Proiectarea unei unități centrale

Proiect Structura sistemelor de calcul

Student: Teocan Patricia Claudia

Grupa: 30234

Cuprins

1. Introducere

1.1 Context

Principalul scop al proiectului este implementarea și funcționarea ale unui circuit cu 20 instrucțiuni ce respectă standardul IEEE configurate pe 16 biți. Principalele funcționalități ale circuitului sunt efectuarea operațiilor de tip aritmetic, logic, de transfer, de salturi condiționate sau necondiționate, rotiri. Se pot realiza operații de adunare și scădere, de tip shift și mai multe tipuri de rotiri la baza biților, la stânga sau la dreapta, aritmetice sau logice, salturi la alte instrucțiuni sau operațiile de tip AND, OR, XOR, NOT, de comparație, întregul cod fiind scris in limbajul de descriere hardware VHDL si utilizându-se IDE-ul VIVADO si plăcuța BASYS 3.

Scopurile acestui dispozitiv sunt variate. Principalul mod în care acesta poate fi folosit este în varianta lui inițiala, drept circuit propriu-zis pentru a efectua operațiile pe care acesta le are implementate: operații pe 16 biți ce respecta formatul IEEE. O altă opțiune ar fi utilizarea drept aparat periferic și utilizarea funcționalităților acestuia de către al dispozitiv. În plus, mai există varianta de integrare a acestuia într-un alt circuit si folosirea instrucțiunilor in cadrul lui, mapându-se foarte ușor în stil structural, utilizând codul VHDL.

1.2 Specificații

Pentru început, menționez modul de implementare și utilizare al circuitului. Întregul cod este scris în limbajul bine cunoscut de descriere hardware, și anume VHDL în care am realizat multiple proiecte în anii trecuți de facultate la alte materii de tip hardware, precum Proiectarea Sistemelor Numerice, Arhitectura Calculatoarelor. Instrucțiunile descrise respectă standardul IEEE. Implementarea si simularea circuitului este realizată cu ajutorul IDE-ului de la XILINX, VIVADO. Pentru utilizarea dispozitivului și demonstrarea funcționalităților acestuia se folosește plăcuța din gama XILINX, BASYS 3. Codul VHDL se încărcă pe aceasta și circuitul poate fi utilizat folosind butoanele, switch-urile, afișoarele cu 7 segmente.

Pentru ca dispozitivul să fie mai ușor de folosit de către utilizatori, atât datele de intrare, introduse de acesta, cât și datele de ieșire sunt într-un limbaj înțeles de către om, întrucât se folosește sistemul hexa, cu cel binar cu care lucrează circuitul. La nivelul executării operațiilor, datele sunt interpretate în sistem binar, cum este specific tuturor circuitelor hardware. Astfel, este nevoie de 2 componente de conversie a datelor: unul pentru input din hexa în binar și unul pentru output din binar în hexa. În plus, utilizatorul are la dispoziție o funcție de tip RESTART ce poate fi folosită în orice moment al utilizării dispozitivului în caz că acesta ar fi ales o operație nedorita sau a introdus greșit unul dintre numere. Odată cu apelarea acestei funcționalități se vor șterge toate datele introduse sau selectate de utilizator în cadrul ultimei utilizări a circuitului. Numere introduse de acesta vor fi șterse complet din memoria de care se folosește dispozitivul,

dar la fel si operațiile selectate pentru returnarea unui rezultat dorit. Ceea ce se întâmplă la nivel intern, este revenirea la o stare inițială setata de programator în implementarea device-ului. După folosirea acestei funcții, circuitul poate fi folosit în mod obișnuit.

Vorbind despre nivelul intern al circuitului, modul în care acesta este implementat este cel structural. Se folosesc bistabile și porți logice de diferite tipuri care vor fi mapate. Modul de lucru este cel de împachetare, întrucât se începe de la cele mai mici circuite care sunt utilizate drept componente propriu-zise. În cadrul implementării se folosesc 9 registre interne, dintre care unul are rolul de acumulator, iar celelalte 8 sunt registre generale. În plus, tipul de adresare este cea indirectă și principiul care sta la baza acesteia este cel de stivă care funcționează în mod LIFO. Mai exact, ultimul element introdus este primul extras din structura de date.

1.3 Obiective

Obiectivul acestui proiect este implementarea, simularea și asigurarea funcționalității corecte unui circuit ce are implementate 20 de instrucțiuni cu numere întregi și în complement față de 2 pe 16 biți. Pe lângă buna funcționare a dispozitivului și corectitudinea rezultatelor returnate de circuit, este important ca utilizatorul să îl poată folosi ușor iar rezultatul să fie înțeles de către acesta.

Modul în care utilizatorul folosește circuitul este cu ajutorul plăcuței BASYS 3. Rezultatul este vizibil pe afișoarele cu 7 segmente. Valorile suportate de dispozitiv sunt limitate de cei 16 biți. La nivel hexa, se pot folosi numere cu în complement față de 2 cu valori cuprinse în intervalul de la $-32{,}768$ și până la $32{,}767$.

2. Studiu bibliografic

În cadrul acestui proiect, numerele sunt stocate pe 16 biți astfel: primul bit este rezervat pentru stocarea semnului numărului (0 pentru numerele pozitive, respectiv 1 pentru cele negative) iar restul biților reprezintă valoarea în modul. Pentru ca valorile să poate fi interpretate corect de către VHDL, datele de intrare vor fi convertite în complement față de 2. Așadar, valorile respectă standardul IEEE al limbajului de descriere hardware VHDL.

semn	valoare
1 bit	15 biți

Exemple

zecimal	26,013			
hexa	6	5	9	D

binar	0	1	1	0	0	1	0	1	1	0	0	1	1	1	0	1

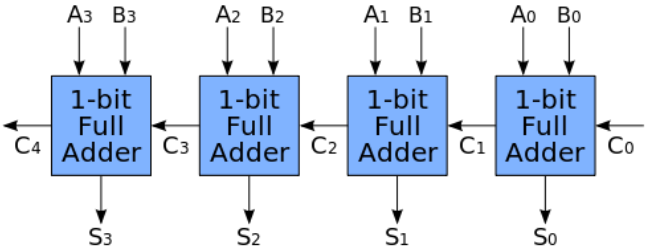
- Operații aritmetice -3
  - De adunare și scădere pe numere de 16 biți, cu semn implementate prin cascadare de sumatoare complete (2)
  - De shiftare aritmetică la dreapta în care pe poziția rămasă liberă se pune 1 în cazul în care numărul a fost negativ și 0 în caz contrar (1)
- Operații logice -11
  - De shiftare logică stânga și la dreapta în care pe poziția rămasă liberă se pune 0 (2)
  - De rotire logică la stânga și la dreapta în care se shiftează toți biții iar pe poziția rămasă liberă se pune valoarea bitului care a fost eliminat (2)
  - Clasice pe biți (AND, OR, XOR, NOT, NAND, NOR, NXOR) (7)
- Salturi necondiționate
- Salturi condiționate
- Operații de comparare
  - Realizate cu ajutorul operațiilor de adunare și scădere
- Operații de testare
  - Test dacă un număr este pozitiv sau negativ verificând primul bit

Implementarea operațiilor menționate mai sus se realizează cu bistabile și porți logice. Se începe de la cea mică componentă, crescând treptat complexitatea componentelor. În componența fiecărei porți se vor regăsi componente mai micuțe implementate anterior, respectând acest principiu. Pentru realizarea fiecărei componente, codul va fi descris în mod structural fără a avea descrieri comportamentale. La fiecare pas se vor mapa componente anterioare.

3. Design

- Operații aritmetice
  - Adunare

    Pentru realizarea adunării a două numere pe 16 biți în complement față de 2, sunt cascadate 4 sumatoare complete pe 4 biți ce sunt implementate la rândul lor prin cascadarea a 4 sumatoare complete pe 1 bit. Transportul de intrare al sumatorului pe 16 biți este deja setat ca mereu să fie valoarea ‘0’. Astfel, utilizatorul nu trebuie să trimită niciun fel de semnal pentru această intrare pentru a ne asigura că nu se strecoară greșeli în procesul de efectuare al acestei operații.



Sumator pe 4 biți prin cascadarea
a 4 sumatoare pe 1 bit

- Scădere
  Modul de implementare folosește circuitul de adunare în întregime. Singura diferență dintre adunare și scădere este reprezentată de valorile de intrare ale sumatorului. Circuitul de intrare are integrată și o componentă care înmulțește cel de-al doilea operand cu -1 pentru putea fi realizată adunarea cu opusul. Astfel este realizată operația de scădere a 2 numere.

- Shiftare aritmetică la dreapta
  Este realizată cu ajutorul unui registru pe 16 biți, aceștia fiind deplasați cu atâtea poziții câte au fost precizate cu ajutorului celui de-al doilea termen la dreapta, iar pentru determinarea valorii care va veni pe primii biți, se folosește un multiplexor pe 1 bit a cărui selecție va intra vechiul prim bit al numărului.

- Operații logice
  - Shiftare logică la stânga și dreapta
    Sunt realizate cu ajutorul unui registru pe 16 biți, aceștia fiind deplasați cu atâtea poziții câte au fost precizate cu ajutorului celui de-al doilea termen la stânga sau la dreapta, iar pe biții rămași liberi se va pune valoarea ‘0’.

  - Rotații logice la stânga și la dreapta
    Sunt realizate cu ajutorul unui registru pe 16 biți, aceștia fiind rotiți de atâtea ori câte au fost precizate cu ajutorului celui de-al doilea termen la stânga sau la dreapta.

  - Operații logice clasice (AND, OR, XOR, NOT, NAND, NOR, NXOR)
    Sunt implementate cu ajutorul porților logice clasice specifice. Valorile de intrare și de ieșire sunt pe 16 biți.

- Salt necondiționat
  Saltul necondiționat se execută cu ajutorul unui registru în care se stochează adresă dorită.

- Operații de testare
  - De egalitate
    Cu ajutorului circuitului de scădere si celui care verifică dacă un număr este egal cu zero, utilizatorul va putea primi o confirmare sau o infirmare pentru egalitatea celor 2 numere introduse.

  - Mai mic
    Cu ajutorului circuitului de scădere si celui care precizează dacă un număr este pozitiv sau nu, utilizatorul va putea primi o confirmare sau o infirmare pentru verificarea daca primul număr introdus este mai mic decât cel de-al doilea.

  - Mai mare
    Cu ajutorului circuitului de scădere si celui care precizează dacă un număr este pozitiv sau nu, utilizatorul va putea primi o confirmare sau o infirmare pentru verificarea daca primul număr introdus este mai mare decât cel de-al doilea.

- Salt condiționat
  Saltul implementat este BLE ce compară două numere, iar în cazul în care primul este mai mic decât al doilea, se executa saltul modificându-se contorul programului.

- Test număr pozitiv sau negativ
  În cazul acestei operații, doar primul număr introdus de utilizator va fi luat în considerare. Folosind un multiplexor ce va avea pe selecție primul bit al numărului, se va afla dacă numărul este pozitiv sau negativ.

4. Implementare

a. Instruction fetch

Instruction fetch este prima componenta a mips-ului. Aici, în memoria de tip ROM denumită specific ROM sunt stocate instrucțiunile care se vor executa folosind plăcuța. În plus, tot în cadrul acestei părți se incrementează și counter-ul “pc_counter” la fiecare semnal de tact. Un alt rol este de a implementa mux-urile pentru instrucțiunile de tip BRANCH și JUMP pe baza semnalelor de tip selecție, și anume PCsrc și Jump.

```
architecture Behavioral of instructionFetch is
    signal pc_counter, NextAddress, PCAux, PCinc,  AuxSgn: STD_LOGIC_VECTOR(15 downto 0) := x"0000";
    type MEM is array(0 to 255) of STD_LOGIC_VECTOR(15 downto 0);
    signal ROM: MEM := (

        B"000_001_010_011_0_000", -- add
        B"000_001_010_011_0_001", -- sub
        B"000_001_010_011_0_010", -- arithm shift

        B"000_001_010_011_0_011", -- logical left shift
        B"000_001_010_011_0_100", -- logical right shift
        B"000_001_010_011_0_101", -- logical left rotate
        B"000_001_010_011_0_110", --  logical right rotate

        B"000_001_010_011_0_111", --  and
        B"001_001_010_011_0_000", -- nand
        B"001_001_010_011_0_001", -- or
        B"001_001_010_011_0_010", -- nor
        B"001_001_010_011_0_011", -- xor
        B"001_001_010_011_0_100", -- nxor
        B"001_001_010_011_0_101", -- not

        B"011_100_010_010_0_000", -- ble
        B"000_000_000_000_0_000", -- noop
        B"000_000_000_000_0_000", -- noop
        B"000_000_000_000_0_000", -- noop
        B"001_001_010_011_0_110", -- equality test fail
        B"001_100_101_011_0_110", -- equality test pass
        B"001_010_001_011_0_111", -- greater test fail
        B"001_001_010_011_0_111", -- greater test pass
        B"010_001_010_011_0_000", -- lt test fail
        B"010_010_001_011_0_000", -- lt test pass
        B"010_001_010_011_0_001", -- positivity test    0 poz, 1 neg
```

```
begin
    -- PROGRAM COUNTER
    process(clk)
    begin
      if reset = '1' then
        pc_counter <= x"0000";
      end if;
        if rising_edge(clk) then
            if enable = '1' then
               pc_counter <= NextAddress;
             end if ;
        end if;
    end process;
    Instruction <= ROM(conv_integer(pc_counter));
    PCAux <= pc_counter + 1;
    PCinc <= PCAux;
    -- mux branch
    process(PCsrc, PCAux, BranchAddress)
    begin
        if PCsrc = '0' then
            AuxSgn <= PCAux;
        else
            AuxSgn <= BranchAddress;
        end if;
    end process;
    -- mux jump
        process(Jump, AuxSgn, JumpAddress)
        begin
            if jump = '0' then
                NextAddress <= AuxSgn;
            else
                NextAddress <= JumpAddress;
            end if;
        end process;
    PC <= pc_counter;
end Behavioral;
```

b. Instruction decode

În aceasta componentă sunt stocate registrele care păstrează la fiecare pas valorile asupra cărora se vor efectua operațiile implementate. Avem 8 registre generale care sunt inițializate cu valori aleatorii pentru a putea detecta greșelile de implementare care s-ar putea strecura și unul acumulator în care se va stoca rezultatul operației.

```
architecture Behavioral of instructionDecode is
signal input: STD_LOGIC_VECTOR(2 downto 0);

type IDtype is array(0 to 7) of STD_LOGIC_VECTOR(15 downto 0);
    signal ID: IDtype := (
         x"0001",   -- 00
         x"0050",   -- 01
         X"0029",   -- 10
         X"0002",   -- 11
         X"0002",
       others => x"1234");

signal acumulator: STD_LOGIC_VECTOR(15 downto 0);
begin

    process(clk, enable, regWrite)
    begin
       if rising_edge(clk) then
           if  regWrite = '1' and enable = '1' then
               ID(conv_integer(instruction(6 downto 4))) <= wd;
           end if;
       end if;
  end process;
  acumulator <= wd;

  func <= instruction(2 downto 0);

end Behavioral;
```

c. Instruction execute

Rolul acestei componente, după cum îi spune și numele, este de a executa operațiile implementate. Sunt mapate toate instrucțiunile implementate pentru a putea fi folosite în mod structural în cadrul proiectului: cele aritmetice, logice și, respectiv, de testare. După cum se vede și în poză, sunt doar 18 operații mapate, celelalte două sunt operații BRANCH și JUMP în cazul cărora nu trebuie implementate suplimentar, întrucât au fost deja codificate in componenta Instruction decode, sub formă de multiplexoare.

```
begin
      adder: fullAdder_16bit port map('0', RD1, RD2, ADDres, coutADD); -- 1
      subtract: subtract_16bit port map( '0', RD1, RD2, SUBres, coutSUB);-- 2
      rightArithmShift_pm: rightArithmShift_16biti port map(RD1, rightArithShift);--3

      -- logice
      leftLogicalShift_pm: leftLogicalShift port map(RD1, leftLogShift);  -- 4
      rightLogicalShift_pm: rightLogicalShift port map(RD1, rightLogShift);  -- 5
      leftLogicalRot_pm: leftLogicalRot port map (RD1, leftLogRot);  --6
      rightLogicalRot_pm: rightLogicalRot port map (RD1, rightLogRot);  --7

      ORgate_16bit_pm: ORgate_16bit port map(RD1, RD2, ORres);   -- 8
      NORgate_16bit_pm: NORgate_16bit port map(RD1, RD2, NORres);  --9
      ANDgate_16bit_pm: ANDgate_16bit port map(RD1, RD2, ANDres);  -- 10
      NANDgate_16bit_pm: NANDgate_16bit port map(RD1, RD2, NANDres);    -- 11
      XORgate_16bit_pm: XORgate_16bit port map(RD1, RD2, XORres);   -- 12
      NXORgate_16bit_pm: NXORgate_16bit port map(RD1, RD2, NXORres);    -- 13
      NOTgate_16bit_pm: NOTgate_16biti port map(RD1, NOTres);   -- 14

      -- testare
      pozORneg_pm: pozORneg port map(RD1, pozORnegres); -- 15

      TESTeg_pm: TESTeg port map(RD1, RD2, TESTegRes);   -- 16
      TESTgt_pm: TESTgt port map(RD1, RD2, TESTgtRes);   -- 17
      TESTlt_pm: TESTlt port map(RD1, RD2, TESTltRes);   -- 18
```

```
process(ALUCTRL, RD1, ALUinput2, func)
begin
   if ALUCTRL = "000" THEN   -- sub
       case func is
           when "000" => ALUout <= ADDres;
           when "001" => ALUout <= SUBres;
           when "010" => ALUout <= rightArithShift;
           when "011" => ALUout <= leftLogShift;
           when "100" => ALUout <= rightLogShift;
           when "101" => ALUout <= leftLogRot;
           when "110" => ALUout <= rightLogRot;
           when "111" => ALUout <= ANDres;
       end case;
   end if;
   if ALUCTRL = "001" then
       case func is
               when "000" => ALUout <= NANDres;
               when "001" => ALUout <= ORres;
               when "010" => ALUout <= NORres;
               when "011" => ALUout <= XORres;
               when "100" => ALUout <= NXORres;
               when "101" => ALUout <= NOTres;
               when "110" => ALUout <= "000000000000000" & TESTegRes;
               when "111" => ALUout <= "000000000000000" & TESTgtRes;
           end case;
   end if;
   if ALUCTRL = "010" then
       case func is
               when "000" => ALUout <= "000000000000000" & TESTltRes;
               when "001" => ALUout <= "000000000000000" & pozORnegres;
               when others => ALUout <= "XXXXXXXXXXXXXXXX" ;
           end case;
   end if;
```

```
if ALUCTRL = "011" then
    case func is
            when "000" => ALUout <= "000000000000000" & TESTltRes; -- ble
            when others => ALUout <= "XXXXXXXXXXXXXXXX" ;
        end case;
end if;

BranchAddress <= PC + 1 + RD1;
if ALUCTRL = "011" and func = "000" and TESTltRes = '1' then
    zero <= '1';
else
    zero <= '0';
end if;
ALURes <= ALUOut;

d process;
```

Selecția operației care se execută se face în funcție de semnalele ALUCTRL si FUNC care sunt trimise din componenta de control Main Unit. În plus, tot aici se setează și bitul Zero din cadrul operației de tip BRANCH. Acesta se activează dacă este îndeplinită condiția impusă și daca semnalul ALUCTRL ilustrează faptul că operația curentă este de acest tip.

d. Memory

La fel ca celelalte componente, și aceasta are un nume sugestiv. Aici este implementată memoria RAM denumita specific, RAM, care stochează toate numerele pentru operații. Astfel, tipul de adresare în cadrul acestui proiect este indirectă. Pe ieșirile acestei componente sunt informații din memoria RAM, de la adresele cerute de instrucțiunea curentă.

```
architecture Behavioral of memory is

    type MEM is array(0 to 15) of std_logic_vector(15 downto 0);
    signal RAM: MEM := (
           x"0000",   -- 000
           x"0045",   -- 001
           X"0033",   -- 010
           X"0000",   -- 011
           X"0003",   -- 100
           X"0003",   -- 101
            OTHERS => X"0000");

begin
    -- write first
    process(clk, enable, MemWrite)
     begin
          MemData <= RAM(conv_integer(ALUResIn));
          MemDataRD1 <= RAM(conv_integer(rd1_code));
          MemDataRD2 <= RAM(conv_integer(rd2_code));

    end process;
   AluResOut <= AluResIn;
end Behavioral;
```

e. Test env

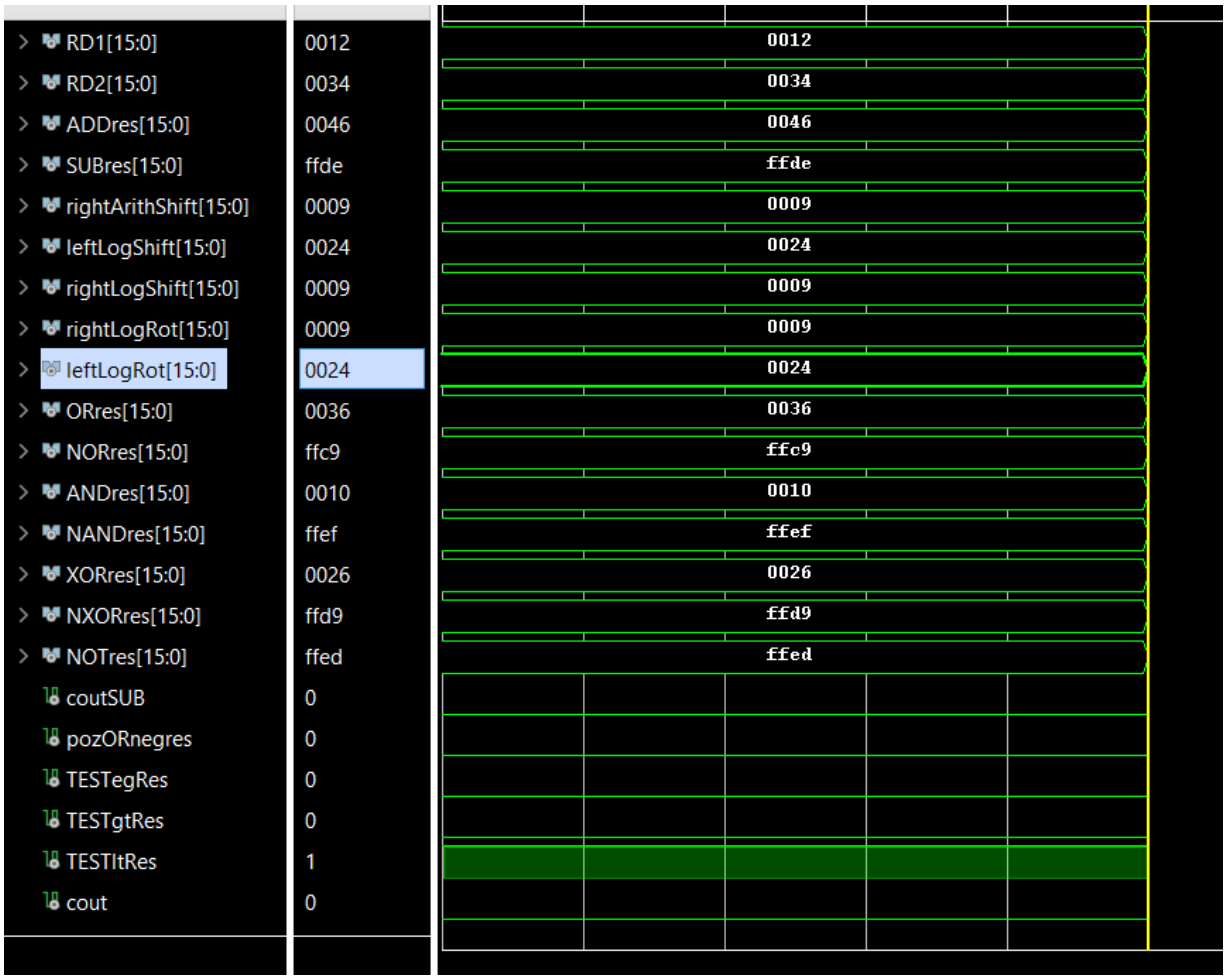
```
MPG1portmap: MPG port map(btn(0), clk, enable);
MPG2portmap: MPG port map(btn(1), clk, resetEnable);

IFportmap: instructionFetch port map(JumpAddress, BranchAddress, Jump, PCSrc, clk, resetEnable, enable, PCcounter, instruction);
IDportmap: instructionDecode port map(clk, enable, instruction(12 downto 0), WriteData, RegWrite, func);
MainUnitportmap: MainUnit port map(Instruction(15 downto 13), RegDst, RegOp, ALUSrc, Branch, Jump, ALUOp, MemWrite, MemToReg, RegWrite);
EXportmap: instructionExecute port map(PCcounter, MemDataRD1, MemDataRD2, ALUOp, func, BranchAddress, ALURes, Zero);
MEMportmap: memory port map(Instruction(12 downto 10), Instruction(9 downto 7), MemWrite, ALURes, RD2, MemData, MemDataRD1, MemDataRD2, ALUResOut, enable, clk);
```

Această componentă reprezintă partea principală a proiectului mips. Aici sunt apelate toate componentele implementate și descrise mai sus. În plus, tot aici se setează si adresa pentru instrucțiunea de tip JUMP și semnalul de selecție pentru operațiile de tip Branch. Pentru a putea fi folosită plăcuța BASYS 3 cu ușurință, a fost implementat un debouncer pentru butoane și afișarea pe afișoarele cu 7 segmente integrate.

```
process(MemToReg, ALURes, MemData, PCcounter)
  begin
      if MemToReg = '0' then
          WriteData <= ALURes;
      else
          WriteData <= MemData;
      end if;
  end process;

-- BRANCH
  PCSrc <= Branch and zero;

-- JUMP
  JumpAddress <= PCcounter(15 downto 13) & Instruction(12 downto 0);

 -- ssd
 with sw(15 downto 13) select
     digits  <= MemDataRD1 when "100",
     MemDataRD2 when "110",
     ALURes when "111",
     WriteData when "000",
     (others => 'X') when others;

SSDportmap: SSD port map(digits, clk, cat, an);
MPG1portmap: MPG port map(btn(0), clk, enable);
MPG2portmap: MPG port map(btn(1), clk, resetEnable);
```

5. Testare și implementare

Următoarele exemple dovedesc funcționalitatea corecta ale operațiilor implementate. RD1 și RD2 reprezintă termenii asupra cărora se aplică operațiile. În cazul operațiilor care au o singură valoare de intrare, se va alege RD1.



6. Concluzii

Rolul acestui proiect a fost proiectarea unei unități centrale cu aproximativ 20 de instrucțiuni(aritmetice, logice, de transfer si de salt). În implementare au fost necesare 9 registre, 1 de acumulator și 8 registre generale. Tipul de adresare a trebuit sa fie indirectă, așadar am decis ca numerele pentru operațiile descrise să fie stocate în memoria RAM din

cadrul componentei de Memory ilustrată mai sus în capitolul de implementare. Pentru memoria în care sunt stocate instrucțiunile, a fost necesar ca adresarea să fie de tip stivă. La nivel intern, implementarea operațiilor a fost realizată prin cascadări, întrucât a fost necesar să folosesc porți logice și bistabile.