** te **B0-A3 –** dva 4-bitna operanda koji sudjeluju u određenoj operaciji.
- **Cn** - Ulaz za *carry.*
- **Cn+4 –** Izlazni *carry.*
- **F0-F3** - 4-bitni rezultat operacije nad operandima A i B.
- **A=B** - izlaz koji se aktivira ukoliko su operandi međusobno jednaki.

ALU ima različite funkcijske tablice za pozitivnu i negativnu logiku. Kako se naš procesor skroz temelji na pozitivnoj logici (logičku jedinicu predstavlja visoka naponska razina), tablica s pojedinim funkcijama je sljedeća:

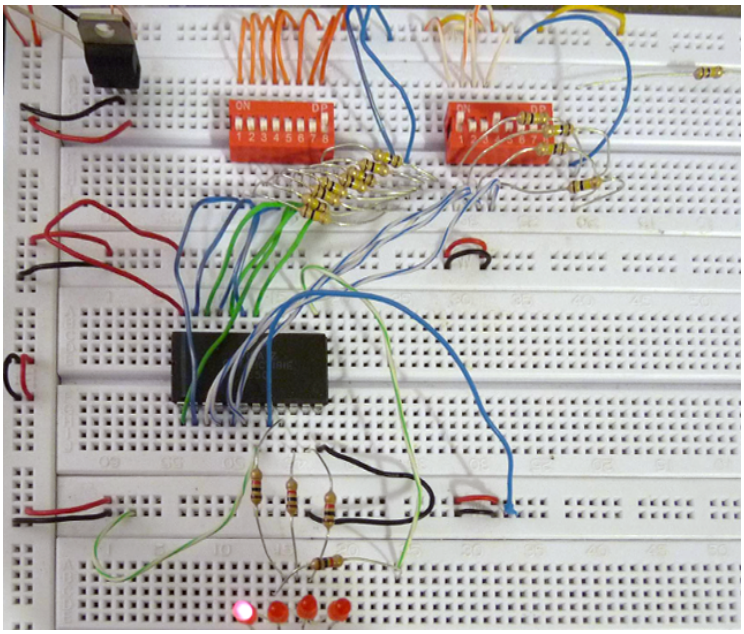
Selection				Active High Data		
				M = H	M = L; Arithmetic Operations	
S3	S2	S1	S0	Logic Functions	C_n = H (no carry)	C_n = L (with carry)
L	L	L	L	$F = \overline{A}$	F = A	F = A Plus 1
L	L	L	H	$F = \overline{A + B}$	F = A + B	F = (A + B) Plus 1
L	L	H	L	$F = \overline{A}B$	F = A + $\overline{B}$	F = (A + $\overline{B}$) Plus 1
L	L	H	H	F = 0	F = Minus 1 (2's Compl)	F = Zero
L	H	L	L	$F = \overline{AB}$	F = A Plus A$\overline{B}$	F = A Plus A$\overline{B}$ Plus 1
L	H	L	H	$F = \overline{B}$	F = (A + B) Plus A$\overline{B}$	F = (A + B) Plus A$\overline{B}$ Plus 1
L	H	H	L	F = A ⊕ B	F = A Minus B Minus 1	F = A Minus B
L	H	H	H	F = A$\overline{B}$	F = A$\overline{B}$ Minus 1	F = A$\overline{B}$
H	L	L	L	$F = \overline{A} + B$	F = A Plus AB	F = A Plus AB Plus 1
H	L	L	H	$F = \overline{A \oplus B}$	F = A Plus B	F = A Plus B Plus 1
H	L	H	L	F = B	F = (A + $\overline{B}$) Plus AB	F = (A + $\overline{B}$) Plus AB Plus 1
H	L	H	H	F = AB	F = AB Minus 1	F = AB
H	H	L	L	F = 1	F = A Plus A*	F = A Plus A Plus 1
H	H	L	H	F = A + $\overline{B}$	F = (A + B) Plus A	F = (A + B) Plus A Plus 1
H	H	H	L	F = A + B	F = (A + $\overline{B}$) Plus A	F = (A + $\overline{B}$) Plus A Plus 1
H	H	H	H	F = A	F = A Minus 1	F = A

Slika 3: Funkcije ALU-a (izvadak iz datasheeta)

Vrijedi napomenuti da se u tablici "+" odnosi na operator disjunkcije, a "plus" na operator zbrajanja. Gledamo na slijedeći način:

- Ukoliko je bit (pin) M jednak 1 (high), tada se obavlja logička operacija (lijeva kolona)
- Ukoliko je bit (pin) M jednak 0 (low), tada se obavlja aritmetička operacija (desna kolona)

Koja će se operacija obaviti, ovisiti će o 4-bitnom kodu kojeg pošaljemo na ulaz S0-S3. Tako će se, primjerice, ukoliko postavimo M = 1 te S(0-3) = "000" komplementirati ulaz A te pojaviti na izlazima F(0-3). Ulaz B se tada zanemaruje. Ukoliko postavimo M = 0 te S(0-3) = "1001", tada će se zbrojiti operandi A i B, te njihov zbroj pojaviti na izlazu F(0-3). Analogno vrijedi za sve ostale retke u tablici. Prije nego što smo krenuli dalje na razvoj procesora, provjeren je rad kupljenih 4-bitnih arimetičko-logičkih jedinica na prototipnoj ploči, što je prikazano na slici 4.

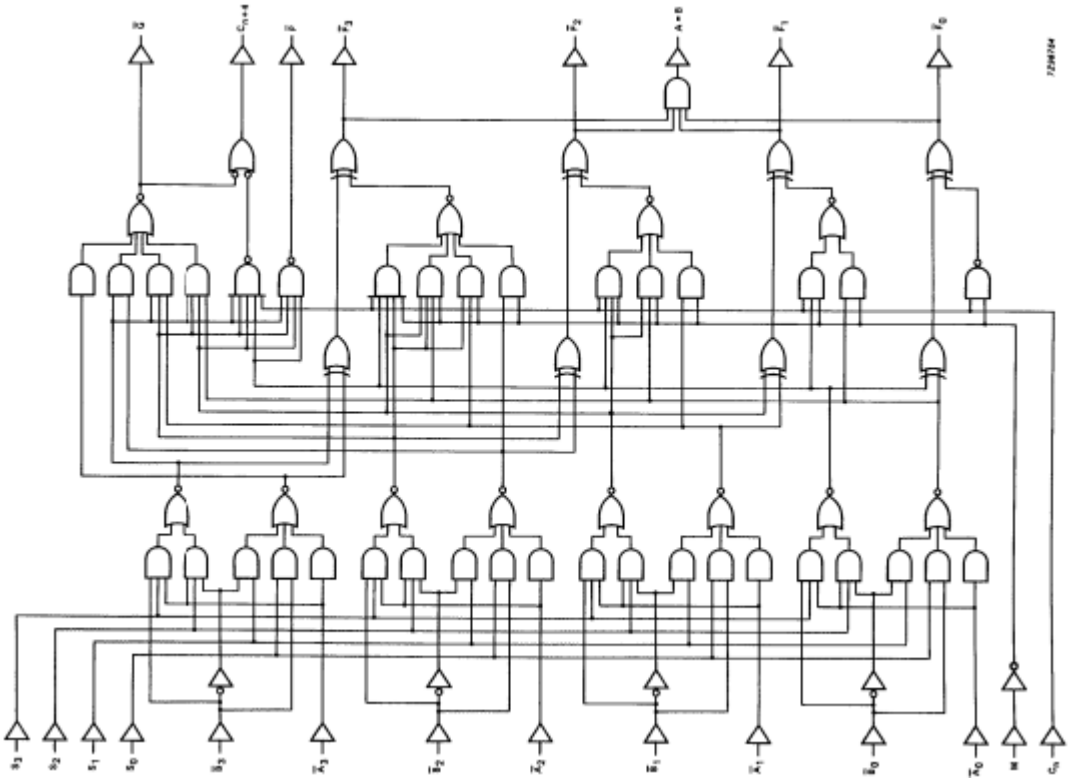


Slika 4: Testiranje 4-bitnog ALU-a

2.1. VHDL model 4-bitne aritmetičko-logičke jedinice

Kao što je već rečeno, radi potrebe simuliranja rada procesora (otkrivanje grešaka isključivo na sklopovlju bilo bi jako teško) napravljeni su VHDL modeli pojedinih komponenti. Prva i najbitnija takva komponenta je upravo aritmetičko-logička jedinica. Nju smo odlučili modelirati strukturalno na temelju sheme iz *datasheeta.* (Ponašajnim modeliranjem postojala bi mogućnost da smo zapravo modelirali krivo ponašanje, dok strukturnim modeliranjem je ono zasigurno jednako modeliranom

elementu, s obzirom da su im strukture iste). Strukturalna shema 74HC181 4-bitne ALU jedinice prikazana je na slici 5. Na temelju te slike napisan je strukturalni VHDL model aritmetičko-logičke jedinice.



Slika 5: Shema 74HC181 4-bitnog ALU-a (izvadak iz datasheeta)

Prvo smo na shemi numerirali sve signale po redu, a zatim smo pojedine veze opisali VHDL kodom. Na samom kraju napisan je *testbench* kojim je provjerena ispravnost VHDL modela. Konačni VHDL kod 4-bitne aritmetičko-logičke jedinice dan je u nastavku:

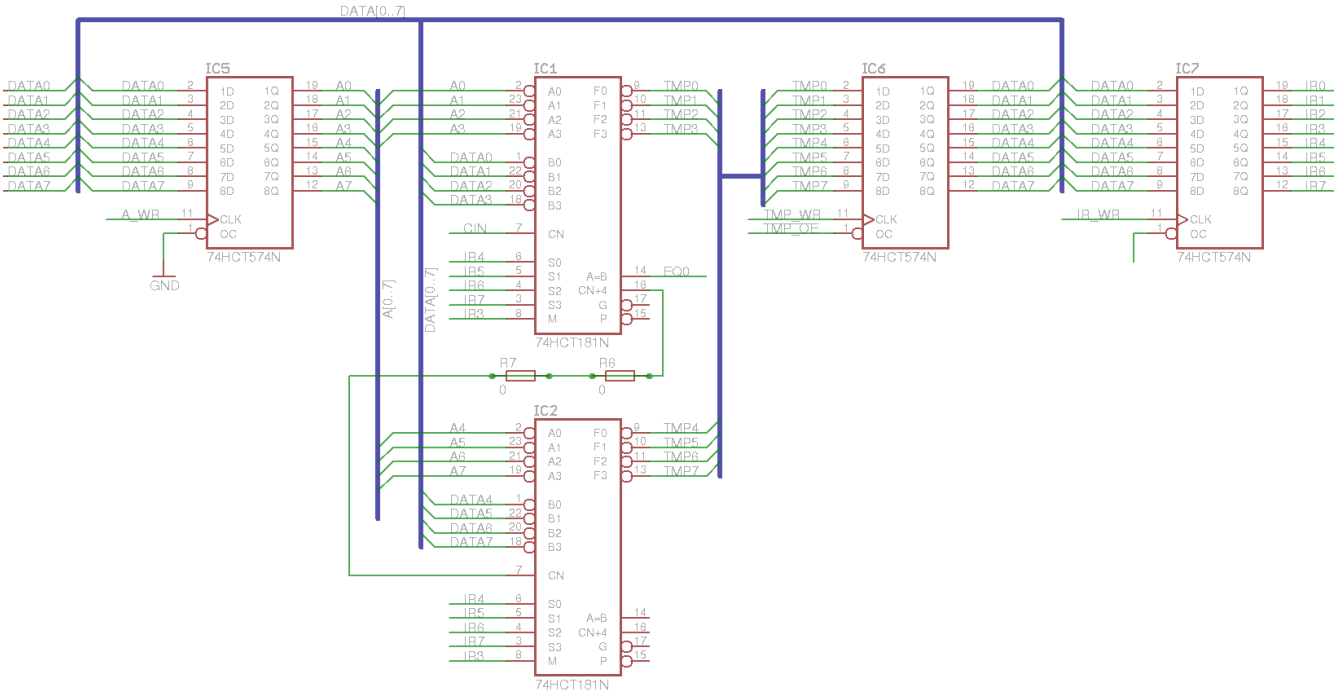
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity alu181 is
    port(
        M: in std_logic; -- Mode (aritmetička ili logička operacija)
        S: in std_logic_vector(3 downto 0); -- operation Select
        A: in std_logic_vector(3 downto 0); -- first operand
        B: in std_logic_vector(3 downto 0); -- second operand
        Cn: in std_logic; -- carry input
        F: out std_logic_vector(3 downto 0); -- function outputs
        P: out std_logic; -- carry propagate
        G: out std_logic; -- carry generate
        Cn4: out std_logic; -- carry output
        AeqB : out std_logic -- 1 if operators (A and B) equal
    );
end alu181;

architecture Behavioral of alu181 is
    signal s1 : std_logic_vector(19 downto 0);
    signal s2 : std_logic_vector(7 downto 0);
```

```
    signal s3 : std_logic_vector(19 downto 0);
    signal s4 : std_logic_vector(3 downto 0);
    signal s5 : std_logic_vector(4 downto 0);
begin
        -- prvi red signala
        s1(0) <= B(3) and S(3) and A(3);
        s1(1) <= A(3) and S(2) and not B(3);
        s1(2) <= not B(3) and S(1);
        s1(3) <= S(0) and B(3);
        s1(4) <= A(3);

        s1(5) <= B(2) and S(3) and A(2);
        s1(6) <= A(2) and S(2) and not B(2);
        s1(7) <= not B(2) and S(1);
        s1(8) <= S(0) and B(2);
        s1(9) <= A(2);

        s1(10) <= B(1) and S(3) and A(1);
        s1(11) <= A(1) and S(2) and not B(1);
        s1(12) <= not B(1) and S(1);
        s1(13) <= S(0) and B(1);
        s1(14) <= A(1);

        s1(15) <= B(0) and S(3) and A(0);
        s1(16) <= A(0) and S(2) and not B(0);
        s1(17) <= not B(0) and S(1);
        s1(18) <= S(0) and B(0);
        s1(19) <= A(0);

        -- drugi red signala
        s2(0) <= not( s1(0) or s1(1) );
        s2(1) <= not( s1(2) or s1(3) or s1(4) );
        s2(2) <= not( s1(5) or s1(6) );
        s2(3) <= not( s1(7) or s1(8) or s1(9) );
        s2(4) <= not( s1(10) or s1(11) );
        s2(5) <= not( s1(12) or s1(13) or s1(14) );
        s2(6) <= not( s1(15) or s1(16) );
        s2(7) <= not( s1(17) or s1(18) or s1(19) );

        -- treći red signala
        s3(0) <= s2(1);
        s3(1) <= s2(0) and s2(3);
        s3(2) <= s2(0) and s2(2) and s2(5);
        s3(3) <= s2(0) and s2(2) and s2(4) and s2(7);
        s3(4) <= not ( s2(0) and s2(2) and s2(4) and s2(6) and Cn);
        s3(5) <= not ( s2(0) and s2(2) and s2(4) and s2(6) );
        s3(6) <= s2(0) xor s2(1);

        s3(7) <= Cn and s2(6) and s2(4) and s2(2) and not M;
        s3(8) <= s2(4) and s2(2) and s2(7) and not M;
        s3(9) <= s2(2) and s2(5) and not M;
        s3(10) <= s2(3) and not M;

        s3(11) <= s2(2) xor s2(3);
        s3(12) <= Cn and s2(6) and s2(4) and not M;
        s3(13) <= s2(4) and s2(7) and not M;
        s3(14) <= s2(5) and not M;
        s3(15) <= s2(4) xor s2(5);

        s3(16) <= Cn and s2(6) and not M;
        s3(17) <= s2(7) and not M;
        s3(18) <= s2(6) xor s2(7);
        s3(19) <= not (Cn and not M);

        -- četvrti red signala
        s4(0) <= not ( s3(0) or s3(1) or s3(2) or s3(3) );
        s4(1) <= not ( s3(7) or s3(8) or s3(9) or s3(10) );
        s4(2) <= not ( s3(12) or s3(13) or s3(14) );
        s4(3) <= not ( s3(16) or s3(17) );

        -- peti red signala
```

```
        s5(0) <= not s4(0) or not s3(4);
        s5(1) <= s3(6) xor s4(1);
        s5(2) <= s3(11) xor s4(2);
        s5(3) <= s3(15) xor s4(3);
        s5(4) <= s3(18) xor s3(19);

        -- izlazi
        G <= s4(0);
        Cn4 <= s5(0);
        P <= s3(5);
        F(3) <= s5(1);
        F(2) <= s5(2);
        F(1) <= s5(3);
        F(0) <= s5(4);
        AeqB <= s5(1) and s5(2) and s5(3) and s5(4);
end Behavioral;
```

2.2. Izrada 8-bitne aritmetičko-logičke jedinice

Kako je procesor 8-bitni, potrebna je i 8-bitna aritmetičko-logička jedinica. Kao što je već spomenuto, nju smo napravili od dvije gotove 4-bitne ALU jedinice 74xx serije. Spajanje 4-bitnih jedinica je napravljeno tako da *`carry out`* bit niže jedinice ulazi (propagira) u *`carry in`* bit više jedinice. *`Carry in`* niže i *`carry out`* više jedinice su spojene na kontrolnu jedinicu. *`Equal`* signali iz više i niže jedinice se dovode do kontrolne jedinice gdje se obavlja logička operacija *AND* nad njima. (S obzirom da su *equal* signali zapravo *open-collector*, bilo je moguće samo "spojiti žice" tih signala te postaviti *pull-up* otpornike, međutim taj je detalj uočen tek kasnije.) 8-bitni podatci iz akumulatora (A) i podatkovne sabirnice (*dbus*) se granaju tako da nižih 4 bita odlazi na ALU koji računa nižih 4 bita, dok viših 4 bita odlazi u ALU koji računa viših 4 bita te dobiva *carry* od nižeg ALU-a. Kako obje jedinice moraju istovremeno obavljati istu operaciju nad podijeljenim podacima, potrebno je iste upravljačke signale (S0-S3 te M) iz instrukcijskog registra, koji određuju operaciju koja će se obaviti, dovesti na pripadne ulaze obiju jedinica. Izlazi pojedinih ALU jedinica se opet spajaju u TMP registar gdje tvore 8-bitni rezultat. Izlazi nižeg ALU-a se dovodi na niže bitove TMP registra, a izlazi višeg ALU-a se dovode na više bitove TMP registra. Podatak se u TMP registar na kraju zapisuje simultano nakon što su se izlazi obiju jedinica stabilizirali.



Slika 6: Povezivanje dva 4-bitna ALU-a u funkcionalnu 8-bitnu cjelinu

Na slici 6. prikazano je povezivanje obavljeno u sklopovskoj implementaciji procesora (shema je nacrtana u programu *CADSoft Eagle* koji je korišten prilikom izrade pločica (PCB-a) da bi se olakšao posao prospajanja vodova).

Radi potreba simulacije, bilo je potrebno isto tako instancirati dva 4-bitna ALU-a te ih povezati na gore opisani način. To je zapravo napisano u VHDL kodu koji opisuje kako su sve cjeline procesora povezane, a isječak koji se tiče povezivanja ALU-a prikazan je u nastavku:

```
architecture Structural of whole is

    signal dbus : std_logic_vector(7 downto 0); -- data bus
    signal abus : std_logic_vector(15 downto 0); -- address bus

    signal sAA : std_logic_vector(7 downto 0); -- povezivanje A reg - ALU
    signal sAT : std_logic_vector(7 downto 0); -- povezivanje ALU - TMP reg

    -- Control Unit <-> ALU
    signal sCin : std_logic;
    signal sCout : std_logic;
    signal sComp1 : std_logic;
    signal sComp2 : std_logic;

    -- Control Unit -> TMP reg
    signal snET : std_logic; -- OE
    signal sWT : std_logic; -- write

    -- Control Unit -> A reg
    signal sWA : std_logic; -- write
```

```
    -- IR reg -> ALU
    signal sfunc : std_logic_vector(3 downto 0);

    -- IR optype -> ALU, CU
    signal sopt : std_logic;

    -- veza izmedju dva 4 bitna alua
    signal scarry : std_logic;

begin

    IR : entity work.registar port map(
        clk => sWIR,
        noe => '0',
        d => dbus,
        q(3 downto 0) => sfunc,
        q(4) => sopt,
        q(7 downto 5) => sopcode(2 downto 0)
    );

    A : entity work.registar port map(
        clk => sWA,
        noe => '0', -- OE je stalno ukljucen
        d => dbus,
        q => sAA -- A -> ALU(operand1)
    );

    aluLow : entity work.alu181 port map(
        A => sAA(3 downto 0),
        B => dbus(3 downto 0),
        F => sAT(3 downto 0),
        Cn4 => scarry, -- carry output -> signal carry
        M => sopt, -- operation type (logic/arith)
        S => sfunc, -- alu operation
        Cn => sCin, -- carry in iz CU
        AeqB => sComp1 -- comparator output
    );

    aluHigh : entity work.alu181 port map(
        A => sAA(7 downto 4),
        B => dbus(7 downto 4),
        F => sAT(7 downto 4),
        Cn => scarry, -- signal carry -> carry input
        M => sopt, -- operation type (logic/arith)
        S => sfunc, -- alu operation
        Cn4 => sCout, -- carry output u CU
        AeqB => sComp2 -- comparator output
    );

    T : entity work.registar port map(
        clk => sWT,
        noe => snET,
        d => sAT,
        q => dbus
    );

    --------- <cut>-----------------
end Structural;
```

3. Registri A, IR i TMP

Za 8-bitne registre koji se ne moraju povećavati (kao primjerice PC registar) korišteni su 74HC574. Radi se o osmerostrukom D bistabilu CMOS tehnologije koji zapisuje podatak na rastući brid. IC sadrži 8 ulaza, 8 izlaza, ulaz za *clock* CP (tj. signal za zapisivanje) te signal za uključivanje izlaza OE. Osim aritmetičko-logičke jedinice, radi potrebe simulacije (i lakšeg modeliranja kontrolne jedinice) potrebno je bilo modelirati i navedene registre. Korištenje je vrlo jednostavno i intuitivno (za razliku od ALU-a) pa smo se odlučili registar modelirati ponašajno, što je, u ovom slučaju, puno brže od strukturalnog modeliranja. VHDL kod 74HC574 registara je naveden u nastavku:

```
entity registar is

PORT(
clk, noe: IN std_logic;
d: IN std_logic_vector(7 downto 0);
q: OUT std_logic_vector (7 downto 0)
);END registar;

ARCHITECTURE behavioral OF registar IS
    SIGNAL data : std_logic_vector(7 downto 0);
BEGIN
p: PROCESS (clk, d, noe)
BEGIN

IF (rising_edge(clk)) THEN
    data<=d;
END IF;

IF (noe='0') THEN
    q <= data;
ELSE
    q<= "ZZZZZZZZ";
END IF;
END PROCESS p;
```

Kod svakog registra postavljen je konektor koji omogućava spajanje indikatorskih ledica, pomoću kojih je moguće vidjeti stanje registra i trenutak zapisivanja. Kod registara A i IR je to učinjeno na njihovom izlazu (pošto je njihov izlaz stalno uključen), dok je kod registra TMP to učinjeno na ulazu.

4. Registri PC, PT i OFF te upravljanje adresnom sabirnicom

Programsko brojilo je specifično po tome što, osim upisa podatka, zahtjeva mogućnost brzog uvećanja za jedan (bilo bi naravno moguće iskoristiti ALU da uveća PC, no to bi bilo sporo i krajnje neefikasno). Drugi problem koji se javlja je činjenica da bi sa 8 bita mogli adresirati adresni prostor od svega 256 bytea, što je definitivno malo za ikakvu realnu aplikaciju. Da bi riješili ovu ograničenje, odlučeno je uvesti adresnu sabirnicu širine 16 bita. Time je omogućeno adresiranje ukupno 65 536 bytea, odnosno 64 kB. Da bi se 8-bitnim procesorom moglo upravljati 16-bitnom adresnom sabirnicom uveden je PT registar (pointer register) koji je 16-bitni gledano na izlazu, no zapravo se tvori od dva zasebna registra (viših i nižih 8 bitova) u koja se može zasebno upisivati. PC registar je isto tako 16-bitni registar (s mogućnošću povećanja za 1 i resetiranja), a u njega se može upisivati podatak preko PT registra, što je potrebno prilikom operacija skoka.

Uveden je i OFF registar (o*ffset*) koji služi brzom pristupanju najviših 256 memorijskih lokacija. On je 8-bitni registar koji sadrži nižih 8 bitova adrese, dok je viših 8 bitova direktno povezano na VCC (te tako tvore logičku jedinicu). Proces adresiranja tih lokacija je ubrzan time što nije potrebno učitati prvo niži byte adrese a zatim viši (kao kod korištenja PT registra), već se učitava samo niži byte, dok je viši fiksiran na 0xFF.

S obzirom na opseg opisane logike, odlučeno ju je modelirati u VHDL-u te sintetizirati u jedan Xilixov CPLD (označen kao CPLD1).

4.1. Modeliranje PC registra

Kao što je već rečeno, PC registar je 16-bitni te ima mogućnost uvećanja za jedan te resetiranja (na početku te prilikom pritiska tipke reset treba pokazivati na 0x0000). Sučelje PC registra je zamišljeno na sljedeći način:

- **clk** – ulaz signala takta.
- **s** – postavlja podatak koji je na ulazu registra prilikom prvog rastućeg brida takta.
- **r** – resetira brojač (postavlja se na nulu). Asinkrono.
- **i** – uvećanje za 1 (*increase*). Kada je postavljen, uveća se prilikom prvog rastućeg brida.
- **D** – ulaz registra (upisivanje vrijednosti prilikom skokova)
- **Q** – izlaz PC registra

Registar mora imati mogućnost da pamti stanje. To se fizikalno izvodi nizom bistabila koji pamte stanje, dok je prilikom modeliranja VHDL-om dovoljno deklarirati signal potrebne dužine. Sintetizator će kasnije obaviti sve potrebno. Nadalje, potrebno je omogućiti uvećanje za jedan. To je u VHDL-u jednostavno modelirati korištenjem signala tipa ***unsigned*** potrebne duljine. U našem slučaju bi to bilo:

```
signal cnt: unsigned(15 downto 0) := "0000000000000000";
```

Bitno je naglasiti važnost deklariranja inicijalne vrijednosti registra, bez koje ne bi bili sigurni gdje PC registar prikazuje prilikom uključivanja. Uvećanje se postiže jednostavnom aritmetičkom oznakom zbrajanja u VHDL-u. Važno je jedino koristiti "IEEE.std_logic_arith.all", a sintetizator će napraviti pripadno sklopovlje. Uvećanje se obavlja ukoliko je signal *i* postavljen na logičku jedinicu i to prilikom rastućeg brida signala takta.

Na kraju vrijedi napomenuti da u PC registru nije implementiran nikakav signal omogućavanja izlaza (OE – *Output Enable*) već je njegov izlaz stalno omogućen, a preko multipleksera se odabire što će biti postavljeno na adresnu sabirnicu (ili hoće li ona biti u stanju visoke impedancije). No o tome će biti više riječi kasnije. Konačan VHDL kod PC registara naveden je u nastavku:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.std_logic_arith.all;

entity reg16bitSetInc is
    port(
        clk : in std_logic;
        s : in std_logic; -- set
        r : in std_logic; -- reset
        i : in std_logic; -- increment
        D : in std_logic_vector(15 downto 0); -- input
        Q : out std_logic_vector(15 downto 0) -- output
    );
end reg16bitSetInc;

architecture Behavioral of reg16bitSetInc is
    signal cnt: unsigned(15 downto 0) := "0000000000000000";
begin
    p: process(clk, D, s, r, i)
    begin
        if (r = '1') then
            cnt <= "0000000000000000";
        else
            if (rising_edge(clk)) then
                if (s = '1') then
                    cnt <= unsigned(D);
                elsif (i = '1') then
                    cnt <= cnt + 1;
                end if;
            end if;
        end if;
    end process p;
```

```
    Q <= std_logic_vector(cnt);
end Behavioral;
```

4.2. Modeliranje PT i OFF registara

Registri PT i OFF su jednostavniji od već opisanog PC registra. Oni trebaju samo učitati vrijednost, sačuvati ju te prikazati na izlazu. Iz istog razloga kao i kod PC registra nema potrebe za signalom koji uključuje izlaz (multiplekserom se bira registar čiji se izlaz prikazuje na adresnoj sabirnici). VHDL kod koji se koristi u instancama tih registara je sljedeći:

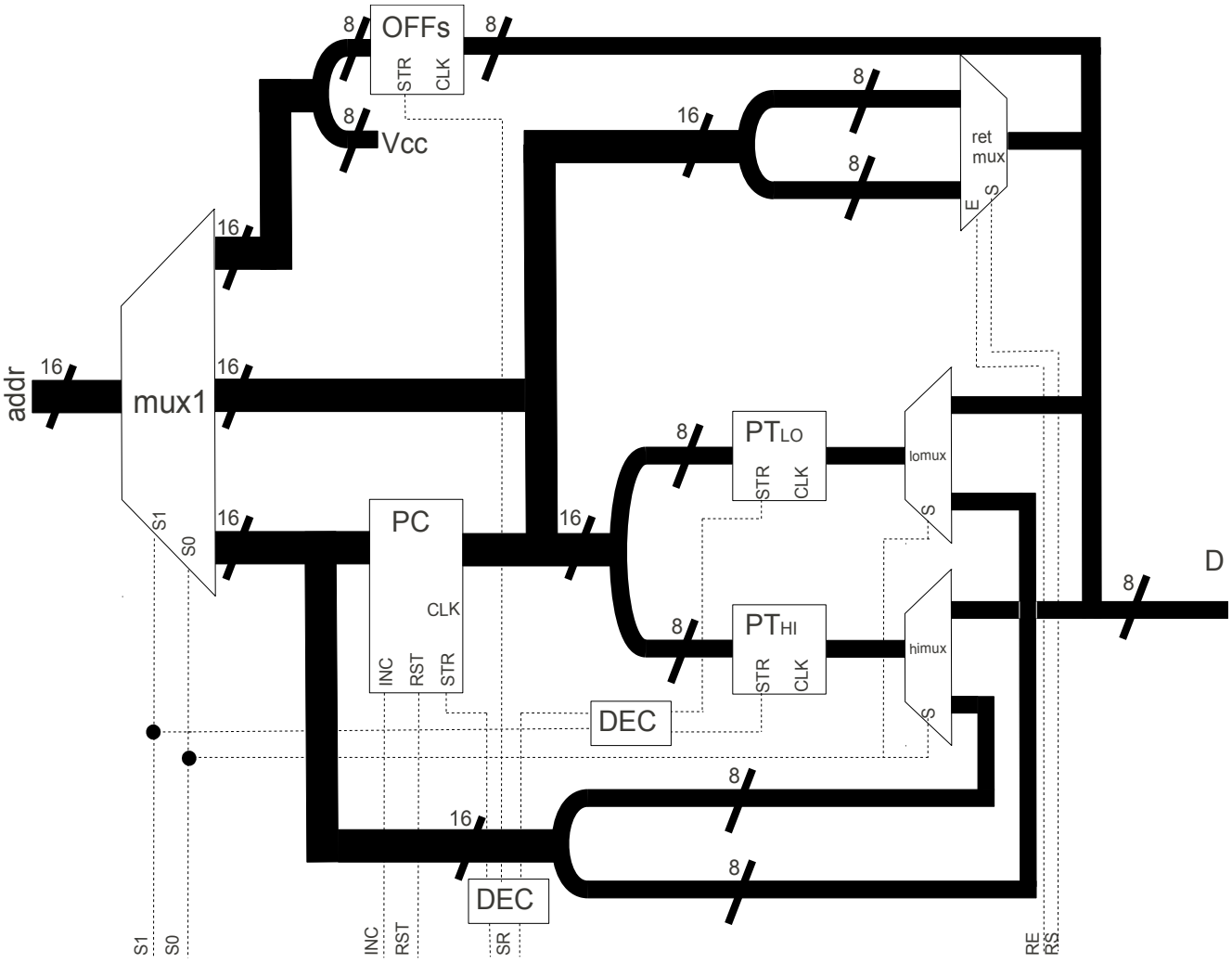
```
entity reg8bitSet is
    port(
        clk : in std_logic;
        s : in std_logic; -- set
        D : in std_logic_vector(7 downto 0); -- input
        Q : out std_logic_vector(7 downto 0) -- output
    );
end reg8bitSet;

architecture Behavioral of reg8bitSet is
begin
    p: process(clk, D, s)
    begin
        if (s = '1') then
            if (rising_edge(clk)) then
                Q <= D;
            end if;
        end if;
    end process p;
end Behavioral;
```

Razlika između tih registara je u načinu spajanja. Ulaz OFF registra je direktno spojen na adresnu sabirnicu a njegov je izlaz spojen na nižih 8 bitova jednog od ulaza multipleksera koji odabire izlaz na adresnoj sabirnici. Preostalih 8 bitova je, kao što je već rečeno, spojeno na napon napajanja VCC čime ih se postavlja na logičku jedinicu.

PT registar se sastoji od dva ovakva registra. Jedan registar (PTLow) čuva nižih 8 bitova, dok drugi takav registar (PTHigh) čuva viših 8 bitova. Njihovi izlazi zajedno tvore 16-bitni podatak koji se onda dovodi na multiplekser koji određuje izlaz na adresnoj sabirnici. Registri PTLow i PTHigh nisu direktno spojeni na 8-bitnu podatkovnu sabirnicu već se njihov ulaz određuje preko drugog skupa multipleksera. Razlog za takvo što leži u tome da je dodan povratni put iz 16-bitnog PC registra u PT registar čime je omogućeno učitavanje sadržaja PC registra u PT registar koji se onda može odvojeno (bilo prvo 8 nižih bitova pa 8 viših ili obrnuto) i po potrebi učitati u akumulator (registar A). Takva je instrukcija korisna za čuvanje stanja registra PC prilikom izvođenja CALL naredbi. Iz tog razloga se izlaz PTLow i PTHigh registra kroz set multipleksera (čiji se izlaz može postaviti u stanju visoke impedancije – jer bi inače smetao radu podatkovne sabirnice) vraća na podatkovnu sabirnicu.

4.3. Struktura CPLD1

Sada je već jasno da će se unutar prvog CPLD-a (CPLD1) nalaziti registri PC, PT (u obliku dva zasebna registra PTLow i PTHigh) i OFF te 3 skupa multipleksera. Jedan 16-bitni za biranje izlaza na adresnoj sabirnici te ostatak 8-bitnih za biranje ulaza u PT registar i uključivanja povratne veze PT registra (čitanje njegovog sadržaja). Najveći problem prilikom oblikovanja CPLD1 je bio ograničeni broj *pinova* na samom CPLD-u. Veliki broj *pinova* se koristi na adresnoj sabirnici (čak 16) te ih još 8 odlazi na podatkovnoj sabirnici. Stoga nije bilo moguće svaki pojedini signal "izvući" kroz *pinove* CPLD-a, što je riješeno dodavanjem dva dekodera i dijeljenjem par signala koji se ne koriste istovremeno. Prvi dekoder prima na ulazu dvobitni signal *sr* (*select register*) te postavlja *store* signal na pripadni registar. Ukoliko je *sr = 00* tada na niti jednom registru nije postavljen *store* signal.



Slika 7: Struktura CPLD1

Izlazni multiplekser odabire koji će registar biti postavljen na adresnu sabirnicu ili postavlja adresnu sabirnicu u stanje visoke impedancije (što je ostavljeno kao moguće proširenje ukoliko želimo dodati sklop koji bi punio memoriju iz SD kartice). Kako se ti signali ne koriste prilikom punjenja PT registra sadržajem akumulatora ili PC registra, iskorišteni su za upravljanje drugog dekodera kojim odabiremo hoće li se 8-bitni podatak sa podatkovne sabirnice upisati u PTLow ili u PTHigh te za odabiranje hoće li se zapisati podatak sa podatkovne sabirnice ili iz veze prema PC registru.

Multiplekser povratne veze *ret mux* je isto takav da mu izlaz može biti u stanju visoke impedancije. Kao što smo već rekli, to je potrebno da ne bi izlaz PT registra "smetao" na podatkovnoj sabirnici kada su na njoj podatci koji dolaze ili odlaze u memoriju.

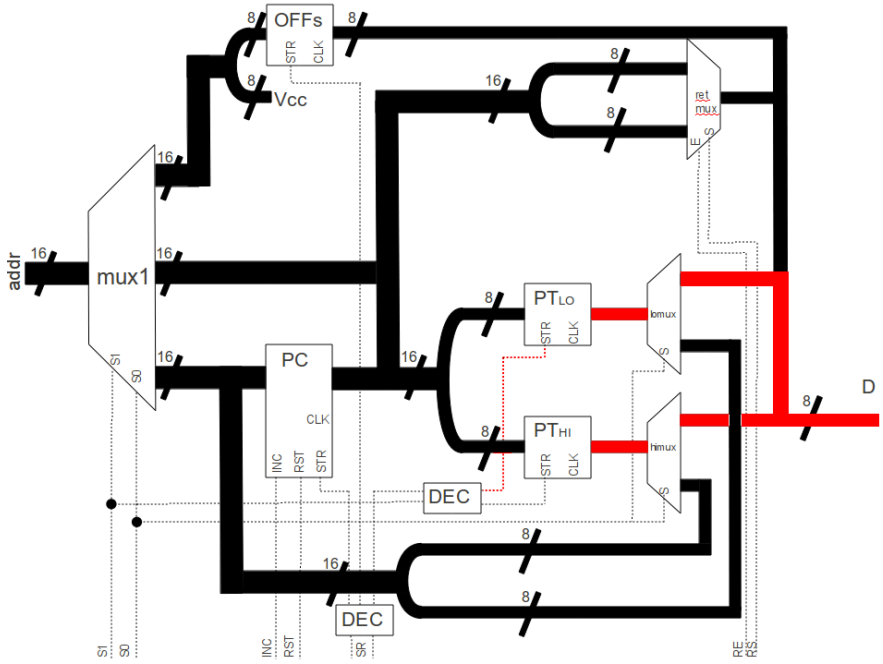
Multiplekseri su opisani ponašajno te je njihov opis prilično jednostavan:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity mux16x4 is
    port(
        sl : in std_logic_vector(1 downto 0); -- select
        d1 : in std_logic_vector(15 downto 0); -- input 1
        d2 : in std_logic_vector(15 downto 0); -- input 2
        d3 : in std_logic_vector(15 downto 0); -- input 3
        o : out std_logic_vector(15 downto 0) -- output
    );
end mux16x4;

architecture Behavioral of mux16x4 is
begin
    p: process(d1, d2, d3, sl)
    begin
        if (sl = "01") then
            o <= d1;
        elsif (sl = "10") then
            o <= d2;
        elsif (sl = "11") then
            o <= d3;
        else
            o <= "ZZZZZZZZZZZZZZZZ";
        end if;
    end process p;
end Behavioral;
```

Primjećujemo da su ovi multiplekseri specifični po tome što su u jednom stanju takvi da im je izlaz u stanju visoke impedancije. Ekvivalent bi bio da imaju normalan broj ulaza te da je prvi ulaz u stanju visoke impedancije. Kod je analogan za multiplekser povratne veze PT registra uz jedinu razliku što je on 8-bitan pa su takve i deklaracije ulaza i izlaza u njegovom kodu.

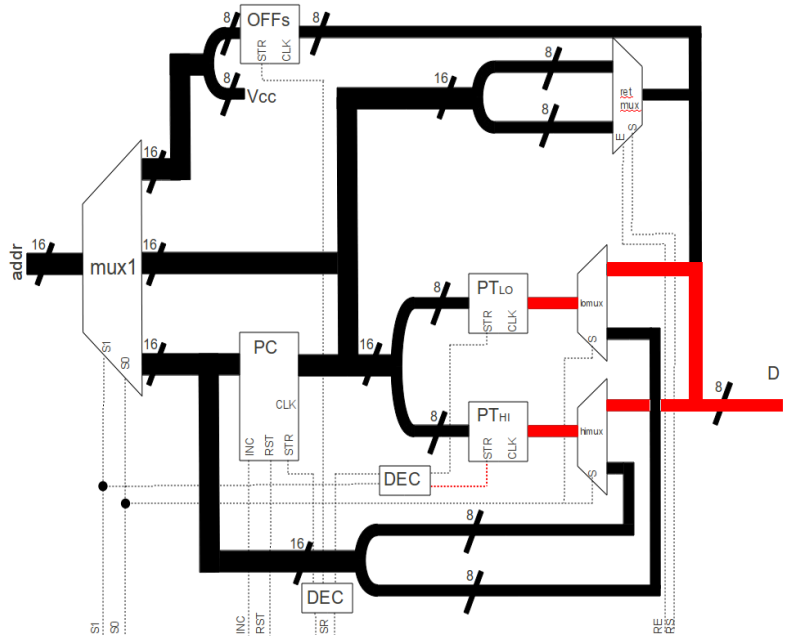
Najbolje je da rad prvog CPLD-a prikažemo kroz par primjera. Krenimo primjerom upisivanja sadržaja u PT registar i njegovo postavljanje na adresnu sabirnicu:

- Najprije je potrebno postaviti *sr* (*select register)* signal na *11* čime se odabire zapis u PT registar.
- Odabire se *s1 = 1* čime se odabire PTLow (nižih 8 bitova).
- Postavi se *s0 = 0* čime se na ulaznim multiplekserima PT registra odabire podatak sa podatkovne sabirnice (da primjerice želimo upisati sadržaj registra PC, tada bi *s0* postavili na logičku jedinicu).
- Dekoder tada pošalje *store* signal u PTLow registar te se u njega zapisuje podatak sa podatkovne sabirnice na prvi rastući brid signala takta. PTHigh pritom ostaje nepromijenjen.



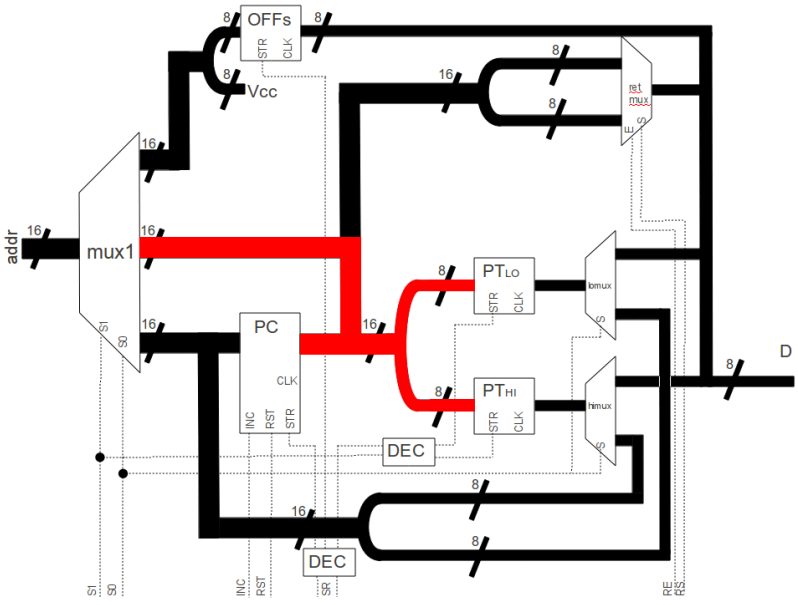
Slika 8: Upis podatka u PTLow

- Sljedeće je potrebno postaviti novi podatak na podatkovnu sabirnicu.
- Zatim prebacujemo *s1* u *0* čime dekoder postavlja *store* signal registru PTLow.
- Novi se podatak tada zapisuje u PTLow registar prilikom prvog rastućeg brida signala takta.
- Na kraju, potrebno je vratiti *sr* (*select register)* signal na *00* kako niti jedan registar ne bi bio odabran za upis.



Slika 9: Upis podatka u PTHigh

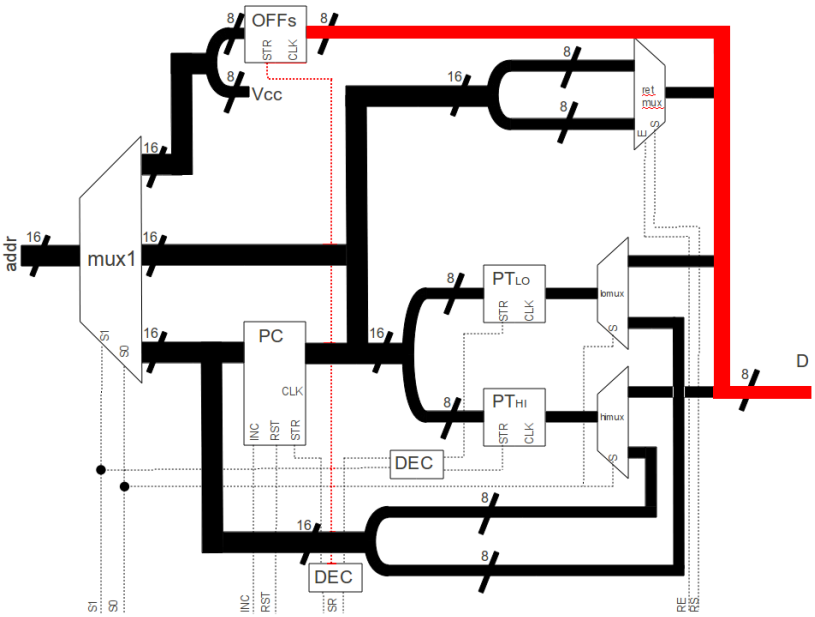
Ukoliko sada želimo postaviti 16-bitni podatak spremljen u PT registre na adresnu sabirnicu, to činimo postavljanjem signala *s0 = 0* i *s1 = 0* čime se na velikom 16-bitnom multiplekseru odabere izlaz iz PT registra.



Slika 10: Postavljanje sadržaja PT registra na adresnu sabirnicu

Jasno je iz primjera da je ovakva procedura dosta dugotrajna te iziskuje podosta taktova da bi se izvršila. Već smo rekli da smo iz tog razloga modelirali OFF (O*ffset*) registar koji omogućuje brže pristupanje najviših 256 lokacija u memoriji. Promotrimo niz radnji u tom slučaju:

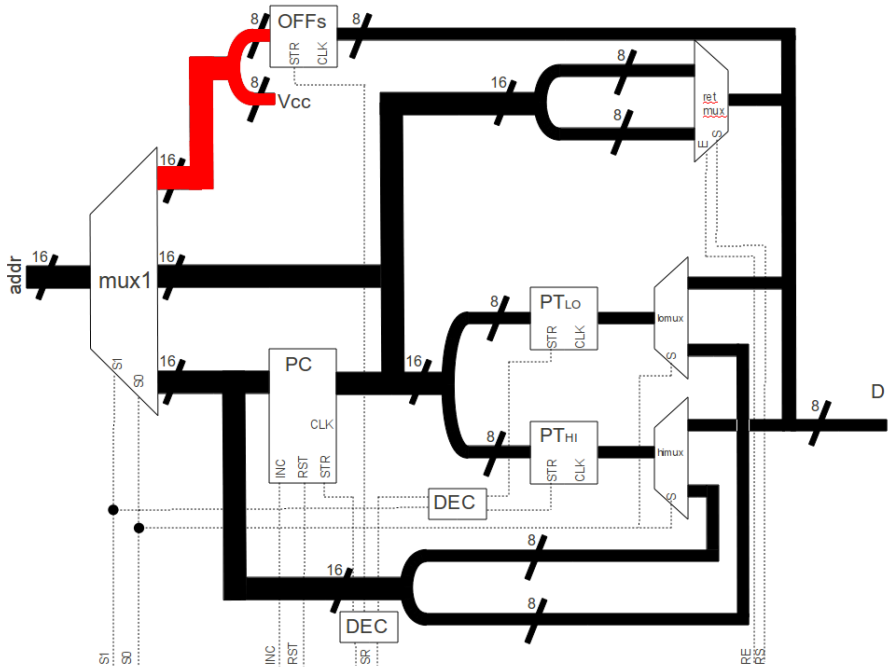
- Najprije se postavi *sr (select register)* na *10* nakon čega dekoder pošalje *store* signal OFF registru.
- 8-bitni podatak mora biti spreman na podatkovnoj sabirnici.
- Prilikom prvog rastućeg brida će podatak biti upisan u OFF registar.



Slika 11: Zapis podatka u OFF registar

Podatak kroz podatkovnu sabirnicu dolazi i do ulaza PT registara. Međutim, *store* signal je postavljen samo OFF registru te će on jedini upisati podatak.

Ukoliko sada želimo adresirati podatak koristeći se OFF registrom, postavljanjem signala *s1 = 1* i *s0 = 1* odabire se postavljanje izlaza OFF registra na adresnu sabirnicu. Kako je adresna sabirnica 16-bitna, a OFF registar 8-bitni, njegov se sadržaj postavlja na nižih 8 bitova adresne sabirnice. Viših 8 bitova se postavlja na logičku jedinicu povezivanjem na napon napajanja VCC. Time se samo jednim upisom podatka u OFF registar (za razliku od dva upisa prilikom korištenja PT registra) mogu brzo adresirati podatci u RAM memoriji od adrese 0xFF00 do adrese 0xFFFF. Te su memorijske lokacije u asembleru nazvane R0 do R255 te se programski mogu koristiti kao registri.



Slika 12: Odabir izlaza OFF registra na adresnoj sabirnici

Kompletan opis pojedinih signala i njihovih funkcija dan je u nastavku:

s0, s1 – odabir izlaza na adresnu sabirnicu

s1 = 0, s0 = 0 – adresna sabirnica je u stanju visoke impedancije

s1 = 0, s0 = 1 – na adresnoj se sabirnici pojavljuje sadržaj PC registra

s1 = 1, s0 = 0 – na adresnoj se sabirnici pojavljuje sadržaj PT registra

s1 = 1, s0 = 1 – na adresnoj se sabirnici pojavljuje sadržaj OFF registra na nižih 8 bitova, te 0xff na viših 8 bitova

sr – odabir registra u kojem se zapisuje podatak (select register)

sr = 00 – niti jedan registar nije odabran

sr = 01 – postavlja ***store*** signal na PC registar

sr = 10 – postavlja ***store*** signal na OFF registar

sr = 11 – postavlja ***store*** signal na PT registar i to:

s1 = 1 – postavlja se ***store*** signal na PTLow (nižih 8 bitova)

s1 = 0 – postavlja se ***store*** signal na PTHigh (viših 8 bitova)

s0 = 0 – zapisuje se podatak sa sabirnice d

s0 = 1 – zapisuje se podatak sa PC registra

inc, rst – upravljanje PC registrom

inc = 1 – PC se poveća za jedan pri rastućem bridu takta

rst = 0 – PC se postavlja na 0x0000

re, rs – upravljanje povratnom vezom PT registra

re = 1 – uključuje multiplekser koji postavlja sadržaj PT registra na podatkovnoj sabirnicu (inače je izlaz multipleksera u stanju visoke impedancije da ne bi smetao podatkovnoj sabirnici)

rs = 0 – na d sabirnicu se postavlja PTLow (donjih 8 bitova)

rs = 1 – na d sabirnicu se postavlja PTHigh (gornjih 8 bitova)

Najprije smo bili pokazali kako su modelirani pojedini registri i multiplekseri, zatim smo pokazali kako bi ih trebalo povezati da oni tvore smislenu i funkcionalnu cjelinu. Zatim je takvu cjelinu bilo potrebno napisati u VHDL-u. Kod kojim je to učinjeno je naveden u nastavku te nije ništa drugo nego strukturalno povezivanje prije modeliranih komponenti na identičan način kako je prikazano u dosadašnjim shemama na kojima smo promatrali rad CPLD-a.

```
entity cpld1 is
    port(
        clk : in std_logic;

        d : inout std_logic_vector(7 downto 0);

        sr : in std_logic_vector(1 downto 0);

        s0 : in std_logic;
        s1 : in std_logic;

        inc : in std_logic;
        rst : in std_logic; -- PC reset

        re : in std_logic;
        rs : in std_logic;

        o : out std_logic_vector(15 downto 0)
    );
end cpld1;

architecture Structural2 of cpld1 is

component mux8x2HiZ
    port(
        sl : in std_logic; -- select
        e : in std_logic; -- enable
        d1 : in std_logic_vector(7 downto 0); -- input 1
        d2 : in std_logic_vector(7 downto 0); -- input 2
        o : out std_logic_vector(7 downto 0) -- output
    );
end component;
```

```
component reg16bitSetInc
    port(
        clk : in std_logic;
        s : in std_logic; -- set
        r : in std_logic; -- reset
        i : in std_logic; -- increment
        D : in std_logic_vector(15 downto 0); -- input
        Q : out std_logic_vector(15 downto 0) -- output
    );
end component;

component reg8bitSet
    port(
        clk : in std_logic;
        s : in std_logic; -- set
        D : in std_logic_vector(7 downto 0); -- input
        Q : out std_logic_vector(7 downto 0) -- output
    );
end component;

component dec1
    port(
        s : in std_logic_vector(1 downto 0); -- select
        o : out std_logic_vector(2 downto 0) -- output
    );
end component;

component dec2
    port(
        s : in std_logic; -- select 1 or 2
        w : in std_logic; -- write enable
        o : out std_logic_vector(1 downto 0) -- output
    );
end component;

component mux8x2
    port(
        sl : in std_logic; -- select
        d1 : in std_logic_vector(7 downto 0); -- input 1
        d2 : in std_logic_vector(7 downto 0); -- input 2
        o : out std_logic_vector(7 downto 0) -- output
    );
end component;

    signal sg1, sg2 : std_logic_vector(15 downto 0);
    signal sg3, sg4, sg5 : std_logic_vector(7 downto 0);
    signal c1, c2, c3, c4, c5: std_logic;
begin
    mux1: entity work.mux16x4 port map(
        o => o,
        d1 => sg1,
        d2 => sg2,
        d3(7 downto 0) => sg3,
        d3(15 downto 8) => "11111111",
        sl(0) => s0,
        sl(1) => s1
    );

    pc : entity work.reg16bitSetInc port map(
        Q => sg1,
        D => sg2,
        i => inc,
        r => rst,
        s => c1,
        clk => clk
    );

    off : entity work.reg8bitSet port map(
        Q => sg3,
        D => d,
```

```
        s => c2,
        clk => clk
    );

    xdec1 : entity work.dec1 port map(
        s(0) => sr(0),
        s(1) => sr(1),
        o(2) => c1,
        o(1) => c2,
        o(0) => c3
    );

    xdec2 : entity work.dec2 port map(
        w => c3,
        s => s1,
        o(1) => c4,
        o(0) => c5
    );

    ptlo: entity work.reg8bitSet port map(
        Q => sg2(7 downto 0),
        D => sg4,
        s => c4,
        clk => clk
    );

    pthi : entity work.reg8bitSet port map(
        Q => sg2(15 downto 8),
        D => sg5,
        s => c5,
        clk => clk
    );

    muxlo : entity work.mux8x2 port map(
        o => sg4,
        d1 => d,
        d2 => sg1(7 downto 0),
        sl => s0
    );

    muxhi : entity work.mux8x2 port map(
        o => sg5,
        d1 => d,
        d2 => sg1(15 downto 8),
        sl => s0
    );

    retmux : entity work.mux8x2HiZ port map(
        o => d,
        d1 => sg2(7 downto 0),
        d2 => sg2(15 downto 8),
        e => re,
        sl => rs
    );
end Structural2;
```

Time je gotova izrada prvog CPLD-a koji sadrži registre PC, PT i OFF te logiku upravljanja adresnom sabirnicom. Nakon sinteze dobivena je sljedeća statistika iskorištenosti: 89% (64 od 72) makroćelija, 56% (40 od 72) registara te 98% (33 od 34) pinova. Vidljivo je da je broj pinova na CPLD-u koji je predstavljao kritični resurs dobro savladan. Iskorištenost po pitanju makroćelija je također dosta velika te je moguće zaključiti da je prvi CPLD zapravo dobro iskorišten; da je kupljeni model (XC9572XL) dobro zadovoljio svrhu te da nije niti predimenzioniran niti poddimenzioniran.

4.4. Provjera CPLD1 simulacijom

Prije daljnjeg nastavka s radom bilo je potrebno provjeriti i potvrditi ispravni rad prvog CPLD-a. Nekolicina takvih testova i pripadni opisi su dani u nastavku.

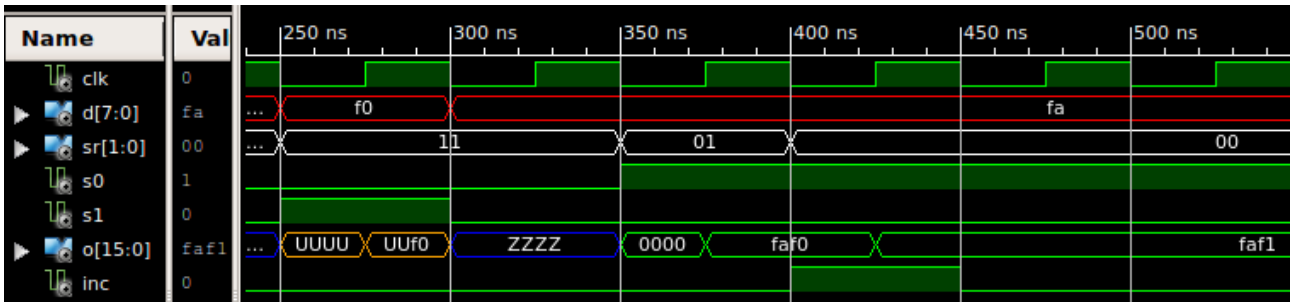
4.4.1. Upis vrijednosti u PT registar

Vidjeli smo prije da upis u PT registar obavljamo u dva koraka. U prvom koraku upisujemo donjih 8 bitova, dok u drugom koraku upisujemo gornjih 8 bitova. Da bi odabrali koji ćemo dio riječi zapisati postavljamo *sl* bit na pripadajuću vrijednost. Zatim, potrebno je poslati ***store*** signal pripadajućem registru. To činimo postavljanjem *sr* na 11.

```
-- ****** zapis podatka u PT registre ****** --
d <= "11110000"; -- postavimo donjih 8 bitova na d
s1 <= '1'; -- odaberemo PTlow
s0 <= '0'; -- ulaz iz D sabirnice
sr <= "11"; -- store signal PT-u
wait for clk_period;


d <= "11111010"; -- postavimo gornjh 8 bitova na d
s1 <= '0'; -- odaberemo PThigh
sr <= "11"; -- store signal PT-u
wait for clk_period;

-- ****** zapis PT-a u PC registar ****** --
sr <= "01"; -- store signal
-- postavimo PC na izlaz
s0 <= '1';
s1 <= '0';
```

Na slici 13. vidimo tijek signala za upravo obavljenu radnju. U trenutku *t = 250ns* su na sabirnicu d postavljeni nižih 8 bitova (0xf0) te je odabran registar PTLow (*sr = 11, s1 = 1*). Na rastući prvi sljedeći brid će se podatak zapisati u registar. U *t = 300ns* se na sabirnicu postavljaju viših 8 bitova (0xfa) te se odabire PTHigh registar (*s1 = 0*). Na rastući brid u t = 325ns se podatak zapisuje u registar.

Zatim, u t = 350ns odaberemo da na izlazu (adresnoj sabirnici) bude sadržaj PC registra (*s0 = 1, s1 = 0*) te istovremeno kažemo PC registru da učita sadržaj PT registra (*sr = 01*). Vidimo da se sadržaj PC-a odmah pojavio na izlazu, no tek se u *t = 375ns* (odnosno na rastući brid) podatak iz PT registra upisao u PC registar.



Slika 13: prikaz signala u CPLD1

4.4.2. Uvećanje PC registra za jedan

Uvećanje PC registra se jednostavno obavlja postavljanjem *inc* signala na logičku jedinicu. Potrebno je pričekati do prvog rastućeg brida, nakon čega će se sadržaj PC registra uvećati za 1.

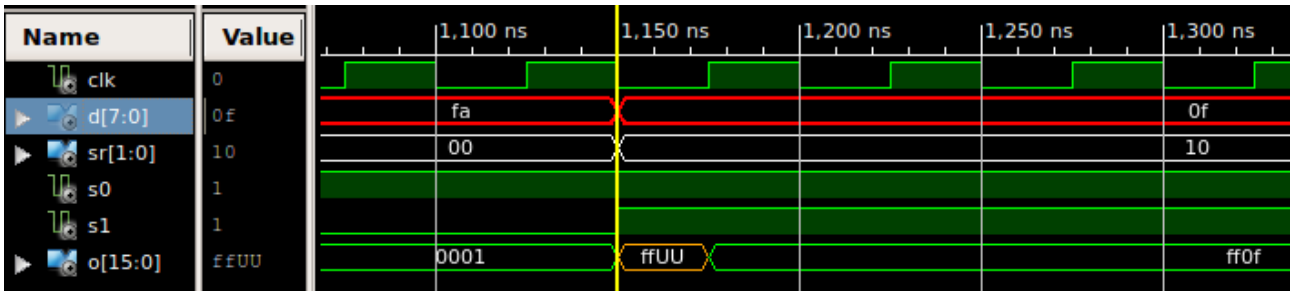
```
-- ****** uvećamo PC ****** --
inc<= '1';
wait for clk_period;
inc <= '0';
```

Na slici 13. vidljivo je postavljanje *inc* signala u $t = 400ns$ te uvećanje registra pri rastućem bridu u $t = 425ns$ sa 0xfaf0 na 0xfaf1.

4.4.3. Korištenje OFF registra

OFF registar koristimo tako da na ulazu d postavimo 8-bitni podatak te odaberemo upis u njega (sr = 10). Podatak će se na rastući brid upisati u OFF registar. Ukoliko je na adresnoj sabirnici odabran izlaz iz OFF registra, tada će se prvih 8 nižih bitova uzeti iz njega, dok će se viših 8 bitova postaviti na logičku jedinicu.

```
-- ****** korišenje offset registra ****** -
sr <= "10"; -- odaberemo OFF registar
d <= "00001111"; -- postavimo 8 bitni podatak
-- odaberemo OFF na izlazu
s0 <= '1';
s1 <= '1';
```



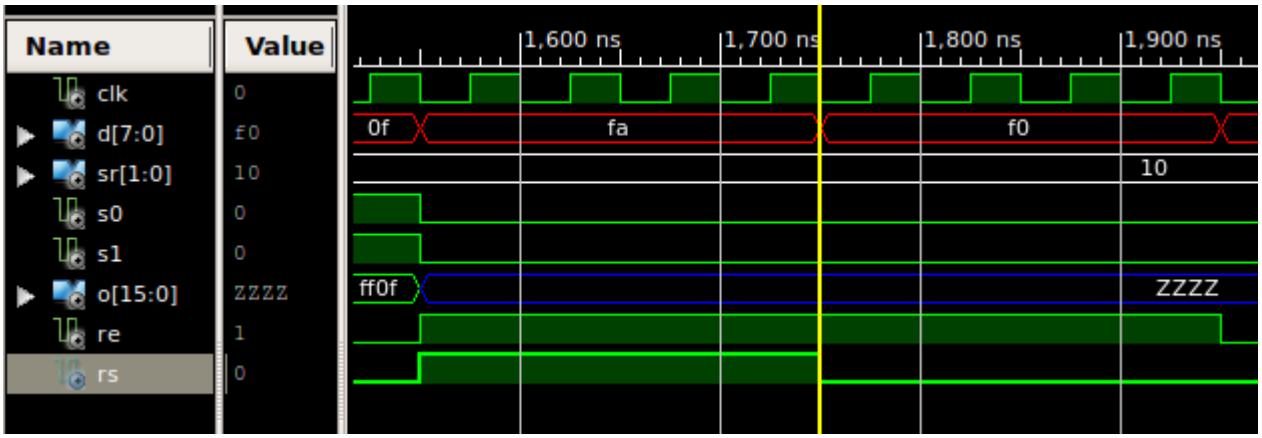
Slika 14: korištenje OFF registra

Na slici 14. vidimo da je u *t = 1150ns* postavljen podatak 0x0f na ulaz d te odabran registar OFF (sr = 10). Također, na izlaz (na adresnoj sabirnici) je postavljen sadržaj OFF registra (*s0 = 1, s1 = 1*). Na rastući brid u *t = 1175ns* se podatak upisuje u registar te pojavljuje na izlazu. Obzirom da je podatak na izlazu 0xff0f, jasno je da se viših 8 bitova postavilo na logičku jedinicu.

4.4.4. Čitanje sadržaja PT registra

Da bi pročitali sadržaj PT registra, koristimo se povratnom vezom iz PT registra i multiplekserom kojim odabiremo želimo li viših osam ili nižih osam bitova. Najprije je potrebno pripaziti da ne pišemo na sabirnicu podataka d. To činimo postavljajući ju u stanje visoke impedancije. Zatim, potrebno je uključiti multiplekser koji će propustiti podatak na sabirnicu d.

```
-- ****** čitanje sadržaja PT registra ***** --
d <= "ZZZZZZZZ"; -- moramo se odspojiti od sabirnice da bi cpld1 mogao pisati
rs <= '1'; -- čitanje viših 8 bitova (PTlow)
re <= '1'; -- enableanje povratne veze
wait for 200ns;
rs <= '0'; -- čitanje nižih 8 bitova (PThigh)
wait for 200ns;
re <= '0';
```



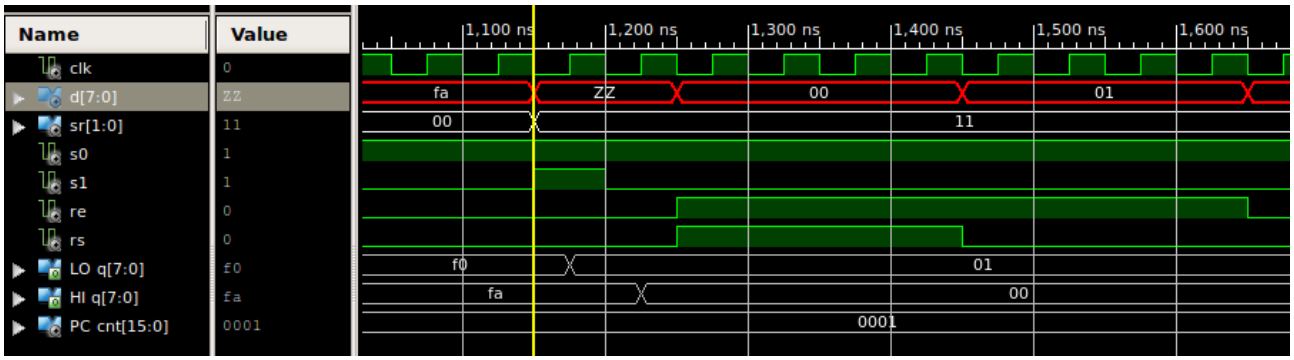
Slika 15: čitanje sadržaja PT registra

Multiplekser povratne veze je posebno izrađen tako da sadrži ulaz omogućavanja *E* te *three-state-buffer* na izlazu, da bi mogao preći u stanje visoke impedancije kada ga ne koristimo (u protivnom ne bi mogli koristiti sabirnicu d kao ulaz u cpld1, što smo i radili u dosadašnjim primjerima). Na slici 4. vidimo da smo u trenutku *t = 1550ns* omogućili povratnu vezu postavljanjem *re = 1*. Sa *rs = 1* su odabrani nižih osam bitova, koji se odmah pojavljuju na sabirnicu d (0xfa). U trenutku *t = 1750ns* smo sa *rs = 0* odabrali viših 8 bitova, koji se odmah pojavljuju na sabirnici d (0xf0). Na kraju rada, potrebno je isključiti povratnu vezu (*re = 0*), čime se multiplekser povratne veze vraća u stanje visoke impedancije te ne smeta normalnom radu podatkovne sabirnice.

4.4.5. Čitanje sadržaja PC registra

PC registar ne možemo očitati direktno, već moramo prvo prebaciti njegov sadržaj u PT registar. To činimo odabirom izlaza PC registra na ulaznim multiplekserima PT registra (*s0 <= 1*) te slanjem `store` signala u PT registre (*sr = 11*) i to prvo u PTLow (*sr = 1)*, a zatim u PTHigh (*sr = 0*). Ovo bi se moglo izvesti u jednom koraku, međutim zbog ograničenog broja kontrolnih *pinova* ovakva je izvedba bila neizbježna. U *t = 1225ns* vidimo da su registri PTLow i PTHigh poprimili vrijednost PC registra. U sljedećem koraku pošaljemo njihove sadržaje na sabirnicu d, na način koji je opisan u prethodnom dijelu.

```
-- **** prebacivanje PC registra u PT **** ---
s0 <= '1'; -- ulaz iz sg1 (povratni put iz PC-a)
sr <= "11"; -- pisanje u PT
s1 <= '1';  -- zapisujemo nižih 8 bitova
wait for clk_period;
s1 <= '0'; -- zapisujemo viših 8 bitova
wait for clk_period;

-- ****** čitanje sadržaja PT registra ***** --
d <= "ZZZZZZZZ"; -- moramo se odspojiti od sabirnice da bi cpld1 mogao pisati
rs <= '1'; -- postavljamo na d viših 8 bitova
re <= '1';
wait for 200ns;
rs <= '0'; -- postavljamo na d nižih 8 bitova
wait for 200ns;
re <= '0';
```



Slika 16: čitanje sadržaja PC registra

5. Instrukcijska arhitektura i upravljačka jedinica

Do sada je pokazana opća struktura procesora, način rada aritmetičko-logičke jedinice, pojedinih registara, put podataka te upravljanje adresnom sabirnicom. Sada je potrebno svim tim elementima smisleno i sinkronizirano upravljati. Tu zadaću naravno preuzima upravljačka jedinica procesora. Nju smo modelirali u drugom CPLD-u te ima definirana sljedeća sučelja (veze prema pojedinim elementima procesora):

```
entity cpld2 is -- upravljacka jedinica --
    port(

        clk : in std_logic;
        rst : in std_logic;
        err : out std_logic;

        -- upravljacki signali za cpld1
        sr : out std_logic_vector(1 downto 0);
        s0 : out std_logic := '1';
        s1 : out std_logic := '0';
        incPC : out std_logic;
        rstPC : out std_logic;
        re : out std_logic;
        rs : out std_logic;

        -- upravljanje memorijom
        nOE : out std_logic;
        nWE : out std_logic;

        -- upravljanje A registrom
        WA : out std_logic;

        -- upravljanje IR registrom
        WIR : out std_logic;

        -- upravljanje TMP registrom
        nET : out std_logic; -- enable output
        WT : out std_logic; -- write

        -- suceljavanje ALU-a
        cIn : out std_logic; -- carry
        comp1 : in std_logic; -- comparator output (hi and low alu)
        comp2 : in std_logic;
        cOut  : in std_logic; -- carry in

        data_dir : out std_logic;
--        addr_HiZ : out std_logic; -- za sada je ovo highwireano jer nemamo potrebe stavljati adresnu sabirnicu u HighZ
        -- ako u buducnosti budemo dodali hardware koji recimo puni program u ram iz sd kartice, tada ce trebati odlemiti
        -- signal sa GND-a te ga ovdje ukljuciti u reset fazi
        nMREQ :  out std_logic;

        -- opcode iz IR
        opcode : in std_logic_vector (2 downto 0);

        -- alu opt type (arith/logic)
        opt : in std_logic


    );
end cpld2;
```

U kontrolnoj jedinici dolaze četiri signala iz instrukcijskog registra. To su signali *opcode* (najniža tri bita instrukcijskog registra) i *opt* (četvrti bit instrukcijskog registra) koji zajedno određuju koja će se

vrsta instrukcije obaviti. *opt* signal je zapravo i *mode select* bit arimetičko-logičke jedinice. Ukoliko se ne radi o ALU operaciji (operaciji koja zahjteva da se podatci obrade u arimetičko-logičnoj jedinici), tada se taj bit koristi zajedno s *opcode* signalima za određivanje instrukcije koje će se izvršiti. Kada je riječ o ALU operaciji, tada *opt* bit (4. bit) koristi aritmetičko-logičku operaciju za odabiranje načina rada (aritmetički ili logički način rada). Ostalih 4 bita (najviših 4 bita) instrukcijske riječi se dovodi isključivo na aritmetičko-logičku jedinicu te one definiraju operaciju koju će se izvesti nad podatcima. Prisjetimo se još jednom kako je definirana kodna riječ:

7 6 5 4	3	2 1 0
ALU operacija	ALU mode/ kod naredbe	kod naredbe

Jasno je da će se najviših 5 bitova stalno dovoditi na aritmetičko-logičku jedinicu te da će ona stalno obavljati nekakve operacije nad ulazim podatcima. No ukoliko kôd instrukcije, odnosno najniža tri bita, određuju da se ne radi o operaciji koja zahjteva operaciju nad podatcima (ALU operaciji), tada se podatci na izlazu iz ALU jedinice jednostavno ne uzimaju u obzir te se ne spremaju u TMP registar. Prilikom odabira operacijskog koda instrukcije bilo je potrebno paziti da treći bit koda bude u skladu sa potrebnim načinom rada ALU jedinice ukoliko se ona koristi nad podatcima (pa makar samo za prijenos podataka iz akumulatora u memoriju). Prvo su definirane instrukcije koje koriste ALU, a zatim su na ostalim pozicijama postavljene ostale instrukcije koje ne koriste ALU i 3. bit instrukcije ne igra nikakvu ulogu u njihovom izvršavanju.

Kod instrukcija koje koriste aritmetičko-logičku jedinicu samo kao put podataka za upis vrijednosti akumulatora u programske registre (R0-R255 dostupnih preko OFF registra), u PT registar ili na memorijskoj lokaciji na koju PT registar pokazuje, ALU se postavlja u logički način rada sa funkcijskim kodom `1111`. U tom stanju aritmetičko-logička jedinica samo propušta A ulaz na izlaz (u našem je slučaju na A ulaz aritmetičko-logičke jedinice spojen akumulator). Izlaz ALU-a se tada sprema u privremeni registar TMP, nakon čega se upisuje na potrebno mjesto (bilo unutar PT registra ili na nekoj od memorijskih lokacija).

U ostalim slučajevima zapravo nije niti bitno koji su 4 najviši bitovi operacijske riječi, odnosno koji se funkcijski kod dovodi na aritmetičko-logičku jedinicu pošto se njezin izlaz ne koristi. No, radi preglednosti su na tim bitovima tada postavljene logičke nule.

Prije nego krenemo na detaljno razmatranje svake pojedine instrukcije koju kontrolna jedinica poznaje, u nastavku ćemo navesti pojedine instrukcije te njihove kodove.

Operacijski kod								**argument**	**opis**
ALU oper.				M/O	opcode				
7	**6**	**5**	**4**	**3**	**2**	**1**	**0**		
0	0	0	0	0	0	0	0	konstanta	A ← const
0	0	0	0	1	0	0	0	broj registra	A ← Rn
x	x	x	x	x	0	0	1	broj registra	Rn ← ALU_Operacija(A, Rn)
0	0	0	0	0	0	1	0	-	A ←PTLow
1	1	1	1	1	0	1	0	-	PTLow ← A
0	0	0	0	0	0	1	1	-	A ← PTHigh
1	1	1	1	1	0	1	1	-	PTHigh ← A
0	0	0	0	0	1	0	0	-	A ← (PT)
1	1	1	1	1	1	0	0	-	(PT) ← A
0	0	0	0	0	1	0	1	-	PC ← PT
0	0	0	0	1	1	0	1	-	PT ← PC
0	0	0	0	0	1	1	0	-	PC ← PT if equal
0	0	0	0	1	1	1	0	-	PC ← PT if carry
0	0	0	0	0	1	1	1	-	clear carry
0	0	0	0	1	1	1	1	-	set carry

Tablica je tako formirana da je u svim slučajevima kada se podatak iz akumulatora "provlači" kroz aritmetičko-logičku jedinicu ona ne mijenja podatak. Vidi se da je u svim tim slučajevima 3. bit (*ALU mode*) jednak 1 te da su bitovi 7 do 4 (*ALU Operation*) također jednaki jedinicama. Ukoliko pogledamo funkcijsku tablicu iz poglavlja aritmetičko-logičke jedinice (slika 3.) vidimo da ona na izlazu daje prvi argument, što je s obzirom na način na koji smo spojili ALU, upravo akumulator. Nadalje, prilikom izvođenja operacije Rn ← ALU_Operacija(A, Rn) se bitovi 7 do 4 postavljaju ovisno o operaciji koju želimo da aritmetičko-logička jedinica izvrši. Postavljanje tih bitova će napraviti asembler, a sama upravljačka jedinica ne poznaje razliku između različitih aritmetičko-logičkih operacija. Ona samo priprema podatke na ulazu aritmetičko-logičke jedinice, te ih nakon što se podatak na izlazu stabilizira spremi na potrebnu lokaciju. Štoviše, upravljačka jedinica niti nema uvid u bitove operacijskog koda koji definiraju rad aritmetičko-logičke jedinice. Operacijski je kod duljine 8 bita, no neke instrukcije imaju i argument koji može biti 8 bitna konstanta ili indeks općeg programskog registra (čiji je rad pomoću OFF registra već opisan). Takve se konstante onda zapisuju u sljedećih 8 bitova te će ih kontrolna jedinica dohvatiti iz memorije ukoliko vidi da se radi o instrukcijama kod kojih je takvo što

nužno. U nastavku će biti opisane pojedine faze svih instrukcija.

U općem slučaju bi se upravljačka jedinica sastojala od instrukcijskog dekodera te generatora slijeda. Instrukcijski dekoder bi na temelju bitova kodne riječi aktivirao jedan signal koji bi označavao trenutnu instrukciju. Na kraju bi se za pojedine signale slijeda i instrukcijske signale generirali upravljački signali povezivanjem AND logičkom operacijom (primjerice: ukoliko se radi o 3. instrukciji i 5. slijedu, treba zapisati podatak u memoriju postavljanjem `write` signala). No, obzirom da naš procesor ima samo četiri bitova kodnih riječi, odlučeno je ne modelirati instrukcijski dekoder već samo uspoređivati pojedine bitove brojača sljedova i operacijskog koda. Time se troši nešto više AND *gateova* jer se provjerava više signala nego da je svaka instrukcija i faza predstavljena samo jednim signalom, no to nije problem jer je kapacitet CPLD-a svejedno dovoljan te je iskorištenost makroćelija na kraju dosta daleko od maksimuma. Također, umjesto pisanja velikog broja *and* operatora i strukturalnog modeliranja upravljačke jedinice u VHDL-u, odabrano je ponašajno opisivanje koristeći se *case statementom* i *if* naredbama. Obzirom na veličinu i broj radnji koje obavalja upravljačka jedinica time je znatno ubrzano pisanje VHDL modela, a nakon sintetizacije ionako nema razlike u radu i iskorištenosti sklopa.

Brojač sekvenci definiran je kao signal te se on uvećava prilikom svake faze. Isto tako definirani su interni signali `carry` i `equal`. Oni pamte zastavice nakon odrađene aritmetičko-logičke operacije. Ukoliko se kasnije pojavi instrukcija uvjetnog skoka, ona tada provjerava te zastavice pa izvrši ili ne izvrši skok ovisno o njihovom stanju.

```
signal mc: unsigned(4 downto 0) := "00000"; -- sequence generator
signal carry : std_logic := '1'; -- bistabili koji pamte carry i equal (default = H, vidi alu datasheet)
signal equal : std_logic := '0'; -- flagove iz prethodne naredbe (koriste se za uvjetni skok)
```

Važno je pritom na početku definirati početnu vrijednost brojila sekvenci. Također, bitno je brojilo sekvenci vratiti na početnu vrijednost prilikom reseta procesora te na kraju zadnje faze svake instrukcije. Ukoliko se to ne napravi ne bi postojala nikakva garancija u kom stanju će se brojilo sekvenci naći, te procesor ne bi ispravno radio. Nadalje, brojilo sekvenci se povećava na svaki rastući brid signala takta. Na početku, kada je brojilo sekvenci na početnoj vrijednosti izvršava se faza pribavi. Ona dohvaća operand iz memorije, uveća programsko brojilo (registar PC) te zapisuje operacijski kod u instrukcijski registar. Nakon toga se na temelju bitova instrukcijskog registra koji određuju operaciju izvršava faza pribavi pripadne instrukcije. U nastavku ovog poglavlja biti će detaljno opisani pojedini koraci faze pribavi te pojedinih instrukcija.

5.1. Faza pribavi

Faza pribavi treba dohvatiti podatak iz memorije sa adrese koju sadrži programsko brojilo (PC registar) te ju zapisati u instrukcijski registar (IR). To čini u sljedećim koracima:

- Najprije pošalje signal prvom CPLD-u da postavi sadržaj PC registra na adresnu sabirnicu. To čini postavljanjem signala *s0 = 1* i *s1 = 0*. Kako IR registar zapisuje na rastući brid, potrebno je prije držati njegov signal na logičku nulu *WIR = 0* (*Write IR*). Također, potrebno je uključiti izlaz memorije *nOE = 0 (OE = 1)* te postaviti memoriju na čitanje sa *nWE = 1*. O bitu koji označava smjer pojačanja će biti riječ u poglavlju koji se bavi izradom sklopovlja, te se za sada može zanemariti.
- Nakon što se izlaz memorije stabilizira stavi se signal instrukcijskog registra na logičku jedinicu *WIR = 1*. Prijelazom signala iz logičke nule generira se rastući brid prilikom kojeg se podatak zapisuje u registar.
- Nakon obavljenog zapisa u instrukcijski registar *WIR* signal se vraća na logičku nulu. Tada se postavi signal *incPC* na logičku jedinicu.
- Na prvi sljedeći rastući brid signala takta će se PC registar uvećati za jedan. Stoga se nakon sljedećeg takta *incPC* vraća na logičku nulu (da to ne učinimo, PC registar bi se nastavljao uvećavati prilikom svakog rastućeg brida signala takta, što bi dovelo do nepoželjnog ponašanja procesora).

```
case mc is
        when "00000" => -- **** faza PRIBAVI *** ---
                nMREQ <= '1'; -- MREQ = 0
                data_dir <= '0'; -- pojacanje (na bufferu) u smjeru citanja iz mem
                -- postavimo sadrzaj PC-a na adresnu sabirnicu
                s0 <= '1';
                s1 <= '0';
                WIR <= '0'; -- write IR u 0 (da kasnije mozemo generirati rising edge)

                -- procitaj podatak iz rama --
                nOE <= '0'; -- output enable = 1
                nWE <= '1'; -- write enable = 0

                re <= '0';
                rs <= '0';
                mc <= mc + 1;
        when "00001" =>  -- PRIBAVI: upis u IR
                WIR <= '1';
                mc <= mc + 1;
        when "00010" => -- PRIBAVI: gotov upis u IR
                WIR <= '0';
                incPC <= '1'; -- povecaj PC
                mc <= mc + 1;
        when "00011" =>
                incPC <= '0'; -- ovdje je incPC = 1 za vrijeme rising edgea clocka
                mc <= mc + 1;
```

```
-- == gotova faza PRIBAVI == --
when others =>
-- == faza IZVRSI == --
```

Time završava faza pribavi. Ona se uvijek obavlja na početku svakog ciklusa, te ne ovisi o instrukciji koja se izvršava. Nakon što je faza pribavi dovela instrukcijski kod u IR registar, upravljačka jedinica ima uvid u operacijski kod instrukcije (pošto dobiva bitove iz instrukcijskog registra), te može započeti sa dekodiranjem i izvršavanjem pojedinih instrukcija.

5.2. Faza izvrši za *A ← const*

Prva po redu instrukcija puni akumulator 8-bitnom konstantom koja se nalazi u memoriji, odmah nakon operacijskog koda. Ova je instrukcija jako slična fazi pribavi.

- Kako je faza pribavi povećala programsko brojilo za jedan, ono sada pokazuje na konstantu koju treba upisati u akumulator (registar A). Prvo je potrebno javiti prvom CPLD-u da na adresnu sabirnicu postavi PC registar, to se čini postavljanjem signala *s0 = 1* i *s1 = 0*. Zatim, potrebno je *write* signal na akumulatoru postaviti na logičku nulu kako bi se kasnije mogao generirati rastući brid (*WA = 0*). Na kraju, potrebno je opet uključiti izlaz memorije *nOE = 0 (OE = 1)* te postaviti memoriju na čitanje sa *nWE = 1*.
- Nakon što se podatak na izlazu iz memorije stabilizirao, *write* signal na akumulatoru postavimo na logičku jedinicu, čime generiramo rastući brid, prilikom kojeg se podatak zapiše u akumulatoru.
- Na kraju se *write* signal akumulatora vraća na nulu te se na isti način kao i prije poveća programsko brojilo, kako bi ono pokazivalo na sljedeću instrukciju.

```
case opcode is
    when "000"  =>
        if opt = '0' then
        -- **** A <- CONST *** ---
            case mc is
                when "00100" => -- dohvacanje konstante
                    --postavi PC na Abus
                    s0 <= '1';
                    s1 <= '0';
                    WA <= '0';
                    data_dir <= '0'; -- pojacanje (na bufferu) u smjeru citanja
                    nOE <= '0'; -- output enable = 1
                    nWE <= '1'; -- write enable = 0

                    re <= '0';
                    rs <= '0';
                    mc <= mc + 1;
                when "00101" => -- upis u A
                    WA<= '1';
                    mc <= mc+1;
                when "00110" => -- kraj upisa u A
                    WA <= '0';
                    incPC <= '1';
```

```
            mc <= mc+1;
        when "00111" =>
            incPC <= '0';
            mc <= "00000"; -- gotovo
        when others =>
            c <= "00000";
    end case;
-- ****END  A <- CONST *** ---
```

5.3. Faza izvrši za *A ← Rn*

Ova instrukcija dohvaća podatak iz jednog od 256 programskih registara kojima se interno pristupa pomoću OFF registra, te ga sprema u akumulator (registar A). U ovom se slučaju, nakon operacijskog koda, u memoriji nalazi 8-bitni indeks registra kojeg je potrebno upisati u OFF registar.

- Programsko brojilo već pokazuje na 8-bitni *offset*. Stoga je dovoljno samo postaviti PC na adresnu sabirnicu, postaviti memoriju na čitanje te uključiti njezin izlaz. To se radi na identični način kao i u prethodnim instrukcijama. Izlaz memorije i ulaz prvog CPLD-a su povezani podatkovnom sabirnicom (*dbus*).
- Nakon što je podatak stabilan, na izlazu se postavi *write* signal OFF registru. To se učini postavljanjem *sr* (*select register*) na vrijednost *10*. Prilikom prvog rastućeg brida signala takta će podatak biti upisan u OFF registar. Nakon toga se *sr* vrati na vrijednost *00*. Da to ne učinimo, OFF registar bi prilikom svakog rastućeg brida takta zapisao podatak sa podatkovne sabirnice.
- Sada kada je indeks spremljen u OFF registar, potrebno ga je postaviti na adresnoj sabirnici. To se čini postavljanjem signala `s1 = 1` te `s0 = 1` prvom CPLD-u. Memoriju stavljamo (tj. i dalje držimo) u stanje čitanja. Nakon nekog vremena vrijednost sa programskog registra na koji OFF pokazuje će se pojaviti na podatkovnu sabirnicu. U međuvremenu na akumulator postavimo `WA = 0` da bi kasnije mogli generirati rastući brid.
- Nakon što se podatak stabilizira na podatkovnoj sabirnici postavimo `WA = 1` čime nastaje rastući brid koji će zapisati podatak u akumulator.
- Na kraju, isto kao i u prethodnoj instrukciji, potrebno je vratiti *WA* na *0* te povećati programsko brojilo za jedan (što činimo u svim instrukcijama kod kojih se nakon operacijskog koda nalazi argument – ukoliko instrukcija nema argument tada programsko brojilo već pokazuje na operacijski kod sljedeće instrukcije).

```
-- **** A <- Rn *** ---
case mc is
    when "00100" =>
        s0 <= '1';
```

```
            s1 <= '0';
            WA <= '0';
            data_dir <= '0'; -- pojacanje (na bufferu) u smjeru citanja iz mem
            nOE <= '0'; -- output enable = 1
            nWE <= '1'; -- write enable = 0

            re <= '0';
            rs <= '0';

            mc <= mc + 1;
        when "00101"=>
            sr<= "10";
            mc <= mc + 1;
        when "00110"=>
            sr<= "00";
            mc <= mc + 1;
        when "00111"=>
            s1<='1';
            s0<='1';
            data_dir <= '0'; -- pojacanje (na bufferu) u smjeru citanja iz mem
            nOE <= '0'; -- output enable = 1
            nWE <= '1'; -- write enable = 0

            mc<=mc+1;
        when "01000"=>
            WA<= '1';
            mc<=mc+1;
        when "01001"=>
            WA<= '0';
            incPC <= '1';
            mc<=mc+1;
        when "01010"=>
            incPC<= '0';
            mc<= "00000";
        when others =>
            mc <= "00000";
    end case;
    -- ****END  A <- Rn *** ---
```

5.4. Faza izvrši za *A ← ALU_Operacija(A, Rn)*

Prethodno opisana instrukcija (A ← Rn) puni akumulator s jednim operandom. Drugi operand se na aritmetičko-logičku jedinicu dovodi direktno preko podatkovne sabirnice sa jednog od 256 programskih registara u memoriji. Kako je podatkovna sabirnica zauzeta donošenjem jednog operanda na aritmetičko-logičku jedinicu, rješenje se mora spremiti u privremeni registar TMP. Nakon što se rezultat operacije spremio, oslobađa se podatkovna sabirnica te se preko nje zapisuje podatak koji je bio spremljen u TMP registru natrag u memoriju. Bitovi koji određuju koju će operaciju aritmetičko--logička jedinica obaviti dovode se direktno iz instrukcijskog registra IR te kontrolna jedinica nema nikakav uvid u to kakva se operacija obavlja nad podatcima. Stoga, tek asembler pravi razliku između pojedinih ALU operacija, dok je kontrolnoj jedinici to jedna vrsta operacije koja je za svaku ALU operaciju ista. Krenimo redom:

- Programsko brojilo (odnosno registar PC) već pokazuje na argument (indeks registra). Potrebno je na adresnu sabirnicu postaviti vrijednost PC registra (*s0 = 1, s1 = 0*) te memoriju postaviti na čitanje (*nOE = 0, nWE = 1*).
- Nakon što se podatak na izlazu iz memorije stabilizira, pošalje se *write* signal OFF registru, što se čini postavljanjem *sr = 10* signala (*select register*). Kako je izlaz iz memorije podatkovnom sabirnicom (*dbus*) direktno spojen sa ulazom u prvi CPLD te unutar njega na OFF registar, podatak se uspješno zapiše u OFF registar prilikom prvog sljedećeg rastućeg brida signala takta.
- Zatim je potrebno maknuti *write* signal sa OFF registra postavljanjem *sr* signala na *00* jer bi se inače stalno zapisivao podatak iz podatkovne sabirnice na svaki rastući brid signala takta.

- Sada kada je ispravna vrijednost spremljena u OFF registar, on se postavlja na adresnu sabirnicu postavljanjem signala *s0 = 1* i *s1 = 1*.
- Memorija se zatim postavi (tj. i dalje drži) u stanje čitanja. Podatak koji tvori drugi argument se tada dovodi preko adresne sabirnice na aritmetičko-logičku jedinicu. (Prisjetimo se da je prvi argument spremljen u registar akumulator, čiji je izlaz stalno spojen na aritmetičko-logičku jedinicu). Na aritmetičko-logičku jedinicu također dolaze bitovi iz instrukcijskog registra koji određuju operaciju koja će se izvršiti nad operandima. Osim operanada, na aritmetičko-logičku jedinicu dovodi se i ulaz *carrya (carry in)*. *U*pravljačka jedinica postavlja *carry in* bit na temelju vrijednosti iz internog registra zastavica. Sada je potrebno *write* signal TMP registra postaviti na logičku nulu da bi se u sljedećoj fazi mogao generirati rastući brid.
- Nakon što se podatak na izlazu aritmetičko-logičke jedinice (rezultat operacije) stabilizirao zapisuje se u TMP registar. To činimo postavljanjem *WTMP* signala u logičku jedinicu, prilikom čega se generira rastući brid te podatak biva zapisan. U tom trenutku upravljačka jedinica također sprema *equal* i *carry*. *Carry* signal dolazi direktno iz više aritmetičko-logičke jedinice, dok *equal* signal se zapravo tvori kao logički AND nad izlazima komparatora obiju aritmetičko- -logičkih jedinica (više i niže). Podatak je jednak samo ukoliko oba izlaza komparatora javljaju da su njihovih 4 bita jednaki. (Komparatori zapravo rade samo kada ALU obavlja oduzimanje. Tada, ukoliko je na izlazu 0xF postavlja se izlaz komparatora na logičku jedinicu.)

- Nakon što je podatak zapisan u TMP registar, potrebno je osloboditi podatkovnu sabirnicu. To se čini isključivanjem izlaza memorije `nOE = 1` *(Output Enable = 0)*.
- Zatim se uključuje izlaz TMP registra postavljanjem `nET = 0` *(Enable T = 1)*.
- Kada se vrijednost na izlazu TMP registra stabilizira, podatak se može upisati natrag u memoriju. Adresna sabirnica i dalje prikazuje sadržaj OFF registra te pokazuje na ispravnu memorijsku lokaciju (nismo ju mijenjali od prije). Zapis u memoriju činimo postavljanjem signala `nWE = 0` *(Write Enable = 1)*.
- Nakon vremena zapisivanja podatka u memoriju isključi se pisanje u memoriju vraćanjem signala `nWE` na logičku jedinicu (*Write Enable = 0)*
- Zatim je potrebno isključiti izlaz TMP registra sa `nET = 1` *(Enable T = 0)*. Da se to ne učini došlo bi do kolizije na podatkovnoj sabirnici te sustav ne bi ispravno radio.
- U drugom djelu se adresiranje izvodi koristeći se OFF registrom, tako da se programsko brojilo može povećati u bilo kojem trenutku. Prvo se `incPC` postavlja na logičku jedinicu. Prilikom sljedećeg brida takta se programsko brojilo poveća, nakon čega upravljačka jedinica mora vratiti `incPC` signal na logičku jedinicu (da se programsko brojilo ne bi nastavilo povećavati).

```
when "001" =>
-- **** R <- ALU(A, R) *** ---
    case mc is
        when "00100" => -- dohvati index (offset) od R
            --postavi PC na Abus
            s0 <= '1';
            s1 <= '0';
            WA <= '0';
            data_dir <= '0'; -- pojacanje (na bufferu) u smjeru citanja iz mem
            nOE <= '0'; -- output enable = 1
            nWE <= '1'; -- write enable = 0

            re <= '0';
            rs <= '0';

            cin <= not carry;
            mc <= mc + 1;
        when "00101" => -- zapisi ga u offset registar
            sr <= "10"; -- salje store signal offset registru (select reg = offset)
            mc <= mc+1;
        when "00110" =>
            sr <= "00"; -- kraj store signala
            incPC <= '1';
            mc <= mc+1;
        when "00111" =>
            incPC <= '0';
            s0 <= '1'; -- postavi offset registar
            s1 <= '1'; -- na adresnoj sabirnici
            mc <= mc+1;
        when "01000" =>
            mc <= mc+1; -- cekamo da se prvo izlaz memorije, a onda i ALU-a stabilizira
            WT <= '0';  -- write u TMP reg na 0 da kasnije mozemo generirati rising edge
            mc <= mc+1;
```

```
        when "01001" =>
            WT <= '1'; -- zapisi u TMP
            carry <= not cOut;
            equal <= comp1 and comp2;
            mc <= mc+1;
        when "01010" =>
            WT <= '0';
            data_dir <= '1'; -- pojacanje (na bufferu) u smjeru pisanja iz mem
            nET <= '0'; -- output enable TMP registra (tmp -> dbus)
            nOE <= '1'; -- sRAM output enable = 0
            mc <= mc+1;
        when "01011" => -- zapisi dbus u ram (offset registar je i dalje na adresnoj sabirnici)
            nWE <= '0'; -- sRAM write  = 1
            mc <= mc+1;
        when "01100" =>
            nOE <= '1'; -- disable output
            nWE <= '1'; -- SRAM write = 0
            mc <= mc+1;
        when "01101" =>
            nET <= '1'; -- TMP reg output disabled
            mc <= "00000";
        when others =>
            mc <= "00000";
    end case;
    -- ****END  R <- ALU(A, R) *** ---
```

5.5. Faza izvrši za *A ←PTLow*

Sljedeća instrukcija puni sadržaj akumulatora sa sadržajem nižih 8 bitova PT registra. To se čini preko podatkovne sabirnice koja povezuje CPLD1 sa akumulatorom, a unutar samog CPLD1 preko povratne veze PT registra (koja je inače u stanju visoke impedancije).

- Najprije je potrebno isključiti izlaz memorije postavljanjem signala *nOE = 1 (OE = 0)* te *nWE = 1 (WE = 0)*. Pošto akumulator zapisuje podatak na rastući brid, potrebno je pripremiti *WA* postavljanjem na logičku nulu. Zatim, uključujemo povratnu vezu iz PTLow registra prema podatkovnoj sabirnici (to je put podataka koji izlaz PT registra donosi na podatkovnu sabirnicu) postavljanjem signala *re = 1 (return enable)* i *rs = 0 (return select)*.
- Podatak se do sljedećeg takta stabilizira na podatkovnoj sabirnici te se može upisati u akumulator. To činimo postavljanjem *WA* u logičku jedinicu, čime se generira rastući brid te podatak biva zapisan.
- Na samom kraju, potrebno je isključiti povratnu vezu PT registra (tj. postaviti ju u stanju visoke impedancije da bi podatkovnom sabirnicom dalje mogli nesmetano "putovati" podatci prema i od memorije).

```
-- **** A <- PTlow **** --
case mc is
    when "00100"=>
    data_dir <= '1'; -- pisanje
    nOE <= '1'; -- disable output (da nam ram ne smeta u dbusu)
```

```
        nWE <= '1'; -- SRAM write = 0
        WA <= '0';
        re <= '1'; -- ukljuci povratnu vezu PT registra (PT -> dbus)
        rs <= '0'; -- odaberi pt high
        mc <= mc+1;
    when "00101" =>
        WA <= '1';
        mc <= mc+1;
    when "00110" =>
        WA <= '0';
        mc <= mc+1;
    when "00111" =>
        re <= '0'; -- iskljuci povratnu vezu PT registra
        mc <= mc+1;
    when others =>
        mc <= "00000";
    end case;
    -- **** END A <- PTlow **** --
```

Dade se primijetiti da ovdje nigdje ne povećavamo programsko brojilo. Tomu je tako iz razloga što ova instrukcija, kao ni ostale koje će biti opisane u nastavku, ne koristi nikakav argument. Iz tog razloga programsko brojilo već pokazuje na sljedeću instrukciju koju će procesor dohvatiti prilikom sljedeće faze pribavi.

5.6. Faza izvrši za *PTLow ← A*

Ova instrukcija puni PTLow registar vrijednošću iz akumulatora. Da bi to bilo moguće potrebno je "provući" podatak kroz aritmetičko-logičku jedinicu i to tako da on ostane nepromijenjen. Da bi to bilo moguće potrebno je da bitovi 7 do 4 operacijskog koda budu postavljeni u logičku jedinicu. To je logički način rada aritmetičko-logičke jedinice koji na izlazu propušta prvi operand (odnosno sadržaj akumulatora) bez da ga ikako promjeni. Podatak je zatim potrebno zapisati u TMP registar. Zatim se oslobađa podatkovna sabirnica te uključuje izlaz TMP registra. Na kraju se podatak zapisuje u PTLow registar koji je smješten unutar prvog CPLD-a.

- Prvo je potrebno isključiti izlaz memorije da bi se oslobodila podatkovna sabirnica (dbus). To se postiže postavljanjem signala `nOE = 1` (*OE = 0*) te `nWE = 1` (*WE = 0*). Kako je u sljedećem koraku potrebno podatak upisati u privremeni registar, *WT* se postavlja na logičku nulu.
- Aritmetičko-logička jedinica je postavljena u način rada u kojem propušta prvi operand, odnosno sadržaj akumulatora na izlazu. Do sljedećeg se otkucaja signala takta podatak sigurno stabilizirao pa možemo generirati rastući brid privremenom registru TMP postavljanjem *WT* na logičku jedinicu.
- Sada je potrebno podatak prebaciti iz privremenog registra u PTLow. Odabir *store* signala za PT

registar činimo postavljanjem *sr* na vrijednost *11*, *a* odabir PTLow registra postavljanjem *s1 = 1*. Na kraju postavimo da se u PTLow zapiše podatak sa podatkovne sabirnice (a ne sa povratne veze PC registra) sa *s0 = 0*.

- Tokom prvog sljedećeg rastućeg brida takta podatak će biti zapisan u PTLow registar. Na samom kraju je potrebno vratiti signale na početne vrijednosti da se ne bi podatak i dalje upisivao u registar prilikom svakog sljedećeg rastućeg brida signala takta.

```
-- **** PTlow <- A **** --
case mc is
    when "00100"=>
        cin <= '1';
        data_dir <= '1'; -- pojacanje u smjeru pisanja
        nOE <= '1'; -- disable output (da nam ram ne smeta u dbusu)
        nWE <= '1'; -- SRAM write = 0
        WT <= '0';
        mc <= mc+1;
    when "00101" =>
        WT <= '1';
        mc <= mc+1;
    when "00110" =>
        sr <= "11";
        s1 <= '1';
        s0 <= '0';

        nET <= '0'; -- ET = 1
        WT <= '0';
        mc <= mc+1;
    when "00111" =>
        mc <= mc+1;
    when "01000" =>
        sr <= "00"; -- kraj pisanja
        s1<= opt;
        mc<= mc+1;
    when "01001" =>
        nET <= '1'; -- ET = 0
        mc <= "00000";
    when others =>
        mc <= "00000";
end case;
-- **** END PTlow <- A **** --
```

5.7. Faza izvrši za *A ←PTHigh*

Ova instrukcija gotovo da je identična instrukciji *A ←PTLow,* uz jedinu razliku da se ovaj put odabire registar PTHigh postavljanjem *rs (return select)* signala na logičku jedinicu. U nastavku je naveden VHDL kod za tu instrukciju, a opis je isti opisu za instrukciju *A ←PTLow* pa ga nećemo ponavljati.

```
-- **** A <- PThi **** --
case mc is
    when "00100"=>
    data_dir <= '1'; -- pojacanje u smjeru pisanja
    nOE <= '1'; -- disable output (da nam ram ne smeta u dbusu)
    nWE <= '1'; -- SRAM write = 0
    WA <= '0';
    re <= '1'; -- ukljuci povratnu vezu PT registra (PT -> dbus)
```

```
        rs <= '1'; -- odaberi PT hi
        mc <= mc+1;
    when "00101" =>
        WA <= '1';
        mc <= mc+1;
    when "00110" =>
        WA <= '0';
        mc <= mc+1;
    when "00111" =>
        re <= '0'; -- iskljuci povratnu vezu PT registra
        mc <= mc+1;
    when others =>
        mc <= "00000";
    end case;
    -- **** END A <- PThi **** --
```

5.8. Faza izvrši za *PTHigh ← A*

Ova je instrukcija opet gotovo identična *PTLow ← A* instrukciji uz jedinu razliku da se ovaj put odabire zapis u PTHigh registar postavljanjem signala *s1 = 0*. Kod je dan u nastavku, a opis je izostavljen iz razloga što je analogan prvotnom:

```
    -- **** PThi <- A **** --
    case mc is
        when "00100"=>
            data_dir <= '1'; -- pisanje
            cin <= '1';
            nOE <= '1'; -- disable output (da nam ram ne smeta u dbusu)
            nWE <= '1'; -- SRAM write = 0
            WT <= '0';
            mc <= mc+1;
        when "00101" =>
            WT <= '1';
            mc <= mc+1;
        when "00110" =>
            sr <= "11";
            s1 <= '0';
            s0 <= '0';

            nET <= '0'; -- ET = 1
            WT <= '0';
            mc <= mc+1;
        when "00111" =>
            mc <= mc+1;
        when "01000" =>
            sr <= "00"; -- kraj pisanja
            s1<= opt;
            mc<= mc+1;
        when "01001" =>
            nET <= '1'; -- ET = 0
            mc <= "00000";
        when others =>
            mc <= "00000";
    end case;
    -- **** END PThi <- A **** --
```

5.9. Faza izvrši za *A ← (PT)*

Ova instrukcija sprema sadržaj iz memorije, na kojeg pokazuje PT registar, u akumulator. To postiže postavljanjem sadržaja PT registra na adresnu sabirnicu, čitanjem podatka i zapisivanjem u akumulator.

- Prvo se postavlja sadržaj PT registra (16-bitne vrijednosti) na adresnu sabirnicu. To se čini signalima *s1 = 1* i *s0 = 0*. Također uključimo izlaz na memoriji te ju postavimo u čitanje (*nOE = 0, nWE = 1)*. MREQ signal se koristi prilikom I/O operacija te ga za sada, kao niti *data_dir* signal, nećemo objašnjavati. Oni će biti razrađeni u kasnijim poglavljima.
- Zatim postavimo signal pisanja u akumulator na logičku jedinicu da bi kasnije mogli generirati rastući brid.
- Kada su podatci na izlazu iz memorije stabilni postavimo *WA* na logičku jedinicu. Time smo generirali rastući brid na akumulatoru te se podatak u njega zapisao.

```
-- **** A <- (PT) **** --
case mc is
    when "00100"=>
        data_dir <= '0'; -- pojacanje (na bufferu) u smjeru citanja iz mem
        s1 <= '1'; -- postavlja sadrzaj PT registra
        s0 <= '0'; -- na adresnu sabirnicu
        nOE <= '0'; -- sRAM output enable = 1
        nWE <= '1'; -- sRAM write = 0
        nMREQ <= '0'; -- MREQ = 1;
        mc <= mc+1;
    when "00101" =>
        WA <= '0';
        mc <= mc+1;
    when "00110" =>
        WA <= '1'; -- zapis u A
        mc <= mc+1;
    when "00111" =>
        WA <= '0'; -- kraj zapisa u A
        nMREQ <= '1'; -- MREQ = 0;
        mc <= "00000";
    when others =>
        mc <= "00000";
    end case;
-- **** END A <- (PT) **** --
```

5.10. Faza izvrši za *(PT) ← A*

Ova instrukcija sprema sadržaj akumulatora na memorijsku lokaciju na koju pokazuje registar PT. Kod ove se instrukcije podatak "provlači" kroz aritmetičko-logičku jedinicu, pa bitovi operacijskog koda koji određuju ALU operaciju moraju biti takvi da se podatak ne promjeni (Bitovi 7 do 4 moraju biti u logičkoj jedinici, kao što je do sada već mnogo puta spomenuto). Sadržaj PT registra se opet postavlja na adresnu sabirnicu, a memorija ovaj put zapisuje podatak sa podatkovne sabirnice, za razliku od prošle instrukcije gdje je memorija postavljala podatak na njoj.

- U prvom se koraku signalima *s1 = 1* te *s0 = 0* sadržaj PT registra postavlja na adresnu sabirnicu. Također, isključuje se izlaz memorije.
- U drugom koraku, kada je izlaz memorije već stabilan podizanjem *WT* signala iz logičke nule u logičku jedinicu stvaramo rastući brid kojim zapisujemo podatak sa izlaza iz aritmetičko-

-logičke jedinice u privremeni registar TMP. Kako je aritmetičko-logička jedinica tako postavljena da samo propušta prvi operand, u TMP registru se spremio sadržaj akumulatora.

- Sada kada je podatak u TMP registru trebamo vratiti *WT* signal na logičku nulu te postaviti memoriju na pisanje (`nWE = 0`). Podatak se sada zapisuje u memoriju na lokaciji na koju pokazuje PT registar.
- Na kraju, potrebno je prvo prekinuti pisanje u memoriju (*`nWE = 1`*) te zatim isključiti izlaz TMP registra (*`nET = 1`*).

```
-- **** (PT) <- A **** --
case mc is
    when "00100"=>
        WT <= '0';
        s1 <= '1'; -- postavlja sadrzaj PT registra
        s0 <= '0'; -- na adresnu sabirnicu
        nOE <= '1'; -- sRAM output enable = 0
        mc <= mc+1;
    when "00101" =>
        WT <= '1'; -- zapis u TMP
        data_dir <= '1'; -- pojacanje (na bufferu) u smjeru pisanja u mem
        nMREQ <= '0'; -- MREQ = 1;
        mc <= mc+1;
    when "00110" =>
        WT <= '0'; -- kraj zapisa u TMP
        nET <= '0'; -- TMP output enable
        nWE <= '0'; -- sRAM write  = 1;
        mc <= mc+1;
    when "00111" =>
        nWE <= '1'; -- sRAM write = 0;
        nMREQ <= '1'; -- MREQ = 0;
        mc <= mc+1;
    when "01000" =>
        nET <= '1'; -- iskljucen output TMp registra (da ne smeta na dbusu
        data_dir <= '0'; -- pojacanje (na bufferu) u smjeru citanja iz mem
        mc <= "00000";
    when others =>
        mc <= "00000";
    end case;
-- **** END (PT) <- A **** --
```

5.11. Faza izvrši za *PC ← PT*

Kod naredbi skoka potrebno je sadržaj PT registra, u kojem je spremljena adresa, upisati u PC registar. Nakon toga će u sljedećoj fazi pribavi procesor dohvatiti instrukciju s te nove adrese. U ovom slučaju radi se o instrukciji bezuvjetnog skoka.

- Najprije se postavlja *`store`* signal PC registra odabirom *`sr = 01`* signala. Kako je ulaz PC registra direktno spojen na izlaz PT registra, nije potrebno ništa drugo napraviti.
- Na prvi rastući brid signala takta će PC registar upisati podatak.
- Na kraju, potrebno je vratiti *sr* na vrijednost *00* te time isključiti *`store`* signal PC registru.

```
-- **** PC <- PT **** --
case mc is
```

```
    when "00100"=>
    sr <= "01"; -- store signal PC-u
    mc <= mc+1;
when "00101" =>
    sr <= "01"; -- cekamo (na rastuci brid ce PC zapisati sadrzaj PT-a)
    mc <= mc+1;
when "00110" =>
    sr <= "00"; -- done
    mc <= "00000";
when others =>
    mc <= "00000";
end case;
-- **** END PC <- PT **** --
```

5.12. Faza izvrši za *PT ← PC*

Registar PC je 16-bitni te stoga nije ga moguće direktno pročitati. Umjesto toga, moguće je prebaciti sadržaj PC-a na PT registar koji se može onda zasebno čitati u dvije grupe od po 8 bitova. Ova instrukcija upravo to i radi: prebacuje sadržaj PC registra u PT registar kroz povratnu vezu PC registra. Prisjetimo se da su na ulazu PT registara postavljeni multiplekseri koji omogućavaju biranje hoće li se na ulaz PT registara dovesti podatak sa podatkovne sabirnice ili sa povratne veze PC registra. Razlog postavljanja tih multipleksera je omogućavanje ove instrukcije.

- Najprije je potrebno isključiti izlaz memorije (`nOE = 1`) da bi se oslobodila podatkovna sabirnica. Zatim, potrebno je odabrati zapis u registar PT postavljanjem `sr = 11`. Također, sa `s0 = 1` odabire se izlaz PC registra, preko multipleksera na ulazu PT registara.
- Postavljanjem `s1 = 1` odabire se zapis nižih 8 bitova.
- Nižih 8 bitova će biti zapisano u PTLow prilikom prvog sljedećeg rastućeg brida takta.
- Zatim, postavi se `s1 = 0`, čime se odabire viših 8 bitova.
- Nakon sljedećeg rastućeg brida signala takta će i oni biti zapisani u PTHigh registar.

Postavlja se pitanje kako to da se oba podataka ne zapišu istovremeno s obzirom da su oba registara 16--bitna te je i povratna veza širine 16 bita. Razlog tomu je ograničeni broj kontrolnih signala (zbog ograničenog broja *pinova* CPLD-a). Da bi se omogućio istovremeni upis, bilo bi potrebno odvojeno "izvući" pojedine `store` signale izvan prvog CPLD-a, što nije moguće (već se, kao što smo prije objasnili, bili primorani koristiti dekodere unutar prvog CPLD-a kojima smanjujemo broj potrebnih signala za upravljanje).

```
-- **** PT <- PC **** --
case mc is
    when "00100"=>
        nOE <= '1'; -- OE = 0 da nam ram ne smeta (ovo ovdje i nije nuzno zapravo)
        s0 <= '1'; -- ulaz u PT je povratna veza sa PC-a (ne dbus)
        sr <= "11"; -- store signal PT registru
```

```
            s1 <= '1'; -- zapisujemo nizih 8 bitova
            mc <= mc + 1;
        when "00101" =>
            mc <= mc + 1;
        when "00110" =>
            s1 <= '0'; -- zapisujemo visih 8 bitova
            mc <= mc + 1;
        when "00111" =>
            mc <= mc + 1;
        when "01000" =>
            sr <= "00";
            mc <= "00000"; -- kraj
        when others =>
            mc <= "00000";
        end case;
    -- **** END PT <- PC **** --
```

5.13. Faza izvrši za *PC ← PT if equal*

Ova je instrukcija gotovo identična instrukciji *PC ← PT,* uz jedinu razliku što provjerava da li je postavljena *equal* zastavica. Ukoliko je, prenese sadržaj PT-a u PC registar, a ukoliko nije, ne obavlja ništa. Time je omogućeno uvjetno grananje odnosno skok ukoliko su akumulator i neki registar jednaki. Da bi se *equal* zastavica zapisala, potrebno je prije toga obaviti aritmetičku operaciju. Kada je rezultat operacije 0xFFFF, tada aritmetičko-logička jedinica postavlja *equal* bit, te ga kontrolna jedinica sprema u svojoj internoj zastavici. Ostale instrukcije ne utječu na zastavicu *equal* te je stoga moguće između aritmetičko-logičke operacije i same instrukcije uvjetnog skoka (koja testira zastavicu) postaviti druge instrukcije, međutim pri tome valja paziti.

```
    -- **** PC <- PT if equal **** --
    if (equal = '1') then
        case mc is
            when "00100"=>
                sr <= "01"; -- store signal PC-u
                mc <= mc+1;
            when "00101" =>
                sr <= "01"; -- cekamo (na rastuci brid ce PC zapisati sadrzaj PT-a)
                mc <= mc+1;
            when "00110" =>
                sr <= "00"; -- done
                mc <= "00000";
            when others =>
                mc <= "00000";
            end case;
    else
        mc <= "00000";
    end if;
    -- **** END PC <- PT if equal **** --
```

5.14. Faza izvrši za *PC ← PT if carry*

Ovdje se radi o još jednoj varijanti instrukcije skoka. Ovaj se put međutim provjerava *carry* zastavica. Vrijedi sve isto kao i za prethodnu instrukciju.

```
-- **** PC <- PT if carry**** --
if (carry = '1') then
    case mc is
        when "00100"=>
            sr <= "01"; -- store signal PC-u
            mc <= mc+1;
        when "00101" =>
            sr <= "01"; -- cekamo (na rastuci brid ce PC zapisati sadrzaj PT-a)
            mc <= mc+1;
        when "00110" =>
            sr <= "00"; -- done
            mc <= "00000";
        when others =>
            mc <= "00000";
    end case;
else
    mc <= "00000";
end if;
-- **** END PC <- PT if carry **** --
```

5.15. Faza izvrši za *clear carry*

Instrukcija *clear carry* briše zastavicu prijenosa (*carry)* odnosno postavlja ju u logičku nulu. Ta zastavica se zatim postavlja na *carry in* ulaz aritmetičko-logičke jedinice prilikom izršavanja *Rn ← ALU_Operacija(A, Rn)* instrukcije.

```
-- **** set carry  **** --
case mc is
    when "00100"=>
        carry <= '1';
        mc <= "00000";
    when others =>
        mc <= "00000";
    end case;
-- **** END set carry  **** --
```

5.16. Faza izvrši za *set carry*

Instrukcija *set carry* postavlja zastavicu prijenosa (*carry)* u logičku jedinicu.

```
-- **** clear carry**** --
case mc is
    when "00100"=>
        carry <= '0';
        mc <= "00000";
    when others =>
        mc <= "00000";
end case;
-- **** END clear carry **** --
```

5.17. Zaključno o upravljačkoj jedinici

Drugi CPLD unutar kojeg je sintetizirana upravljačka jedinica iskorištava 66% makroćelija (47/72), 46% registara (33/72) te 80% *pinova* (27/34). Iz toga vidimo da je kontrolna jedinica zapravo zauzela puno manje prostora nego prvi CPLD. Također, eventualno je kupljeni model CPLD-a malo predimenzioniran te se mogao odabrati slabiji model. Međutim, smatramo da je bolje da smo koristili isti model CPLD-ova u oba slučajeva, a time je i ostavljeno mjesta za moguća buduća proširenja.

6. Cjelokupna simulacija

Radi potreba testiranja upravljačke jedinice i cjelokupnog sustava napravljen je VHDL model koji povezuje sve do sada obrađene cjeline i povezuje ih u smislenu cjelinu. Tokom ovog rada najprije smo modelirali registre, zatim smo napravili model aritmetičko-logičke jedinice te smo na kraju napisali kod za dva CPLD-a. Prvi koji implementira PC, PT i OFF registre te rukuje adresnom sabirnicom i drugi, koji obavlja posao upravljačke jedinice. Od toga dva CPLD-a predstavljaju ključne elemente u samoj izradi procesora, a VHDL modeli registara i aritmetičko logičke jedinice predstavljaju pomoćne modele koji opisuju stvarne elemente koji se pojavljuju u procesoru, te kao takvi služe isključivo tokom razvoja, simulacije i testiranja. Oni se međutim ne sintetiziraju nigdje već samo opisuju gotovi sklop 74xx serije. Također, radi potrebe simulacije bio je potreban model *sRAM-a*. Njega nismo samostalno modelirali već smo koristili gotovi, javno dostupan, VHDL model *sRAM-a*[1].

Cjelokupni sustav je modeliran strukturalno. Definirane su instance pojedinih elemenata koje su zatim pripadnim signalima međusobno povezane u smislenu cjelinu. Na kraju, prilikom izvođenja *testbancha,* jedini su ulazi signal takta i *reset* signal. U fazi simulacije dodan je i jedan izlaz nazvan *err* koji označava da li je procesor dohvatio još nedefiniranu instrukciju. Kako su sve instrukcije na kraju punjene (iskorištene su sve moguće kombinacije bitova operacijskog koda), u sklopovskoj inačici procesora tog signala nema.

Strukturalni VHDL opis koji je u nastavku naveden već je grafički bio ilustriran na slikama 2. i 6. Kod je prilično jednostavan te komentiran pa neće biti dodatno razmatran.

```
entity whole is
    port(
        clk : in std_logic;
        rst : in std_logic;
        err : out std_logic
    );
end whole;

architecture Structural of whole is

    signal dbus : std_logic_vector(7 downto 0); -- data bus
    signal abus : std_logic_vector(15 downto 0); -- address bus

    signal sAA : std_logic_vector(7 downto 0); -- povezivanje A reg - ALU
    signal sAT : std_logic_vector(7 downto 0); -- povezivanje ALU - TMP reg

    -- Control Unit <-> ALU
    signal sCin : std_logic;
    signal sCout : std_logic;
    signal sComp1 : std_logic;
    signal sComp2 : std_logic;

    -- Control Unit -> TMP reg
    signal snET : std_logic; -- OE
```

1 http://tams-www.informatik.uni-hamburg.de/vhdl/models/sram/sram_d.vhd

```
    signal sWT : std_logic; -- write

    -- Control Unit -> A reg
    signal sWA : std_logic; -- write

    -- Control Unit -> IR reg
    signal sWIR : std_logic;
    signal sopcode : std_logic_vector(2 downto 0);

    -- IR reg -> ALU
    signal sfunc : std_logic_vector(3 downto 0);

    -- IR optype -> ALU, CU
    signal sopt : std_logic;

    -- veza izmedju dva 4 bitna alua
    signal scarry : std_logic;

    -- veza prema cpld1

    signal ssr : std_logic_vector(1 downto 0);
    signal ss0 : std_logic;
    signal ss1 : std_logic;
    signal sincPC : std_logic;
    signal srstPC : std_logic;
    signal sre : std_logic;
    signal srs : std_logic;

    -- veza prema mem
    signal snOE : std_logic;
    signal snWE : std_logic;
begin

    IR : entity work.registar port map(
        clk => sWIR,
        noe => '0',
        d => dbus,
        q(3 downto 0) => sfunc,
        q(4) => sopt,
        q(7 downto 5) => sopcode(2 downto 0)
    );

    A : entity work.registar port map(
        clk => sWA,
        noe => '0', -- OE je stalno ukljucen
        d => dbus,
        q => sAA -- A -> ALU(operand1)
    );

    aluLow : entity work.alu181 port map(
        A => sAA(3 downto 0),
        B => dbus(3 downto 0),
        F => sAT(3 downto 0),
        Cn4 => scarry, -- carry output -> signal carry
        M => sopt, -- operation type (logic/arith)
        S => sfunc, -- alu operation
        Cn => sCin, -- carry in iz CU
        AeqB => sComp1 -- comparator output
    );

    aluHigh : entity work.alu181 port map(
        A => sAA(7 downto 4),
        B => dbus(7 downto 4),
        F => sAT(7 downto 4),
        Cn => scarry, -- signal carry -> carry input
        M => sopt, -- operation type (logic/arith)
        S => sfunc, -- alu operation
        Cn4 => sCout, -- carry output u CU
        AeqB => sComp2 -- comparator output
    );
```

```
    T : entity work.registar port map(
        clk => sWT,
        noe => snET,
        d => sAT,
        q => dbus
    );

    -- control unit
    CU : entity work.cpld2 port map(
        clk => clk,
        rst => rst,

        -- upravljacki signali za cpld1
        sr => ssr,
        s0 => ss0,
        s1 => ss1,
        incPC => sincPC,
        rstPC => srstPC,
        re => sre,
        rs => srs,

        -- upravljanje memorijom
        nOE => snOE,
        nWE => snWE,

        WA => sWA, -- upravljanje A registrom
        WIR => sWIR, -- upravljanje IR registrom

        -- upravljanje TMP registrom
        nET => snET,-- enable output
        WT => sWT, -- write

        -- suceljavanje ALU-a
        Cin => sCin, -- carry in
        comp1 => sComp1, -- comparator output (hi and low alu)
        comp2 => sComp2,
        Cout  => sCout, -- carry in

        -- opcode iz IR
        opcode => sopcode,

        opt => sopt,

        err => err
    );

    -- adresni kontroler
    AC : entity work.cpld1 port map(
        clk => clk,
        sr => ssr,
        s0 => ss0,
        s1 => ss1,
        inc => sincPC,
        rst => srstPC,
        re => sre,
        rs => srs,
        o => abus,
        d => dbus
    );

    RAM : entity work.sram port map(
        nCS => '0', -- chip select je stalno ON ovdje
        nOE => snOE,
        nWE => snWE,
        A => abus(14 downto 0),
        D => dbus
    );
end Structural;
```

7. Adresni prostor

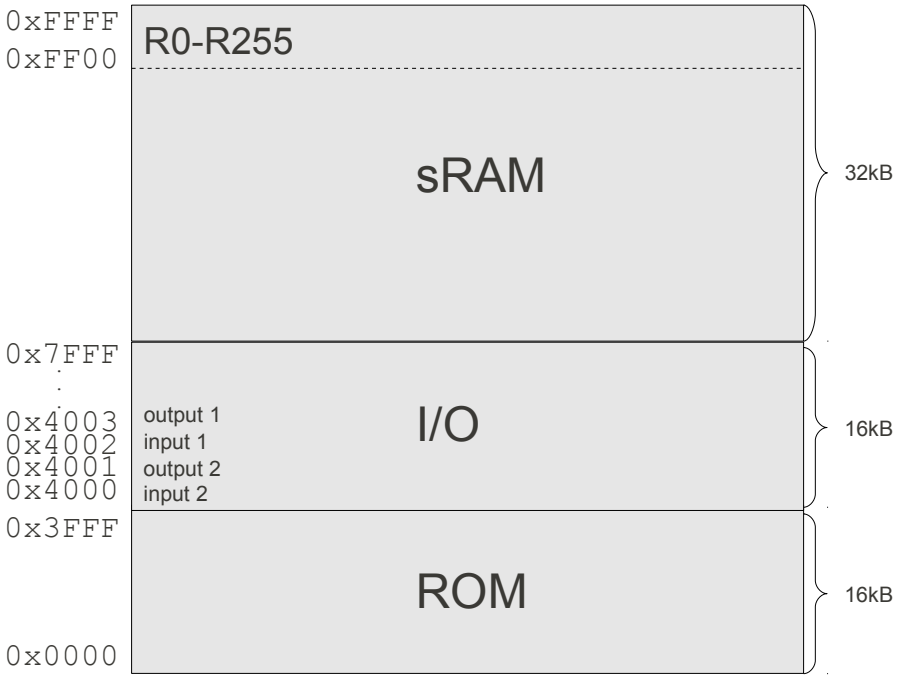
Kako je adresna sabirnica širine 16 bita, procesor može adresirati ukupno 2^{16}, odnosno 65536 memorijskih lokacija (64kB). Svaka je memorijska lokacija naravno veličine 8 bita. U simulaciji nije implementirana podjela adresnog prostora već se sRAM prostire kroz svih 64kB. To je učinjeno iz razloga što je sRAM model takav da se podatci u njemu mogu definirati na bilo kojim lokacijama te se isto tako mogu u realnom vremenu provjeravati. Time je zapravo olakšano testiranje pristupa memoriji i provjera ispravnog rada svih modeliranih komponenti procesora.

U fizičkoj je izradi procesora međutim adresni prostor podijeljen na prostor ROM memorije (u kojoj je trajno spremljen programski kod), prostor za ulazno-izlazno mapiranje (unutar kojeg su trenutno postavljena dva 8-bitna ulaza i dva 8-bitna izlaza) te na prostor RAM memorije. Jasno je da se podatci ne mogu zapisati u ROM (premda procesor neće javiti nikakvu grešku ukoliko se to pokuša učiniti) te da se podatci u RAM memoriji (kao niti u registrima) neće sačuvati prilikom gubitka napajanja. Međutim, vrijedi obratiti pažnju da to nije Harvard arhitektura te da podatci i programski kod nisu razdvojeni. Ovdje se radi o klasičnoj Von Nuemann arhitekturi u kojoj se podatci i programski kod nalaze u istom adresnom prostoru. Stoviše, ideja za buduće proširenje našeg procesorskog sustava je ROM zamijeniti s još jednom sRAM memorijom te dodati sklop koji bi učitavao program sa SD/MMC kartice u memoriju. Iz tog razloga kontrolna jedinica ima mogućnost postaviti adresnu sabirnicu u stanju visoke impedancije (da bi ju oslobodila u trenutku kada bi ju koristio sklop za učitavanje SD/MMC kartice). To međutim nije napravljeno za vrijeme pisanja ovog teksta te za sada neće biti dalje razmatrano.

ROM se prostire kroz 16kB adresnog prostora, te isto toliko zauzima prostor za mapiranje ulazno izlaznih jedinica. Na kraju, RAM (koji se nalazi u jednom sRAM chipu) se prostire kroz 32 kB. Time je raspodjeljen čitav adresni procesor koji se preko 16-bitne sabirnice može adresirati.

Unutar adresnog prostora rezerviranog za mapiranje ulazno-izlaznih uređaja se za sada nalaze dva ulaza i dva izlaza (na adresama 0x4000, 0x4001, 0x4002 i 0x4003). Ostatak tog prostora trenutno nije iskorišten, no proširenja su jednostavna. Na *flat* kabelu se nalaze sabirnica podataka, adresna sabirnica par kontrolnih signala te napajanje. To znači da je krajnje jednostavno napraviti još i druge ulazne ili izlazne uređaje te ih paralelno spojiti uz već postojeću ulazno-izlaznu pločicu.

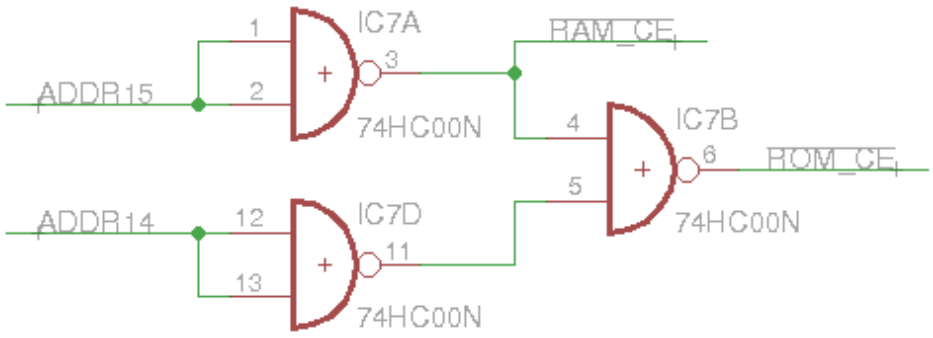
Na slici 17. grafički je prikazan adresni prostor te njegova podjela.



Slika 17: Adresni prostor

Svaki memorijski *chip,* osim do sada opisanih signala, ima i signal $\overline{CE}$ *(Chip Enable)* koji se koristi za odabiranje *chipa* ukoliko ih se u sustavu koristi više. Kako je veličina RAM-a (odnosno sRAM *chipa*) 32 kB, dovoljno je najviši bit adrese dovesti na njegov CE. Međutim, kako *chip* zapravo ima njegov komplement $\overline{CE}$, tada je potrebno staviti inverter (*not gate)* između. Tada će, ukoliko je najviši bit adrese jednak jedinici, $\overline{CE}$ biti jednak nuli (CE = 1) te će se aktivirati sRAM *chip.* Time smo postavili sRAM na najgornjih 32kB memorije.

Nadalje, potrebno je postaviti 16 kB ROM-a na početku adresnog prostora. ROM mora biti aktivan ukoliko su dva najteža bita adrese jednaka nuli (A15 i A14). To postižemo stavljanjem jednog NAND *gatea* nad invertiranim parom najviših bitova. S obzirom da smo A15 već jednom invertirali da bi dobili $\overline{CE}$ signal za sRAM *chip*, to možemo iskoristiti, dok A14 signal trebamo invertirati (negirati).



Slika 18: Adresno dekodiranje za RAM i ROM

Konačni adresni dekoder za RAM i ROM prikazan je na slici 18. Vidimo da se umjesto NOT *gateova* koriste dva NAND *gateova* sa spojenim ulazima. Razlog tomu je što je NAND potreban da bi se generirao signal za ROM, a sam 74HC00N u sebi sadrži četiri NAND *gateova*. Od prije je poznato da se spajanjem ulaza NAND *gatea* može dobiti NOT *gate* odnosno inverter, što je ovdje i učinjeno. U protivnom bi trebalo koristiti dodatni inverter, te bi u ovom *chipu* bio iskorišten samo jedan dio.
Radi potreba mapiranja ulazno-izlaznog modula na postojeći adresni prostor dodan je signal $\overline{MREQ}$ koji je uvijek u logičkoj nuli (odnosno MREQ = 1) kada se pristupa memoriji, bilo za čitanje ili za pisanje. Razlog za takav signal je da se adresa mora prvo stabilizirati, a tek se onda može pristupiti ulazno-izlaznom uređaju na postavljenoj adresi. Na ulazno-izlaznom modulu nalaze se dva ulaza i dva izlaza i to na sljedećim adresama:

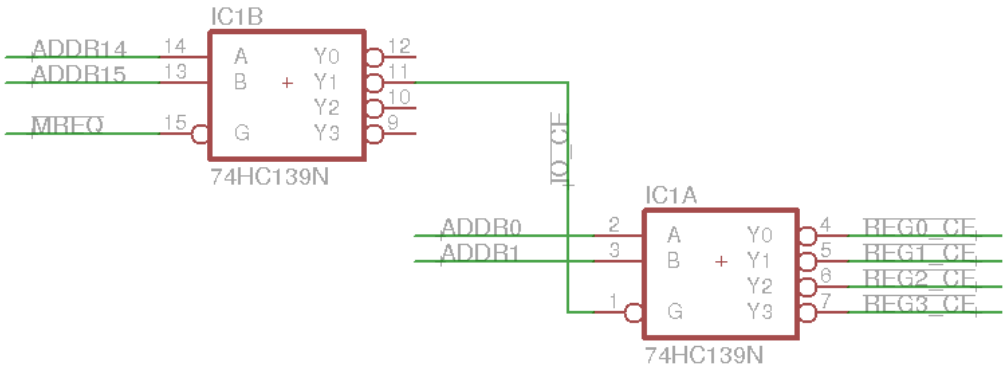
Adresa	Binarni zapis adrese	sučelje
0x4000	01xxxxxxxxxxxx00	input2
0x4001	01xxxxxxxxxxxx01	output2
0x4002	01xxxxxxxxxxxx10	input1
0x4003	01xxxxxxxxxxxx11	Output1

Koristimo dva dekodera (unutar jednog 74HC139N): prvi se aktivira na $\overline{MREQ}$ signal te aktivira sljedećeg ukoliko su dva najviša bita jednaka "01". Drugi dekoder nadalje, ukoliko ga je prvi aktivirao selektira sučelje (jedan od dva ulaznih međuspremnika ili jedan od dva izlaznih registara). Da bi rad dekodera bio jasniji, u nastavku je navedena njihova tablica istine:

A	B	Y0	Y1	Y2	Y3
0	**0**	0	1	1	1
1	**0**	1	0	1	1
0	**1**	1	1	0	1
1	**1**	1	1	1	0

Izlazi dekodera su invertirani, što znači da je aktivni izlaz u logičkoj nuli, dok su svi ostali u logičkoj jedinici. Ukoliko ulaz za uključivanje dekodera (koji je isto u negativnoj logici) G nije postavljen, tada su svi izlazi isključeni (u logičkoj jedinici).

Konačni izgled dekodera za aktiviranje ulaznih i izlaznih sučelja prikazan je na slici 19.



Slika 19: Adresno dekodiranje za IO

Kada su dva najviša bita adresne sabirnice jednaki "01" te kada je postavljen MREQ signal, aktivira se drugi izlaz prvog dekodera. On aktivira drugi dekoder, koji na temelju dva najniža bita adresne sabirnice određuje koji će se izlaz ili ulaz aktivirati. Vrijedi primijetiti da se svi bitovi, osim dva najniža i dva najviša, zanemaruju.

Na izlazima se nalaze registri (koji čuvaju podatak) te zapisuju podatak sa podatkovne sabirnice kada dobiju signal za zapisivanje, dok im je izlaz stalno aktivan. S druge strane, na ulaznim sučeljima se nalaze međuspremnici, koji propuštaju signal sa njihovog ulaza u trenutku kada su uključeni.

8. Izdrada sklopovlja

Do sada je sav rad procesora i periferije objašnjen bez da smo previše zadirali u samu izradu procesora te smo jedino u nekoliko navrata prikazali par shema koje su malo bliže sklopovskoj izradi od ostatka teksta. U ovom poglavlju biti će detaljnije obrađene pojedine pločicama (*PCB)* našeg 8-bitnog CPU sustava.

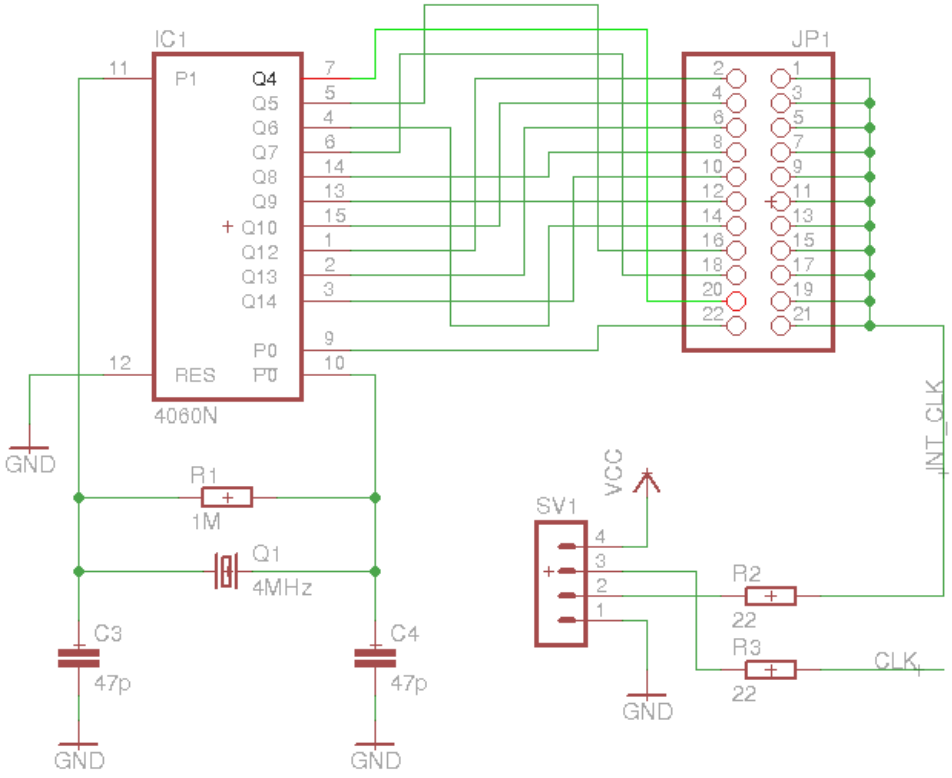
8.1. Memorija, generator takta te napajanje s 5V

Osim CPLD-ova sve se napaja naponom od 5V. Taj napon je osiguran standardnim 7805 regulatorom napona. Na njegovom ulazu, preko konektora za napajanje, dolazi istosmjerni napon od 9-12V (apsolutni maksimum koji može podnijeti je 35V), dok se na njegovom izlazu nalaze stabilnih 5V. Paralelno na njegovim ulazima i izlazima postavljeni su kondenzatori koji poboljšavaju kvalitetu stabiliziranog napona. Potrošnja cjelokupnog procesorskog sustava nije mala (pogotovo zbog velikog broja indikatorskih LED-ica) te se regulator grijao. Iz tog je razloga dodan veći pasivni hladnjak, nakon čega je njegov rad bio zadovoljavajući.

Generator takta je napravljen od jednog 74HC4060N oscilatora - binarnog brojila s oscilatorom spojenog na kvarc od 4MHz. Oscilator u njemu oscilira frekvencijom kvarca (u ovom slučaju je to 4MHz) te se takav izvod dovodi na izlazu *chipa*. Brojilo u nastavku služi dijeljenju osnovne frekvencije i dobivanju nižih frekvencija. Ukupno ima deset izlaza sa različitim "težinama" brojila. Izlaz je moguće birati postavljanjem *jumpera* u željenu poziciju, pa se time mogu dobiti sljedeće frekvencije takta:

jumper	**frekvencija**
11	1 kHz
10	3 kHz
9	7 kHz
8	15 kHz
7	30 kHz
6	63 kHz
5	125 kHz
4	250 kHz
3	500 kHz
2	1 MHz
1	4 MHz

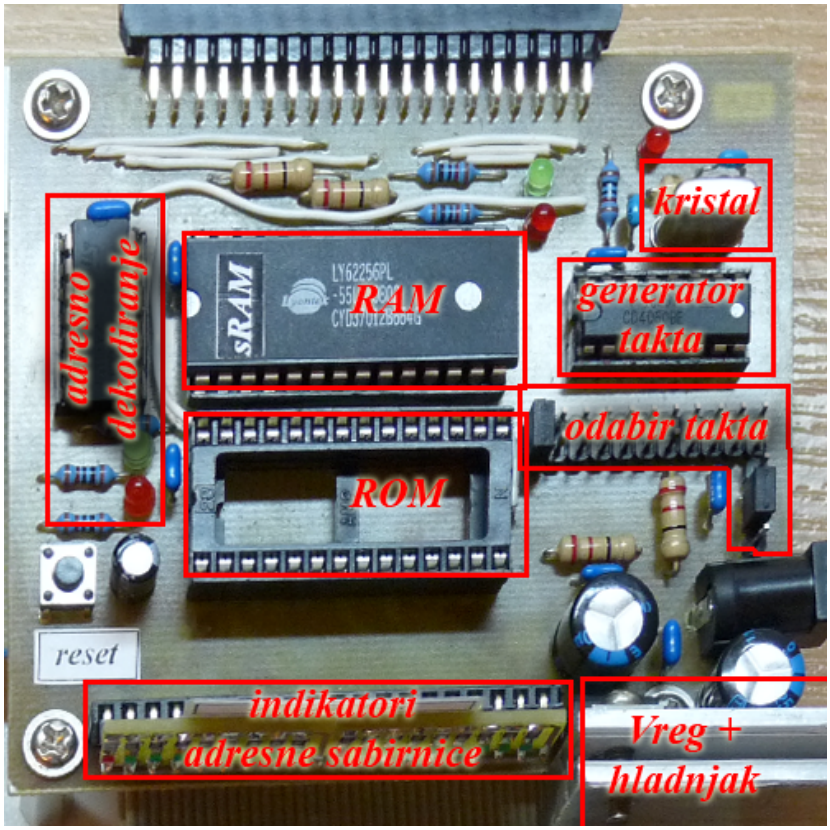
Kako je ideja ovog procesorkog sustava ta da se može izvoditi korak po korak te promatrati njegove operacije, dodan je konektor na kojeg se može spojiti ručno generirani signal takta. Generator ručnog signala takta će biti obrađen kasnije, a sada ćemo samo napomenuti da je moguće odabrati automatski signal takta postavljanjem jumpera na taj konektor ili ručni takt odspajanjem jumpera i spajanjem ručnog generatora.



Slika 20: Generator signala takta

Na kraju, na prvoj se pločici (PCB) osim napajanja i generatora takta nalazi i memorija. Radna je memorija jedan KM62256C *Static RAM* veličine 32kB (odnosno 32K memorijskih lokacija širine 8 bita), dok je ROM zapravo 27C256 preprogramirljivi ROM kojeg se može brisati ultraljubičastom svjetlošću. ROM i RAM su jednako spojeni na adresnu i podatkovnu sabirnicu, a uz njih se nalazi sklop za mapiranje istih na adresnom prostoru procesora koji je već razrađen u prošlom poglavlju, pa se ovdje neće detaljnije spominjati. Nad signalima za čitanje i pisanje (OE, WR) te za odabir *chipa* (*Chip Select ROM-a* ili RAM-a) su postavljene indikatorske LED-ice. Također, na svim bitovima adresne sabirnice je postavljen konektor na kojeg je moguće spojiti indikatorske pločice koje sadrže niz od 8+1 SMD LED-ica.

Zadnja stvar koja se još nalazi na ovoj pločici je mikroprekidač za *reset* tipku, čiji se signal invertira koristeći zadnji slobodan NAND gate (sjetimo se iz prošlog poglavlja da smo ih iskoristili samo 3 od 4 sadržana) te se preko *flat* kabela šalje na pločicu sa CPLD-ovima.



Slika 21: Napajanje, memorija (ROM nije priključen) i generator takta

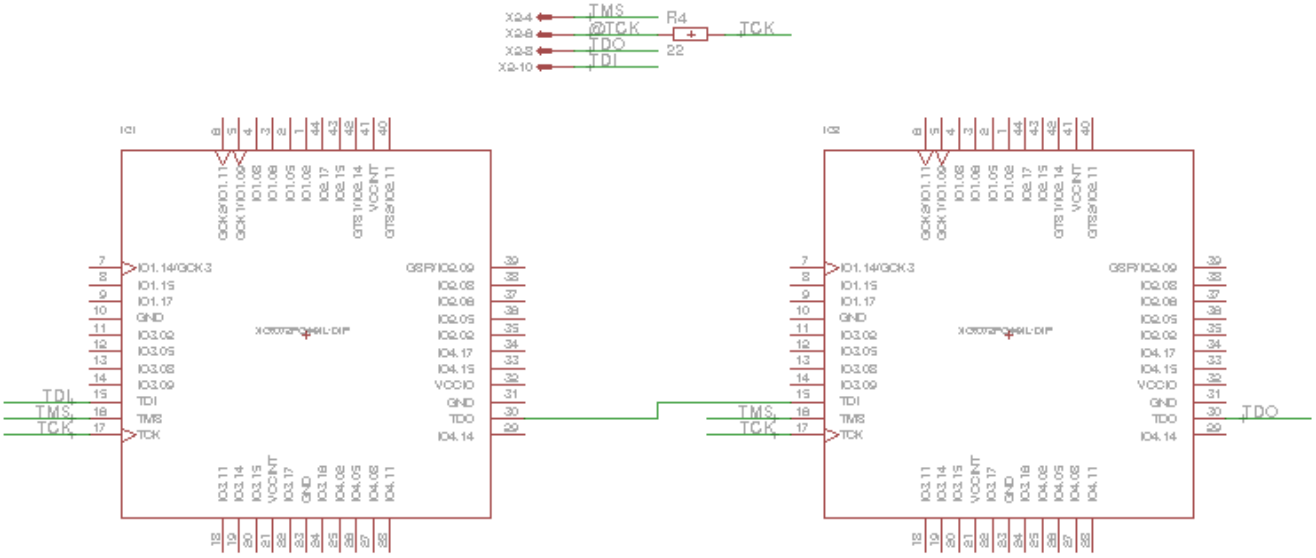
8.2. CPLD-ovi (upravljačka jedinica i upravljanje adresnom sabirnicom)

Na drugoj po redu pločici (PCB), nalaze se dva CPLD-a (prvi u kojemu se nalaze registri PC, PT i OFF upravlja adresnom sabirnicom te drugi koji služi kao upravljačka jedinica). CPLD-ovi rade na napon od 3.3V (ulazi im mogu raditi do maksimalno 5V). Iz tog razloga na toj je pločici bilo potrebno postaviti regulator napajanja koji daje 3.3V te međuspremnici koji služe ispravljanju naponske razine, ne bi li logička jedinica CPLD-a bila preniska za rad ostalih sklopova.

Kao regulator napajanja za 3.3V korišten je LD1117V33C. On pruža maksimalnu struju od 800mA što bi trebalo biti više nego dovoljno za rad CPLD-ova. Naravno, kao i kod svakog drugog regulatora postavljeni su veći elektrolitski kondenzatori te manji keramički kondenzatori od 100nF na njegovim

ulazima i izlazima. Nismo primijetili da se ovaj regulator uopće grije, tako da nije bilo potrebe postaviti hladnjak na njemu.

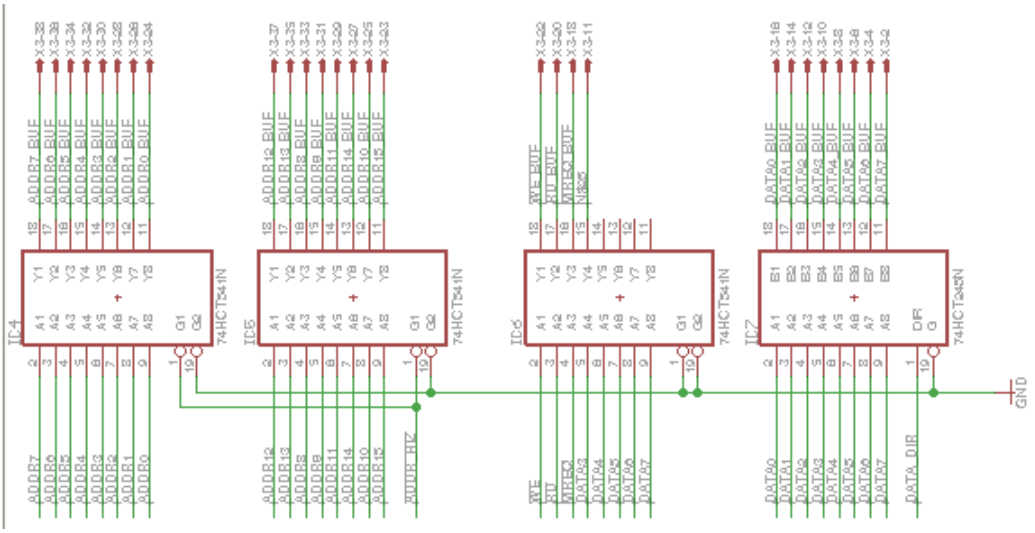
Dva CPLD-a su spojena u tzv. *JTAG Chain.* Time je omogućeno njihovo programiranje bez da ih se mora vaditi iz PLCC *socketa*. Povezivanje u *JTAG Chain* je krajnje jednostavno. Na sve CPLD-ove se donosi TMS i TCK signali, dok se TDI (*input*) i TDO (*output*) spajaju tako da je izlaz prvog CPLD-a ulaz drugom, izlaz drugog ulaz u treći i tako dalje, sve dok se njihovo "nizanje u lanac" ne dovrši povratkom u konektor sa zadnjim signalom TDO. Naravno, potrebno je spojiti i napajanje CPLD-ova (VCC i GND) te VCC donijeti na Vref (*voltage reference*) pin, pošto *Xilinx Platform Cable* to koristi kao referentni napon prilikom komunikacije s CPLD-ovima.



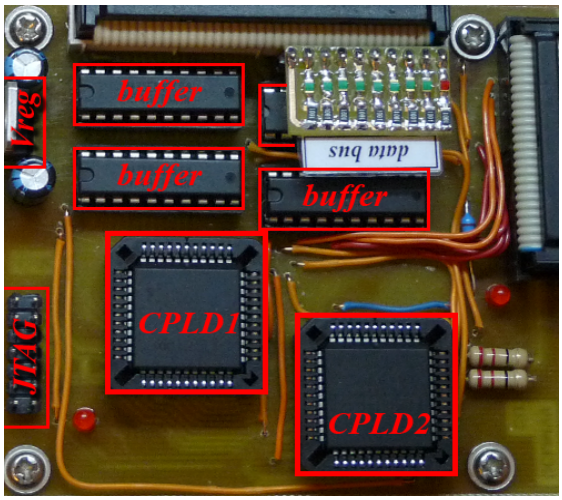
Slika 22: Spajanje CPLD-ova u JTAG Chain (VCC i GND nisu naznačeni)

Već smo spomenuli da CPLD-ovi rade na naponu od 3.3V, a ostatak sklopovlja na 5V te da će iz tog razloga biti potrebno postaviti međuspremnike kojima bi se ispravljale (povećavale) naponske razine. Prvi tip međuspremnika koji je korišten je 74HCT541N. Radi se o osmerostrukom međuspremniku sa ulazom za uključivanje. Takvi su međuspremnici postavljeni na adresnu sabirnicu te na izlaznim upravljačkim signalima upravljačke jedinice. Signal za uključivanje međuspremnika adresne sabirnice doveden je do upravljačke jedinice gdje se naziva *addr_High_Z* te omogućuje postavljanje adresne sabirnice u stanju visoke impedancije. Za sada je taj signal stalno spojen na GND (ulaz za uključivanje međuspremnika je u negativnoj logici), a koristio bi se jedino ako se u budućnosti doda sklop koji bi u memoriji učitavao program sa SD/MMC kartice.

Adresna sabirnica i upravljački signali upravljačke jedinice su jednosmjerni (podatci/signali putuju u jednom smjeru) te je takav i među na njima (propušta i "pojačava" signal u jednom smjeru). Problem nastaje kod podatkovne sabirnice jer je ona dvosmjerna, odnosno podatci mogu ići u oba smjera. Tu je postavljen jedan drugi tip međuspremnika i to 74HCT245N kojemu se može odrediti smjer. Taj signal se onda dovodi do upravljačke jedinice gdje se naziva *data_dir*. Kada podatci idu u smjeru od memorije prema registrima, tada upravljačka jedinica postavlja *data_dir* u logičkoj nuli, čime se odabire taj smjer pojačanja. U slučaju da podatci idu prema memoriji (npr. kod zapisivanja), tada se *data_dir* postavlja u logičku jedinicu te se time odabire smjer pojačanja prema memoriji. Važno je da upravljačka jedinica sinkronizirano postavlja signale uključivanja izlaza pojedinih registara i memorije i *data_dir* signal, kako ne bi došlo do istovremenog postavljanja podatka na sabirnicu od strane više uređaja.



Slika 23: Međuspremnici na adresnoj i podatkovnoj sabirnici te upravljačkim signalima

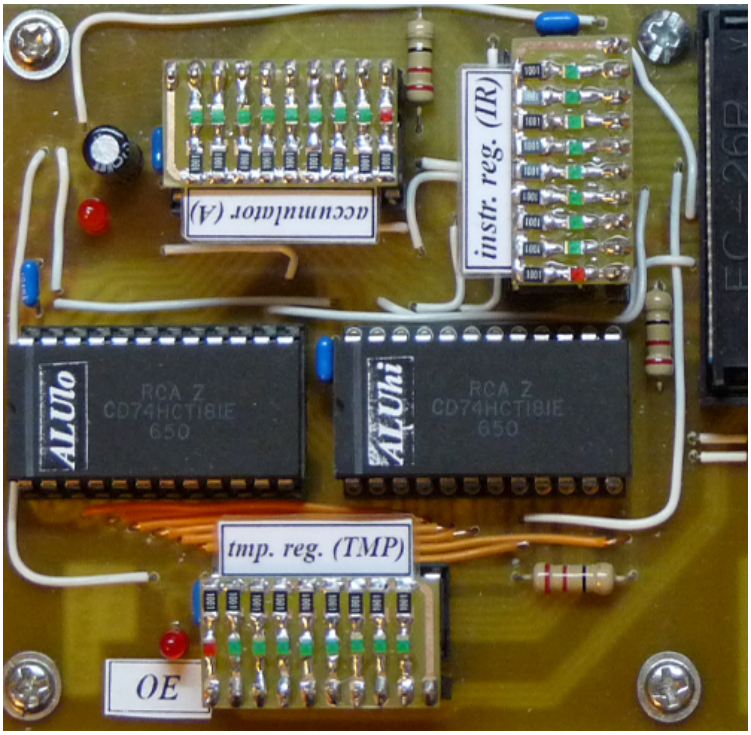


Slika 24: Pločica s CPLD-ovima

8.3. Aritmetičko-logička jedinica i registri

Na trećoj pločici nalaze se registri IR, A i TMP te dvije 4-bitne aritmetičko-logičke (74HC181) jedinice koje zajedno tvore 8-bitnu aritmetičko-logičku jedinicu na način koji je prije pokazan u poglavlju 2. Registri koji se koriste su 74HCT574N te je njihov način rada opisan u poglavlju 3, gdje je napravljen VHDL model tih registara za potrebe simulacije.

Sama pločica (PCB) prima napajanje od 5V sa prve pločice preko druge pločice i *flat* kabela. Nikakvih drugih bitnih elemenata osim ALU jedinica i registara nema, osim naravno konektora koji su paralelno spojeni na svim registrima te omogućavaju postavljanje indikatorskih pločica sa LED-icama iznad svakog registra. Indikatorske pločice se sastoje od 8 zelenih LED-ica koje prikazuju binarni podatak u registru te jedne crvene koja označava *store* signal (prisjetimo se da registri zapisuju podatak na rastući brid *store* signala). Kako su izlazi IR i A registara stalno uključeni, konektori za indikatorske LED-ice su kod njih postavljeni na njihovom izlazu. TMP registar, s druge strane, nema stalno uključen izlaz (jer bi smetao memoriji na podatkovnoj sabirnici) pa je stoga konektor za indikator postavljen na njegovom ulazu. Kako je njegov ulaz zapravo izlaz aritmetičko-logičke jedinice, vrijedi obratiti pažnju da taj set LED-ica ne pokazuje stanje registara cijelo vrijeme, već samo u trenutku njegovog zapisa (kada crvena LED-ica pređe iz stanja logičke nule u logičku jedinicu, odnosno kada se generira rastući brid na *store* signalu).



Slika 25: Aritmetičko-logičke jedinice te registri IR, A i TMP

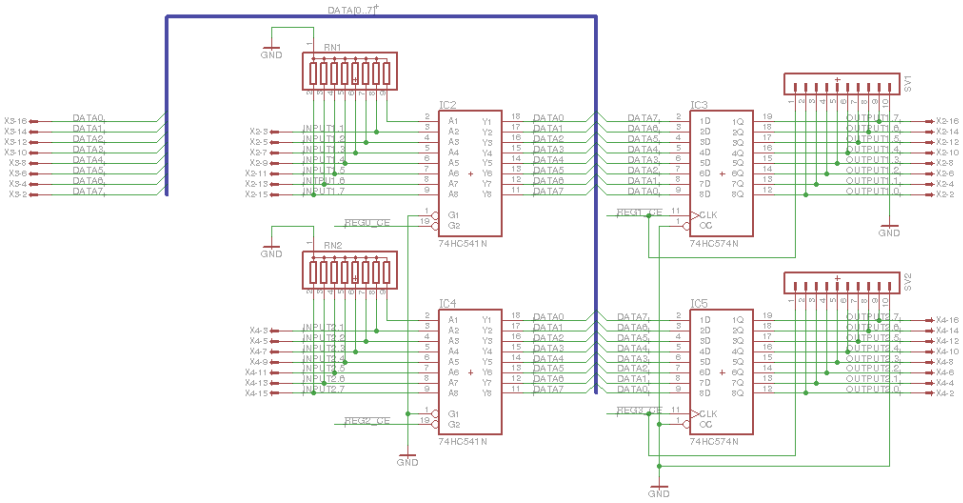
8.4. Ulazno-izlazni modul

Zadnja pločica (PCB) našeg procesorskog sustava predstavlja ulazno-izlazni modul. Taj se modul sastoji od dva ulaza i dva izlaza. U 7. je poglavlju već prikazan adresni dekoder na ulazno-izlaznom modulu. Adresni dekoder generira signale koji uključuju pojedine ulaze ili izlaze.

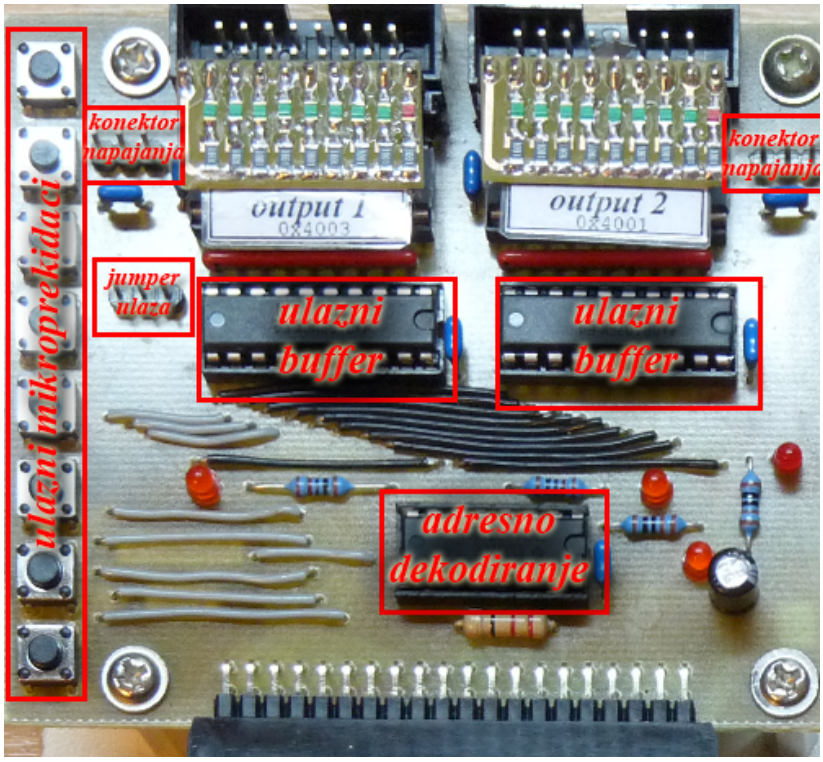
Izlazi su zapravo 8-bitni registri i to 74HCT574N. Kada im adresni dekoder pošalje *store* signal, oni zapisuju podatak s podatkovne sabirnice, a izlaz im je s druge strane stalno uključen. Da njih nema, uređaji spojeni na taj izlaz ne bi imali stabilan podatak dovoljno dugo vremena da ga ispravno učitaju. Na izlaznim registrima postavljeni su konektor za periferiju (na koji se primjerice može spojiti znakovni LCD modul) i konektor za indikatorske pločice s LED-icama koje prikazuju sadržaj na izlazu.

Na ulazima se nalaze 74HC541N međuspremnici. Kada im adresni dekoder pošalje signal na ulazu za aktiviranje, oni propuste podatak na podatkovnu sabirnicu. Razlog tomu je nemogućnost direktnog spajanja na podatkovnu sabirnicu pošto je ona dijeljeni medij. Na ulazima je isto tako spojen konektor koji omogućava spajanje perifernih uređaja, a indikatorskih LED-ica nema, pošto se podatak može pročitati u trenutku kada se on postavlja na podatkovnu sabirnicu. Također, na jednom od ulaza dodano je 8 tipkica (mikroprekidača). Ulazi rade u pozitivnoj logici te svaki od njih sadrži niz *pull-down* otpornika koji ih drži u logičkoj nuli kada nisu spojeni.

Na pločici se nalaze još i LED-ice koje prikazuju trenutke kada se pojedini ulaz ili izlaz aktivira te nekoliko *jumpera* i konektora. Pored svakog od izlaza nalazi se konektor sa napajanjem od 5V na kojeg se može spojiti periferni uređaj (primjerice LCD). Pored mikroprekidača se nadalje nalazi *jumper* kojim se oni mogu uključiti ili isključiti (ukoliko se na ulazu spaja neki drugi periferni uređaj).



Slika 26: Povezivanje ulaznih međupsmrenika te izlaznih registara

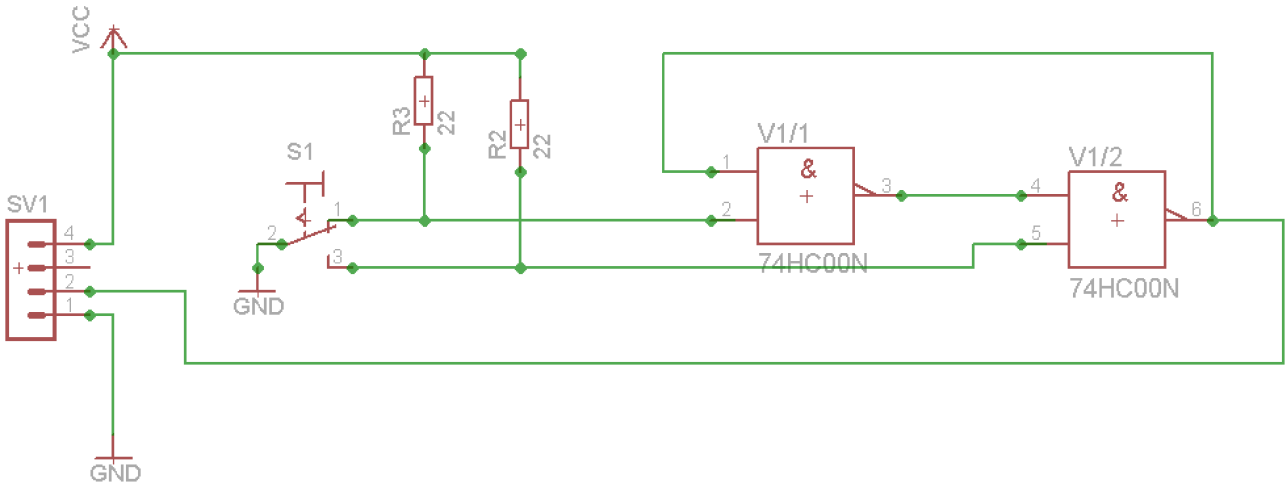


Slika 27: Ulazno-izlazni modul

8.5. Ručni generator takta

Primarna je ideja ovog projekta mogućnost izvođenja instrukcija na procesoru korak po korak. Iz tog je razloga, osim već opisanog generatora takta dodan i ručni generator takta (već smo spomenuli da se na konektor ručnog generatora takta postavljanjem *jumpera* odabire "strojni" generator takta). Problem kod ručnog generatora takta predstavljaju oscilacije koje nastaju prilikom pritiskanja i otpuštanja mikroprekidača. Zbog takvih bi oscilacija zapravo nastalo više impulsa takta nepredvidljive (i mnogo više) frekvencije. Da bi taj problem riješili dodan je SR bistabil. Nakon što je stisnuta tipka *set*, nebitno je da li se ona više puta aktivira (zbog oscilacija smetnji) pošto će i dalje bistabil ostati postavljen. Analogno vrijedi i za *reset* signal na RS bistabilu. Sam bistabil je napravljen od dva NAND *gatea* jednog 74HC00N *chipa.*

Slika 28: Ručni generator takta



Slika 29: Shema ručnog generatora signala takta

9. Primjer programa

U nastavku je opisan primjer programa za naš procesor. Radi se o jednostavnoj petlji koja se vrti od 0x00 do 0x10 te ispisuje trenutnu vrijednost na prvom izlazu (output1). Asemblerski kod bi izgledao ovako:

```
        # postavi 0 u nultu lokaciju najviše stranice (brojač)
        mov a, $0x00
        mov r0, a

        #postavi 0x10 u prvu lokaciju najviše stranice (krajnja vrijednost)
        mov a, $0x10
        mov r1, a

loop:   # postavi adresu ouput1 u PT registar
        mov a, $0x03        # donji dio adrese output1 (0x4003)
        mov ptl, a
        mov a, $0x40        # viši dio adrese output1
        mov pth, a

        # zapiši vrijednost r0 (brojača) na adresi na koju pokazuje PT (output1)
        mov a, r0
        mov (pt), a

        # testiraj da li je r0 = r1
        # koristimo r254 kao privremeni registar da bi na kraju r0 i r1 sačuvali vrijednosti
        mov a, r1
        mov r254, a
        mov a, r0
        setc            # postavi carry (ovo je potrebno, vidi izvadak iz datasheeta ALU-a)
        sub r254, a  #ako su jednaki, tu će se postaviti equal zastavica

        # ukoliko su jednaki, skačemo na labelu kraj
        mov a, $0x2F        #donjih 8 bitova adrese labele kraj
        mov pth, a
        mov a, $0x00        #gornjih 8 bitova labele kraj
        mov pth, a
```

```
        jmpe                    #jump if equal

        # povećavamo r0 za jedan
        mov a, r0
        mov r0, 0x00        # postavljamo r0 na nulu, ovo možemo raditi s ALU-om (vidi tablicu u nastavku)
        setc                # postavi carry (ovo je potrebno, vidi izvadak iz datasheeta ALU-a)
        add   r0, a

        # bezuvjetni skok na labelu loop
        mov a, 0x08
        mov ptl, a
        mov a, 0x00
        mov pth, a
        jmp

kraj:   #beskonacna petlja
        mov a, 0x2F
        mov ptl, a
        mov a, 0x00
        mov pth, a
        jmp             # skoči na labelu kraj.
```

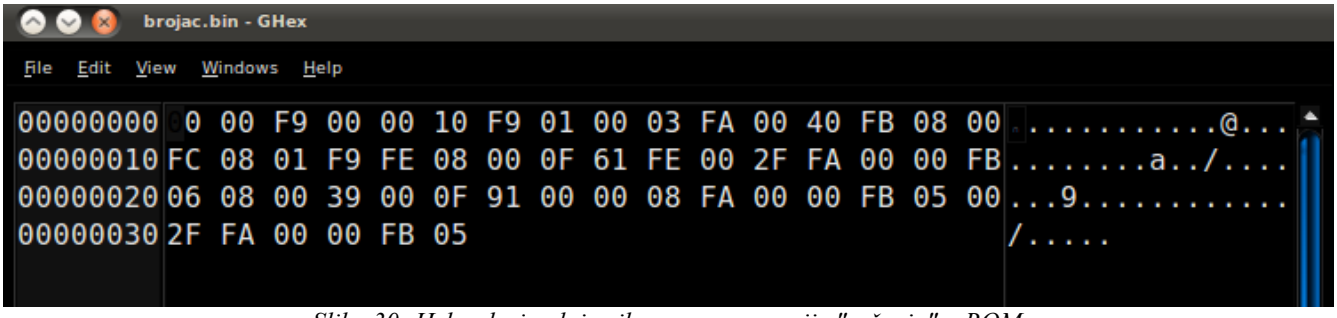
Navedeni programski isječak preveden u operacijske kodove instrukcijskog skupa naveden je u tablici u nastavku:

labela	**adresa**	**operacijski kod**								**ALU operacija**	**opis**	**viši opis**
		ALU oper.				M/O	opcode					
		7	**6**	**5**	**4**	**3**	**2**	**1**	**0**			
	0	0	0	0	0	0	0	0	0	-	A ← const	Postavi 0x00 u akumulator
	1	0	0	0	0	0	0	0	0	-	*vrijednost konstante*	
	2	1	1	1	1	1	0	0	1	A	Rn ← ALU(A, Rn)	Spremi A u R0
	3	0	0	0	0	0	0	0	0	-	*indeks registra*	
	4	0	0	0	0	0	0	0	0	-	A ← const	Postavi 0x10 u akumulator
	5	0	0	0	0	1	0	0	0	-	*vrijednost konstante*	
	6	1	1	1	1	1	0	0	1	A	Rn ← ALU(A, Rn)	Spremi A u

	7	0 0 0 0	0	0 0 1	-	*indeks registra*	R1
loop	8	0 0 0 0	0	0 0 0	-	A ← const	Postavi 0x4003 (adresa od output1) u PT registar
	9	0 0 0 0	0	0 1 1	-	*vrijednost konstante*	
	A	1 1 1 1	1	0 1 0	A	PTLow ← A	
	B	0 0 0 0	0	0 0 0	-	A ← const	
	C	0 1 0 0	0	0 0 0	-	*vrijednost konstante*	
	D	1 1 1 1	1	0 1 1	A	PTHigh ← A	
	E	0 0 0 0	1	0 0 0	-	A ← Rn	Postavi R0 u akumulator
	F	0 0 0 0	0	0 0 0	-	*indeks registra*	
	10	1 1 1 1	1	1 0 0	-	(PT) ← A	Zapiši vrijednost akumulatora na adresi na koju pokazuje PT
	11	0 0 0 0	1	0 0 0	-	A ← Rn	Postavi R1 u akumulator
	12	0 0 0 0	0	0 0 1	-	*indeks registra*	
	13	1 1 1 1	1	0 0 1	A	Rn ← ALU(A, Rn)	Spremi A u R254
	14	1 1 1 1	1	1 1 0	-	*indeks registra*	
	15	0 0 0 0	1	0 0 0	-	A ← Rn	Spremi R0 u A
	16	0 0 0 0	0	0 0 0	-	*indeks registra*	
	17	0 0 0 0	1	1 1 1	-	set carry	postavi carry
	18	0 1 1 0	0	0 0 1	A – Rn - 1	Rn ← ALU(A, Rn)	R254 = A – R254 – 1 (test jesu li A i R254 jednaki)
	19	1 1 1 1	1	1 1 0	-	*indeks registra*	
	1A	0 0 0 0	0	0 0 0	-	A ← const	Adresu labele ***kraj*** (2F) zapiši u PT registar
	1B	0 0 1 0	1	1 1 1	-	*vrijednost konstante*	
	1C	1 1 1 1	1	0 1 0	A	PTLow ← A	
	1D	0 0 0 0	0	0 0 0	-	A ← const	
	1E	0 0 0 0	0	0 0 0	-	*vrijednost konstante*	
	1F	1 1 1 1	1	0 1 1	A	PTHigh ← A	
	20	0 0 0 0	0	1 1 0	-	PT ← (PT) if equal	Skoči @PT ako su isti
	21	0 0 0 0	1	0 0 0	-	A ← Rn	Postavi R0 u akumulator
	22	0 0 0 0	0	0 0 0	-	*indeks registra*	

	23	0 0 1 1	1	0 1 1	logical 0	Rn ← ALU(A, Rn)	R0 = 0x00
	24	0 0 0 0	0	0 0 0	-	*indeks registra*	
	25	0 0 0 0	1	1 1 1	-	set carry	postavi carry
	26	1 0 0 1	0	0 0 1	A + Rn	Rn ← ALU(A, Rn)	R0 = A + R0
	27	0 0 0 0	0	0 0 0	-	*indeks registra*	
	28	0 0 0 0	0	0 0 0	-	A ← const	Adresu labele ***loop*** *(08)* zapiši u PT registar
	29	0 0 0 0	1	0 0 0	-	*vrijednost konstante*	
	2A	1 1 1 1	1	0 1 0	A	TLow ← A	
	2B	0 0 0 0	0	0 0 0	-	A ← const	
	2C	0 0 0 0	0	0 0 0	-	*vrijednost konstante*	
	2D	1 1 1 1	1	0 1 1	A	PTHigh ← A	
	2E	0 0 0 0	0	1 0 1	-	PC ← PT	skoči @PT
kraj	2F	0 0 0 0	0	0 0 0	-	A ← const	Adresu labele **kraj** *(2F)* zapiši u PT registar
	30	0 0 1 0	1	1 1 1	-	*vrijednost konstante*	
	31	1 1 1 1	1	0 1 0	A	TLow ← A	
	32	0 0 0 0	0	0 0 0	-	A ← const	
	33	0 0 0 0	0	0 0 0	-	*vrijednost konstante*	
	34	1 1 1 1	1	0 1 1	A	PTHigh ← A	
	35	0 0 0 0	0	1 0 1	-	PC ← PT	skoči @PT

Heksadecimalni ispis prevedenog programa koji je spreman za zapis u ROM i izvršavanje prikazan je na slici 29.



Slika 30: Heksadecimalni prikaz programa prije "prženja" u ROM

10. Troškovnik

U nastavku je prikazan troškovnik. Cijenu koju on prikazuje je malo veća od cijene izrade samog procesora iz razloga što je dodana određena zalihost komponenata. Tako je primjerice umjesto dva potrebnih CPLD-a kupljeno tri. Zalihosti kod lako nabavljivih komponenata i mehaničkih dijelova nema.

naziv	**kom**	**HRK**
CMOS CPLD, 9572, PLCC44, 3.3V	3	**55**
V REG, LDO, 3.3V, 0.8A,	1	**5**
R SMD 1206 1% 0R 0,25W 33 E 100	25	
SMD 1206 1% 22R 0,25W 11 E 416	20	
SMD 1206 1% 1K 0,25W 11 E 496	80	
R SMD 1206 1% 1R 0,25W	5	
R 0,5 W 1 MR	5	
R 0,5 W 100 KR	10	
R 0,5 W 1 KR	10	
R 0,5 W 22 R	10	
CONN. RIBBON SOCKET FAS 40	4	
CONN. RIBBON SOCKET FAS 14	2	
CONN. RIBBON SOCKET FAS 16	2	
CONN. RIBBON SOCKET FAS 26	2	
CONN.HEADER NISKI FAP 16	2	
CONN.HEADER NISKI FAP 26	2	
ODSTOJNIK M-Ž M3 x 30 mm	16	
STRIP M 2X40 POLNI RAVNI	1	
STRIP M 40 POLNI RAVNI	3	
STRIP Ž 40 POLNI RAVNI	3	
STRIP M 2X40 POLNI KUTNI	3	
LED GREEN SMD 0805 DIFFUSED	60	
LED RED SMD 0805 DIFFUSED	20	
74HCT245 CMOS IC	4	
4060 CMOS IC	2	
74HC00 CMOS IC	2	
74HC541 CMOS IC	2	
74HC574 CMOS IC	2	
COND. MUL 0,1 UF 50V RM2,5	20	
COND. MUL. 47 PF	5	
ELCO 1 MF 100 V AKCIJA!!!!!!!	5	
ELCO 10 MF 100 V	5	

ELCO 100 MF 25 V	4	
ELCO 470 MF 25 V	4	
SOCKET DIL 28 PIN ŠIROKI	2	
SOCKET DIL 24-6 PIN ŠIROKO *!	2	
SOCKET DIL 20 PIN	10	
DC konektor za pločicu	1	
pripadni za kabel	1	
PLOČICA VETR.FOTO 100x160 JEDNOSTRAN	3	**530**
Flat kabeli i jumperi	-	**60**
		650

11. Zaključak

Procesor je na kraju projekta proradio te uspješno izvodi instrukcije. Osim što je sama izrada bila veoma poučna, smatramo da je i samo promatranje stanja u pojedinim fazama njegovog izvršavanja ručnim signalom takta ("*steppanje"*) veoma zanimljivo i poučno i za osobe koje nisu sudjelovale u izradi ovog projekta. Obzirom da je ovo prvi prototip, nije bilo moguće unaprijed predvidjeti određene stvari koje bi možda bilo bolje drugačije napraviti. Tako bi primjerice bilo bolje zamijeniti ulaze na aritmetičko-logičkoj jedinici, čime bi se dobila naredba Rn = Rn – A umjesto sadašnje naredbe Rn = A – Rn. To bi omogućilo puno efikasnije izvođenje raznih petlji. Druga greška u oblikovanju predstavlja instrukcija za učitavanje PC registra u PT registar. Pretpostavimo da želimo implementirati funkciju CALL; ona bi mogla izgledati ovako:

```
mov pt, pc        # spremi sadržaj PC-a u PT

#spremi sadrzaj PT registra
mov r0, ptl
mov r1, pth

#upiši adresu skoka u PC
mov ptl, (adresa_nizih_8_bitova_odredisne_adrese)
mov pth, (adresa_visih_8_bitova_odredisne_adrese)

mov pc, pt  #jump
```

Faza pribavi povećava PC za jedan te će PC pokazivati na sljedeću instrukciju. `mov PT, PC` će također povećati PC za 1 te spremiti ga u PT. Primijetimo da će spremljena adresa pokazivati na instrukciju `mov r0, ptl`, što definitivno nije željena povratna adresa. Srećom je to problem koji se može riješiti prilikom pisanja asemblera (drugačijim funkcijskim pozivima gdje asembler napravi *push* povratne vrijednosti (koja je konstanta) na stog), pa to nije predstavljalo kritični problem.

Loša (i do sada samo djelomično riješena) strana je osjetljivost na smetnje. Inicijalno je bio još osjetljiviji na smetnje te je bilo potrebno postaviti prikladnije kondenzatore na CPLD-ovima, dodati otpornike u sabirnicama (terminiranje) te pojačati GND-ove na pojedinim mjestima.

Drugi problem s kojim smo se susreli je bilo preslušavanje između *reset* signala i podatkovne sabirnice na *flat* kabelu zbog čega bi se pojavio kratki *reset* impuls kada bi na podatkovnoj sabirnici bile sve

jedinice (0xFF). Taj je problem riješen ispravnim terminiranjem linije (dodavanjem otpornika) te dodavanjem kondenzatora na *reset* signal.

Zaključno možemo reći da je projekt bio jako zanimljiv i poučan te da nam je na žalost više nedostajalo znanje iz elektronike (uvidjeli smo pojave kojih nema prilikom jednostavne VHDL simulacije) nego iz arhitekture računala prilikom izrade ovog procesora i njegove periferije. Na kraju bismo se željeli zahvaliti kolegi Davoru Cihlaru na brojnim savjetima (pogotovo pri otklanjanju smetnji i preslušavanja podatkovne sabirnice) bez kojih procesor vjerojatno ne bi zaživio izvan VHDL simulatora.

12. Literatura

1. Xilinx, *Bulletproof CPLD Design Practices,* 2005.
2. Xilinx, *Planning for High Speed XC9500XL Designs* , 1998.
3. B. Segee, J. Field, *Microprogramming and Computer Architecture*, 1991.