**IMPLEMENTARE PROCESOR RISC-V**

**Candidat: Andrei-Constantin VARODI**

**Coordonator științific: Ș.l. dr.ing. Eugen-Horațiu GURBAN**

Sesiunea: Septembrie 2023

**Cuprins**

**Cap 1: Introducere**...............................................................................................................3

1.1 Contextul temei................................................................................................................3

1.2 Motivația..........................................................................................................................3

**Cap 3: Fundamente teoretice** .............................................................................................4

3.1 Limbaj de asamblare.......................................................................................................4

3.2 VHDL ..............................................................................................................................5

**Cap 4: Implementarea soluției**............................................................................................6

4.1 Specificarea cerințelor.....................................................................................................7

4.2 Memoria de instructiuni....................................................................................................8

4.3 Blocul de registrii.............................................................................................................9

4.4 Unitatea aritmetica-logica..............................................................................................15

4.5 Memoria de date............................................................................................................19

4.6 Procesorul.....................................................................................................................22

**Cap 5: Manual de utilizare**.................................................................................................32

**Cap 6: Concluzie** ...............................................................................................................38

**Bibliografie**........................................................................................................................39

**1. INTRODUCERE**

**1.1 CONTEXTUL TEMEI**

Arhitectura RISC-V (Reduced Instruction Set Computer – V) este o arhitectură open-source bazată pe principiul unui set redus de instrucțiuni [1]. Această arhitectură a captat atenția comunității de dezvoltatori pentru că aduce cu sine o serie de caracteristici și potențiale avantaje care pot transforma felul în care gândim despre dezvoltarea procesoarelor.

Procesoarele care au evoluat pe parcursul deceniilor precum Intel x86, ARM și multe altele, au adăugat un număr impresionant de instrucțiuni specifice pentru a răspunde nevoilor diverse ale aplicațiilor astfel formându-se procesoare extrem de complexe și costisitoare, iar dezvoltarea software pentru aceste arhitecturi a devenit tot mai dificilă. Spre deosebire de aceste procesoare, RISC-V în loc să adauge tot mai multe instrucțiuni, optează pentru un set de bază simplu, cu un număr mic de instrucțiuni. Aceste instrucțiuni sunt concepute pentru a fi extrem de eficiente fiind executate într-un singur ciclu de ceas. Simplitatea acestui set de instrucțiuni permite dezvoltatorilor să înțeleagă mai ușor funcționarea procesorului care ajută la o dezvoltare software mai rapidă.

De asemenea, RISC-V este proiectat pentru a fi modular și extensibil. Acest lucru înseamnă că se pot adaugă funcționalități suplimentare la procesor în funcție de necesități. De exemplu, dacă dezvoltați un sistem de calcul intensiv în matematică se pot adaugă instrucțiuni vectoriale pentru a accelera operațiile matematice.

**1.2 MOTIVAȚIA**

Motivația principală care m-a determinat să mă angajez în acest proiect a fost explorarea arhitecturii hardware pentru a obține experiență de devoltare la nivel scăzut. Am avut mereu o pasiune pentru tehnologia hardware și pentru modul în care funcționează calculatoarele și procesoarele. Implementarea unui procesor RISC-V a reprezentat o oportunitate excelență de a învăța cum să proiectez și dezvolt un procesor.

**3. FUNDAMENTE TEORETICE**

În acest capitol voi explica fundamentele teoretice folosite pentru dobândirea cunostiintelor necesare pentru implementarea temei alese împreună cu câteva generalități.

**3.1 LIMBAJE DE ASAMBLARE**

Un limbaj de asamblare este un limbaj de programare a calculatoarelor care, în locul codului mașină, folosește o desemnare simbolică a elementelor programului, relativ ușor de citit și interpretat de către om [2]. Programarea în limbaj de asamblare presupune o bună cunoaștere a structurii procesorului și a componentelor sale adiacente[3]. Ea face ca utilizatorul să aibă acces la toate facilitățile unui calculator, dar programul rezultat va putea funcționa numai pe un tip de procesor deoarece fiecare tip de procesor deține propriul său limbaj de asamblare. Dacă programul trebuie portat pe mai multe tipuri de calculatoare, atunci se preferă limbajele de programare de nivel mai înalt.

A table with numbers and symbols

Description automatically generatedLimbajele de asamblare folosesc simboluri (nume) și mnemonice (coduri scrute) pentru a reprezenta instrucțiunile și adresele de memorie[2]. Acestea fac ca codul să fie mai ușor de înțeles și de scris pentru programator decât codul binar direct. Programarea în astfel de limbaj poate duce la cod extrem de eficient și performant deoarece programatorii au control total asupra resurselor hardware de aceea acest tip de programare este deosebit de important în dezvoltarea de drivere pentru dispozitive, sisteme de operare și aplicații cu cerințe stricte de performantă [3]. Învățarea acestui tip de limbaj este, în general, mai dificil comparativ cu un limbaj de nivel înalt precum Java, datorită detaliilor legate de arhitectura specifică procesorului. Scrierea și depanarea codului asamblor poate fi un proces laborios care necesită o înțelegere profundă a hardware-ului.

Figura 4.1 –Exemplu instrucțiuni limbaj de asamblare RISC-V (Sursa: [4] figura 3.12)

**3.2 VHDL**

VHDL a luat naștere în laboratoarele de cercetare ale Departamentului de Apărare al SUA în anii '80. Inițial dezvoltat pentru a permite proiectarea și testarea circuitelor electronice, VHDL a evoluat rapid și a devenit un standard în industria de design electronic. Unul dintre aspectele sale remarcabile este abilitatea de a descrie atât circuite digitale, cât și sisteme mixte care combină componente analogice și digitale. VHDL este acronim de la VHSIC Hardware Description Language care conține alt acronim VHSIC însemnând Very High-Speed Integrated Circuit. [5]

VHDL este un adevărat limbaj de programare acompaniat de un set de sintaxă și reguli de folosire, dar spre deosebire de limbajele de nivel înalt, VHDL este folosit în principal la descriere hardware. Cea mai comună greșeală făcută de persoanele care doresc să învețe acest limbaj de programare este că îl consideră a fi un limbaj de programare de nivel înalt precum C sau Java care natural se execută secvențial. Codul scris în VHDL este executat în paralel în același timp. [6]

VHDL are două roluri principale: sinteza și simularea. Sinteza se referă la procesul prin care descrierea VHDL este transformată într-un model hardware real. Cu ajutorul instrumentelor de sinteză putem crea circuite integrate (IC) și matrici de porți programabile (FPGA). Pe de altă parte, simularea ne permite să testăm și să validăm comportamentul sistemului fară a crea un hardware fizic pentru teste. [6]

O carcateristica fundamentală a VHDL este organizarea sa în entități și arhitecturi. O entitate definește interfața unui bloc de hardware stabilind semnalele de intrare și ieșire, în timp ce arhitectura descrie implementarea reală a entității. Această abordare modulară permite programatorilor să creeze componente hardware reutilizabile și să le integreze în diferite proiecte sporind eficiența dezvoltării. [7]

**4. IMPLEMENTAREA SOLUȚIEI**

Modelul procesorului RISC-V implementat este alcătuit din 4 componente:

-memorie de instrucțiuni, această memorie conține programul de asamblare translatat în program mașină pentru a fi înțeles de către procesor.

-blocul de registrii, conține valorile registrilor utilizați pe parcursul rulării programului.

-unitate aritmetică-logică, conține setul de operații care pot fi executate de către procesor.

-memoria de date este memoria în care sunt salvate și citite informații pentru procesor.

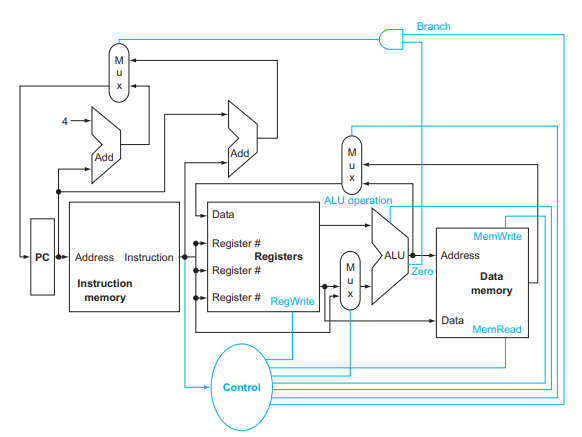
 Aceste 4 componente vor fi instanțiate în entitatea procesorului împreună cu legăturile logice reprezentate pe figura 1.

Figura 4.1 – Implementarea de bază a procesorului RISC-V, incluzând multiplexoarele și liniile de control necesare. (Sursa: [4] figura 4.2)

Toate componentele vor fi implementate și testate folosind limbajul VHDL în mediul de programare Active-HDL Student Edition.

**4.1 SPECIFICARE CERINȚELOR**

-implementarea memoriei de instrucțiuni (InstructionMemory entity)

-creare de test bench pentru a verifica funcționarea corectă a memoriei de instrucțiuni

-implementarea blocului de registrii (Registers entity)

-creare de test bench pentru a verifica funcționarea corectă a blocului de registrii

-implementarea unității aritmetico-logice (ALU entity)

-creare de test bench pentru a verifica funcționarea corectă a unității aritmetico-logice

-implementarea memoriei de date (DataMemory entity)

-creare de test bench pentru a verifica funcționarea corectă a memoriei de date

-implementarea procesorului (RISC\_V\_Processor entity)

-creare de test bench pentru a verifica funcționarea corectă a procesorului

**4.2 MEMORIA DE INSTRUCȚIUNI**

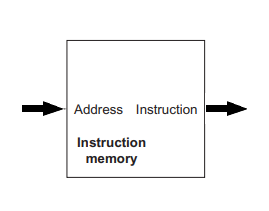
**SCHEMA**

Figura 4.2 – Schema bloc memoria de instrucțiuni

Din figura 4.2 se poate deduce faptul că componenta de memorie pentru instrucțiuni primește că valoare de intrare o adresă și va expune ca valoare de ieșire instrucțiunea aflată la adresa respectivă.

**ENTITATE**

entity **InstructionMemory** is

    port (

        clk : in STD\_LOGIC;

        address : in STD\_LOGIC\_VECTOR(31 downto 0);

        instruction : out STD\_LOGIC\_VECTOR(31 downto 0);

        memory : in mem\_array

    );

end entity **InstructionMemory**;

-semnalul clk de tip in STD\_LOGIC, este folosit drept ceasul componentei.

-semnalul address de tip in STD\_LOGIC\_VECTOR(31 downto 0), este folosit drept intrare pentru adresa dată de către procesor pentru a afla următoarea instrucțiune.

-semnalul memory de tip in mem\_array, este un semnal custom de intrare care are scopul de a introduce în memorie un set de instrucțiuni.

-semnalul instruction de tip out STD\_LOGIC\_VECTOR(31 downto 0), este folosit drept semnal de ieșire pe care va fi transmisă instrucțiunea din memorie prezentă la adresa data că semnal de intrare

**ARHITECTURA**

architecture **Behavioral** of **InstructionMemory** is

begin

 process(clk)

    begin

        if **rising\_edge**(clk) then

            instruction <= memory(**to\_integer**(unsigned(address)));

        end if;

    end process;

end architecture **Behavioral**;

Arhitectura este compusă dintr-un singur proces.

La fiecare front urcător al semnalului clk va fi executat corpul structurii if în care semnalul de ieșire „instruction” va primi valoare aflată în memorie la adresa dată de semnalul de intrare „address”.

**TESTBENCH**

entity **InstructionMemory\_TB** is

end **InstructionMemory\_TB**;

architecture **Behavioral** of **InstructionMemory\_TB** is

    component **InstructionMemory**

        port (

            clk : in STD\_LOGIC;

            address : in STD\_LOGIC\_VECTOR(31 downto 0);

            instruction : out STD\_LOGIC\_VECTOR(31 downto 0);

            memory : in mem\_array

        );

    end component;

    signal tb\_clk : STD\_LOGIC := '0';

    signal tb\_address : STD\_LOGIC\_VECTOR(31 downto 0) := (others => '0');

    signal tb\_instruction : STD\_LOGIC\_VECTOR(31 downto 0);

    signal tb\_memory : mem\_array;

begin

    tb\_memory(0) <= X"00C30333";  *-- add x1, x2, x3*

    tb\_memory(1) <= X"40C30333";  *-- sub x1, x2, x3*

    tb\_clk <= not tb\_clk after 5ns;

    tb\_address <= (others => '0'), (0 => '1', others => '0') after 15ns, (others => '0') after 25ns;

**uut** : InstructionMemory port map (tb\_clk, tb\_address, tb\_instruction, tb\_memory);

end **Behavioral**;

Pentru testarea componentei, fiecare semnal din entitate va avea un semnal de testbench care vor fi schimbate în timp pentru a testa toate cazurile. Semnalul pentru memorie nu va fi schimbat pe decursul testului.

A screenshot of a computer

Description automatically generatedAcest test va genera următoarea formă de unde:

Figura 4.3 – Formele de undă generate in urma rulării testului pentru memoriei de instrucțiuni

Pe forma de undă se pot observa următoarele:

La începutul programului, echivalent cu 0ns:

-semnalul tb\_clk are valoarea ‚0’ logic

-semnalul tb\_address are valoarea de adresa 0

-semnalul tb\_instruction are valoarea ‚U’ din cauză că nu este definit, va primi valori diferite pentru fiecare front urcător a semnalului tb\_clk.

La 5ns și repetitiv o dată la 10ns trecute, semnalul tb\_clk va fi pe front urcător astfel că semnalul tb\_instruction va primi valoare ce se află în memorie la adresa dată de semnalul tb\_address. De exemplu, la momentul de timp 15ns, marcat cu o linie roșie, semnalul tb\_address are valoarea „00000001” (tb\_memory[1]) care face referire la a doua adresă din memorie în care este salvată instrucțiunea „40C30333” care va fi alocată semnalului tb\_instruction.

Semnalul tb\_memory conține multe linii nedefinite deoarece pentru acest test am inițializat doar primele două adrese din memorie

**4.3 BLOCUL DE REGIȘTRII**

**SCHEMA**

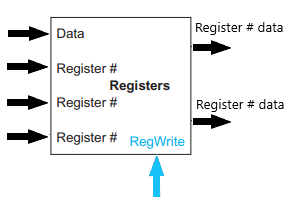


Figura 4.4 – Schema bloc a regiștrilor

Din figura 4.4 putem deduce că blocul de registrii conține ca semnale de intrare 3 adrese pentru regiștrii, primite de la procesor în urma decodării instrucțiunii, un semnal de data pentru a salva date în regiștrii și un semnal RegWrite primit de la procesor. Ca semnale de ieșire vor fi expuse datele de pe regiștrii primiți ca intrare.

**ENTITATE**

entity **Registers** is

    port (

        clk : in STD\_LOGIC;

        read\_reg1 : in STD\_LOGIC\_VECTOR(4 downto 0);

        read\_reg2 : in STD\_LOGIC\_VECTOR(4 downto 0);

        write\_reg : in STD\_LOGIC\_VECTOR(4 downto 0);

        write\_data : in STD\_LOGIC\_VECTOR(31 downto 0);

        reg\_write\_enable : in STD\_LOGIC;

        read\_data1 : out STD\_LOGIC\_VECTOR(31 downto 0);

        read\_data2 : out STD\_LOGIC\_VECTOR(31 downto 0)

    );

end **Registers**;

-semnalul clk de tip in STD\_LOGIC, este folosit drept ceasul componentei.

-semnalul read\_reg1 de tip in STD\_LOGIC\_VECTOR(4 downto 0), este primul registru sursă aflat din decodarea instrucțiunii.

-semnalul read\_reg2 de tip in STD\_LOGIC\_VECTOR(4 downto 0), este al doilea registru sursă aflat din decodarea instrucțiunii.

-semnalul write\_reg de tip in STD\_LOGIC\_VECTOR(4 downto 0), este registru de destinație aflat din decodarea instrucțiunii, în acest registru vă fi salvat rezultatul instrucțiunii.

-semnalul write\_data de tip in STD\_LOGIC\_VECTOR(31 downto 0), este data care va fi salvată în registrul destinație.

-semnalul reg\_write\_enable de tip in STD\_LOGIC, controlează în ce stadiu se află blocul de registrii. Pentru valoarea logică ‚1’ semnalul data va salva datele în registrul destinație, iar pentru valoarea logică ‚0’ se va afișa doar datele din cei 2 registrii sursă fară a se face salvare de date în registrul destinație.

-semnalul read\_data1 de tip out STD\_LOGIC\_VECTOR(31 downto 0), este semnal de ieșire pe care se va afla datele prezente în primul registru sursă.

-semnalul read\_data2 de tip out STD\_LOGIC\_VECTOR(31 downto 0), este semnal de ieșire pe care se va afla datele prezente în al doilea registru sursă.

**ARHITECTURA**

architecture **Behavioral** of **Registers** is

    type **reg\_data\_array** is array (0 to 31) of STD\_LOGIC\_VECTOR(31 downto 0);

    signal registers : reg\_data\_array := (others => (others => '0'));

begin

    process(clk)

    begin

        if **rising\_edge**(clk) then

            if reg\_write\_enable = '1' then

                if not(write\_reg(0) = '0' and write\_reg(1) = '0' and write\_reg(2) = '0') then

                    registers(**to\_integer**(unsigned(write\_reg))) <= write\_data;

                end if;

            end if;

            read\_data1 <= registers(**to\_integer**(unsigned(read\_reg1)));

            read\_data2 <= registers(**to\_integer**(unsigned(read\_reg2)));

        end if;

    end process;

end **Behavioral**;

Arhitectura este compusă dintr-un singur proces care verifică dacă semnalul clk se află pe un front urcător. Dacă acesta se află pe front urcător atunci valoarea semnalelor de ieșire, read\_data1 și read\_data2, primesc valoarile aflate în memorie pentru read\_reg1, respectiv read\_reg1. Dacă semnalul clk se află pe front urcător și semnalul reg\_write\_enable are valoarea digitală ‚1’ și registrul primit prin semnalul write\_reg este diferit de registrul x0 atunci în memorie la registrul respectiv se va scrie datele primite prin semnalul de intrare write\_data.

Memoria blocului de registrii va avea toți biții setați pe ‚0’.

**TESTBENCH**

entity **Registers\_TB** is

end **Registers\_TB**;

architecture **Behavioral** of **Registers\_TB** is

    component **Registers**

        port (

            clk : in STD\_LOGIC;

            read\_reg1 : in STD\_LOGIC\_VECTOR(4 downto 0);

            read\_reg2 : in STD\_LOGIC\_VECTOR(4 downto 0);

            write\_reg : in STD\_LOGIC\_VECTOR(4 downto 0);

            write\_data : in STD\_LOGIC\_VECTOR(31 downto 0);

            reg\_write\_enable : in STD\_LOGIC;

            read\_data1 : out STD\_LOGIC\_VECTOR(31 downto 0);

            read\_data2 : out STD\_LOGIC\_VECTOR(31 downto 0)

        );

    end component;

    signal tb\_clk : STD\_LOGIC := '0';

    signal tb\_read\_reg1, tb\_read\_reg2, tb\_write\_reg : STD\_LOGIC\_VECTOR(4 downto 0);

    signal tb\_write\_data : STD\_LOGIC\_VECTOR(31 downto 0);

    signal tb\_reg\_write\_enable : STD\_LOGIC;

    signal tb\_read\_data1, tb\_read\_data2 : STD\_LOGIC\_VECTOR(31 downto 0);

begin

    tb\_clk <= not(tb\_clk) after 5ns;

*-- Write data to register 2*

    tb\_write\_data <= X"12345678";

    tb\_write\_reg <= "00010";

    tb\_reg\_write\_enable <= '0', '1' after 5ns, '0' after 10ns;

*-- Read data from registers 2 and 1*

    tb\_read\_reg1 <= "00010";

    tb\_read\_reg2 <= "00001";

**uut**: Registers port map (tb\_clk, tb\_read\_reg1, tb\_read\_reg2, tb\_write\_reg, tb\_write\_data, tb\_reg\_write\_enable, tb\_read\_data1, tb\_read\_data2);

end **Behavioral**;

Pentru testarea componentei, fiecare semnal din entitate va avea un semnal de testbench care vor fi schimbate în timp pentru a testa toate cazurile.

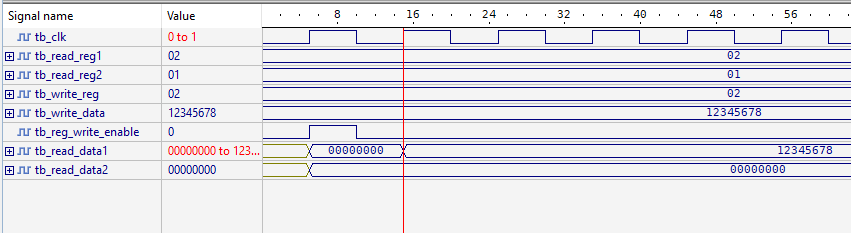
Acest test va genera următoarea formă de unde:

Figura 4.5 – Formele de undă generate in urma rulării testului pentru blocul de regiștrii

Pe forma de undă se pot observa următoarele:

La începutul programului, echivalent cu 0ns:

- tb\_clk are valoarea ‚0’ logic

- tb\_read\_reg1 are valoarea 02 făcând referire la registrul x2

- tb\_read\_reg2 are valoarea 01 făcând referire la registrul x1

- tb\_write\_reg are valoarea 02 făcând referire la registrul x2

- tb\_write\_data are valoarea 12345678, valoare aleatorie pentru testare

- tb\_reg\_write\_enable are valoarea 0, înseamnă că scrierea în registrii este oprită

- tb\_read\_data1 și tb\_read\_data2 au valori nedefinite deoarece vor primi valori doar când semnalul tb\_clk va fi pe front urcător

La 5ns și repetitiv o dată la 10ns trecute, semnalul tb\_clk va fi pe front urcător astfel că tb\_read\_data1 și tb\_read\_data2 vor primi datele salvate în memorie la registrii primiți prin semnalele tb\_read\_reg1, respectiv tb\_read\_reg2.

La 5ns valoarea semnalului tb\_reg\_write\_enable are valoarea 1 însemnând că în memorie la registrul primit prin semnalul tb\_write\_reg, registrul x2, vor fi salvate datele din semnalul tb\_write\_data. La următorul front urcător al semnalului tb\_clk, marcat cu linie roșie, se va afișa noua valoare a registrului x2.

**4.4 UNITATE ARITMETICĂ-LOGICĂ (ALU)**

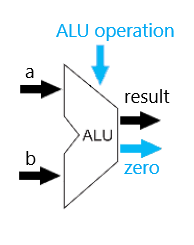
**SCHEMA**

Figura 4.6 – Schema bloc unitate aritmetică-logică

Din figura 4.6 putem deduce că unitatea aritmetică-logică are 3 semnale de intrare, a și b fiind operanzii ecuațiilor, iar ALU operation este un cod pentru selectarea operației executate. Ca semnale de ieșire sunt 2, result conține rezultatul operației, iar zero are valoarea ‚1’ logic dacă rezultatul este 0.

**ENTITATE**

entity **ALU** is

    port (

        op : in STD\_LOGIC\_VECTOR(2 downto 0);

        a : in STD\_LOGIC\_VECTOR(31 downto 0);

        b : in STD\_LOGIC\_VECTOR(31 downto 0);

result : out STD\_LOGIC\_VECTOR(31 downto 0);

        zero : out STD\_LOGIC

    );

end **ALU**;

-semnalul op de tip in STD\_LOGIC\_VECTOR(2 downto 0), este folosit pentru a decide ce tip de operație va fi executată

-semnalul a de tip in STD\_LOGIC\_VECTOR(31 downto 0), este valoarea primului operand.

-semnalul b de tip in STD\_LOGIC\_VECTOR(31 downto 0), este valoarea al doilea operand.

-semnalul result de tip out STD\_LOGIC\_VECTOR(31 downto 0), este valoarea rezultatului operației dintre a și b.

-semnalul zero de tip out STD\_LOGIC are valoarea ‚1’ logic dacă rezultatul este 0, respectiv are valoarea ‚0’ logic dacă rezultatul este diferit de 0.

**ARHITECTURA**

architecture **Behavioral** of **ALU** is

signal temp\_result : STD\_LOGIC\_VECTOR(31 downto 0);

begin

    process

    begin

        wait on op, a, b;

        case op is

            when "000" => *-- Addition*

                temp\_result <= std\_logic\_vector(unsigned(a) + unsigned(b));

            when "001" => *-- Subtraction*

                temp\_result <= std\_logic\_vector(unsigned(a) - unsigned(b));

            when "010" => *-- Bitwise AND*

                temp\_result <= a and b;

            when "011" => *-- Bitwise OR*

                temp\_result <= a or b;

            when others =>

                temp\_result <= (others => 'U'); *-- Default: Set result to undifined*

        end case;

*-- Check if result is zero*

        if temp\_result = X"00000000" then

            zero <= '1';

        else

            zero <= '0';

        end if;

        wait on temp\_result;

        result <= temp\_result;

    end process;

end **Behavioral**;

La fel ca la restul componentelor, arhitectura este compusă dintr-un singur proces dar de data aceasta procesul nu rulează pe frontul urcător al semnalului de clock. Procesul așteaptă modificarea oricărui din semnalele a, b sau op. Semnalul op este parametru pentru structura de switch, în funcție de valoarea acestuia se vă executa o operație între a și b, iar rezultatul va fi salvat într-un semnal temporar numit temp\_result. Am ales salvarea rezultatului într-un semnal temporar pentru că un semnal de tip out, tipul semnalului original, nu poate fi comparat în condițiile pentru structura if. Dacă valoarea semnalului temp\_result este 0 atunci îi va asocia

semnalului zero valoarea 0, în caz contrar 1. Semnalul final de ieșire result va primi valoarea după „wait on temp\_result” care pune procesul în așteptare până semnalul temp\_result își schimbă valoarea. Folosind acest wait, semnalul result își va schimba valoarea instant, dacă nu era folosit acest wait atunci valoarea semnalului result ar fi fost schimbată la următoarea rulare a procesului cu valoarea anterioare producând astfel un decalaj între operația executată și rezultatul ei.

**TESTBENCH**

entity **ALU\_Testbench** is

end **ALU\_Testbench**;

architecture **tb\_arch** of **ALU\_Testbench** is

    component **ALU**

        port (

            op : in STD\_LOGIC\_VECTOR(2 downto 0);

            a : in STD\_LOGIC\_VECTOR(31 downto 0);

            b : in STD\_LOGIC\_VECTOR(31 downto 0);

            result : out STD\_LOGIC\_VECTOR(31 downto 0);

            zero : out STD\_LOGIC

        );

    end component;

    signal tb\_op : STD\_LOGIC\_VECTOR(2 downto 0);

    signal tb\_a : STD\_LOGIC\_VECTOR(31 downto 0);

    signal tb\_b : STD\_LOGIC\_VECTOR(31 downto 0);

    signal tb\_result : STD\_LOGIC\_VECTOR(31 downto 0);

    signal tb\_zero : STD\_LOGIC;

begin

**uut** : ALU port map (tb\_op, tb\_a, tb\_b, tb\_result, tb\_zero);

    process

    begin

*-- Test case 1: Addition*

        tb\_a <= X"00000004"; *-- 4*

        tb\_b <= X"00000006"; *-- 6*

        tb\_op <= "000";

        wait for 10 ns;

*-- Test case 2: Subtraction*

        tb\_a <= X"00000008"; *-- 8*

        tb\_b <= X"00000005"; *-- 5*

        tb\_op <= "001";

        wait for 20 ns;

*-- Test case 3: Bitwise AND*

        tb\_a <= X"0000000C"; *-- 12*

        tb\_b <= X"00000007"; *-- 7*

        tb\_op <= "010";

        wait for 20 ns;

*-- Test case 4: Bitwise OR*

        tb\_a <= X"0000000A"; *-- 10*

        tb\_b <= X"00000006"; *-- 6*

        tb\_op <= "011";

        wait for 20 ns;

        tb\_op <= "UUU";

        wait;

    end process;

end **tb\_arch**;

Pentru testarea componentei, fiecare semnal din entitate va avea un semnal de testbench care vor fi schimbate în timp pentru a testa toate cazurile.

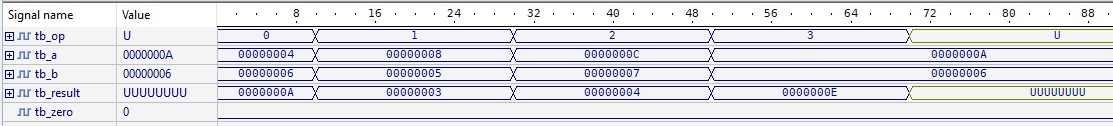
Acest test va genera următoarea formă de unde:

Figura 4.7 – Formele de unda generate in urma rularii testului pentru unitatea aritmetica-logica

După cum se poate observa în figura 4.7, răspunsul unității aritmetico-logice este imediată. Primul caz are semnalul tb\_op setat pe 0 care înseamnă adunare, primul operand, semnalul tb\_a, are valoarea 4, al doilea operand, semnalul tb\_b, are valoarea 6, iar rezultatul, semnalul tb\_result, are valoarea A în baza hexazecimala ce e convertit în 10 pentru baza zecimală.

**4.5 MEMORIA DE DATE**

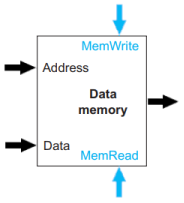
**SCHEMA**

Figura 4.8 – Schema bloc memorie de date

Din figura 4.8 se deduce că componentat memorie de date are 4 semnale de intrare. Prin semnalul address va primi locația adresei cu care va lucra. În funcție de semnalele memWrite și memRead componenta are două moduri de funcționare. Când semnalul MemWrite are valoarea ‚1’ logic atunci la adresa primita că intrare se vor scrie datele primite prin semnalul Data, în caz contrar când MemWrite are valoarea ‚0’ logic atunci nu vă executa nimic. Pentru al doilea mod de funcționare, când semnalul MemRead are valoarea ‚1’ logic atunci componenta va citi datele aflate pe adresa primită ca semnal de intrare și le va expune pe semnalul de ieșire, în caz contrar când MemRead are valoarea ‚0’ logic atunci nu vă executa nimic.

**ENTITATE**

entity **DataMemory** is

    port (

        clk : in STD\_LOGIC;

        address : in STD\_LOGIC\_VECTOR(31 downto 0);

        write\_data : in STD\_LOGIC\_VECTOR(31 downto 0);

        mem\_write\_enable : in STD\_LOGIC;

        read\_data : out STD\_LOGIC\_VECTOR(31 downto 0);

        mem\_read\_enable : in STD\_LOGIC

    );

end **DataMemory**;

-semnalul clk de tip in STD\_LOGIC, este folosit drept ceasul componentei.

-semnalul address de tip in STD\_LOGIC\_VECTOR(31 downto 0), este folosit drept intrare pentru adresa pe care se vor executa operațiile de scriere sau citire.

-semnalul write\_data de tip in STD\_LOGIC\_VECTOR(31 downto 0) oferă datele care urmează a fi scrise la adresa primită ca semnal de intrare.

-semnalul mem\_write\_enable de tip in STD\_LOGIC indica daca în memorie la adresa primită se va executa operația de scriere sau nu.

-semnalul read\_data de tip out STD\_LOGIC\_VECTOR(31 downto 0) va expune înafara componentei datele de la adresa citită.

-semnalul mem\_read\_enable de tip in STD\_LOGIC indică dacă în memorie la adresa primita se va executa operația de citire sau nu.

**ARHITECTURA**

architecture **Behavioral** of **DataMemory** is

    signal memory : mem\_array := (others => (others => '0'));

begin

    process(clk)

    begin

        if **rising\_edge**(clk) then

            if mem\_write\_enable = '1' then

                memory(**to\_integer**(unsigned(address))) <= write\_data;

            end if;

            if mem\_read\_enable = '1' then

                read\_data <= memory(**to\_integer**(unsigned(address)));

            end if;

        end if;

    end process;

end **Behavioral**;

Arhitectura este compusă dintr-un singur proces.

La fiecare front urcător al semnalului clk va fi executat corpul structurii if în care se află două structuri if corespunzătoare pentru cele două funcționari ale componentei. Dacă semnalul mem\_write\_enable are valoarea ‚1’ logic atunci în memorie la adresa respectivă se vor scrie datele din semnalul write\_data. Daca semnalul mem\_read\_enable are valoarea ‚1’ logic atunci semnalul de ieșire read\_data va primi datele aflate în memorie la adresa respectivă.

**TESTBENCH**

entity **DataMemory\_TB** is

end **DataMemory\_TB**;

architecture **Behavioral** of **DataMemory\_TB** is

    signal tb\_clk : STD\_LOGIC := '0';

    signal tb\_address : STD\_LOGIC\_VECTOR(31 downto 0) := (others => '0');

    signal tb\_write\_data : STD\_LOGIC\_VECTOR(31 downto 0);

    signal tb\_mem\_write\_enable : STD\_LOGIC := '0';

    signal tb\_read\_data : STD\_LOGIC\_VECTOR(31 downto 0);

    signal tb\_mem\_read\_enable : STD\_LOGIC := '0';

    component **DataMemory**

        port (

            clk : in STD\_LOGIC;

            address : in STD\_LOGIC\_VECTOR(31 downto 0);

            write\_data : in STD\_LOGIC\_VECTOR(31 downto 0);

            mem\_write\_enable : in STD\_LOGIC;

            read\_data : out STD\_LOGIC\_VECTOR(31 downto 0);

            mem\_read\_enable : in STD\_LOGIC

        );

    end component;

begin

    tb\_clk <= not tb\_clk after 5ns;

    tb\_write\_data <= X"ABCDEF01";

    tb\_address <= (0 => '1', others => '0');

    tb\_mem\_write\_enable <= '1' after 10 ns, '0' after 25 ns;

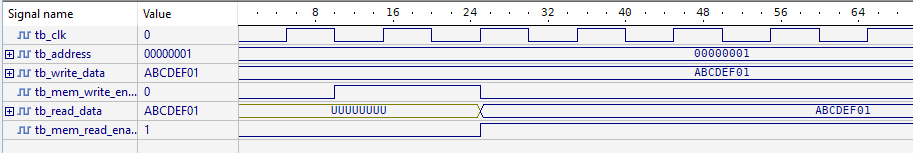
    tb\_mem\_read\_enable <= '1' after 25ns;

**uut**: DataMemory port map (tb\_clk, tb\_address, tb\_write\_data, tb\_mem\_write\_enable, tb\_read\_data, tb\_mem\_read\_enable);

end **Behavioral**;

Pentru testarea componentei, fiecare semnal din entitate va avea un semnal de testbench care vor fi schimbate în timp pentru a testa toate cazurile. Semnalul pentru memorie nu va fi schimbat pe decursul testului.

Acest test va genera următoarea formă de unde:

 Figura 4.9 – Formele de unda generate in urma rulării testului pentru memoria de date

Din forma de undă se pot observa următoarele:

-semnalul tb\_clk începe cu valoarea ,0’ și va fi pe un front urcător la fiecare 10ns trecute, primul front urcător find la 5ns.

-semnalul tb\_address are aceeași valoare pe parcursul testului pentru că folosim aceeași adresa.

-semnalul tb\_read\_data are aceeași valoare pe parcursul testului pentru că folosim aceleași date.

-semnalele tb\_mem\_write\_enable și tb\_mem\_read\_enable încep cu valoarea ‚0’ logic, de aceea la început semnalul tb\_read\_data are valoare nedefinită, componenta fiind în niciuna dintre cele 2 funcționări.

La momentul de 15ns în memorie se vă scrie datele din semnalul tb\_data\_write deoarce este un front crescător al semnalului tb\_clk și semnalul tb\_mem\_write are valoarea ‚1’ logic dar această modificare a datelor din memorie se vă observa doar la momentul de 25ns când semnalul tb\_mem\_read\_enable are valoarea de ‚1’ logic astfel punând componenta în modul de citire.

**4.6 PROCESORUL**

**SCHEMA**

Schema procesorului este reprezentata in figura 4.1.

**ENTITATE**

entity **RISC\_V\_Processor** is

        port (

*-- clock for all components*

            clk : in STD\_LOGIC;

*--signals for instruction memory component*

            instructions\_memory : in mem\_array;

            pc : out STD\_LOGIC\_VECTOR(31 downto 0);

            instruction : out STD\_LOGIC\_VECTOR(31 downto 0);

*--signals for registers component*

            rs1 : out STD\_LOGIC\_VECTOR(4 downto 0);

            rs2 : out STD\_LOGIC\_VECTOR(4 downto 0);

            rd : out STD\_LOGIC\_VECTOR(4 downto 0);

            rd\_write\_data : out STD\_LOGIC\_VECTOR(31 downto 0);

            rd\_write\_enable : out STD\_LOGIC;

            rs1\_data : out STD\_LOGIC\_VECTOR(31 downto 0);

            rs2\_data : out STD\_LOGIC\_VECTOR(31 downto 0);

*--signals for ALU component*

            alu\_op : out STD\_LOGIC\_VECTOR(2 downto 0);

            alu\_a : out STD\_LOGIC\_VECTOR(31 downto 0);

            alu\_b : out STD\_LOGIC\_VECTOR(31 downto 0);

            alu\_result : out STD\_LOGIC\_VECTOR(31 downto 0);

            alu\_zero : out STD\_LOGIC;

*--signals for data memory component*

            mem\_address : out STD\_LOGIC\_VECTOR(31 downto 0);

            mem\_write\_data : out STD\_LOGIC\_VECTOR(31 downto 0);

            mem\_write\_enable : out STD\_LOGIC;

            mem\_read\_data : out STD\_LOGIC\_VECTOR(31 downto 0);

mem\_read\_enable : out STD\_LOGIC;

*--aditionals signals for decoding instructions*

            funct7 : out STD\_LOGIC\_VECTOR(6 downto 0);

            funct3 : out STD\_LOGIC\_VECTOR(2 downto 0);

            opcode : out STD\_LOGIC\_VECTOR(6 downto 0);

            funct : out STD\_LOGIC\_VECTOR(9 downto 0)

        );

end **RISC\_V\_Processor**;

Entitatea procesorului conține semnalele tuturor componentelor pentru a le putea monitoriza în timpul testelor. Pe lângă semnalele componentelor, entitatea procesorului conține și semnale adiționale pentru decodarea intructiuniilor. Aceste semnale fiind funct7, funct3, opcode și funct. Toatea semnalele provenite din componente și cele adiționale de decodarea instructiuniilor vor fi te tip out pentru a le putea atribui valori în timpul rulării. Un semnal de tip in nu își poate schimba valoarea pe parcursul programului, din acest motiv se vor folosi semnalele „temp\_” de tip signal pentru a fi folosite în procesul din arhitectură. Semnalele de tip signal pot să fie citite și scrise oricând pe parcursul procesului.

**ARHITECTURA**

Arhitectura procesorului este mult mai mare decât a fiecărei componente în parte, dar este formată tot dintr-un singur proces. De aceea nu va fi tot codul inserat, în schimb vor fi explicate doar bucățile importante din cod.

Pentru a folosi fiecare componenta creată pentru procesor ele trebuie să fie declarate în arhitectura procesorului în felul descris mai jos. Fiecare semnal din entitatea procesorului va avea un semnal intern care vor fi alocate componentelor în etapa de istantiere. Aceste semnale vor avea aceeași denumire cu semnalul original la care se va adaugă prefixul „temp\_”.

component **InstructionMemory**

port (

            clk : in STD\_LOGIC;

            address : in STD\_LOGIC\_VECTOR(31 downto 0);

            instruction : out STD\_LOGIC\_VECTOR(31 downto 0);

            memory : in mem\_array

        );

    end component;

    component **Registers** is

        port (

            clk : in STD\_LOGIC;

            read\_reg1 : in STD\_LOGIC\_VECTOR(4 downto 0);

            read\_reg2 : in STD\_LOGIC\_VECTOR(4 downto 0);

            write\_reg : in STD\_LOGIC\_VECTOR(4 downto 0);

            write\_data : in STD\_LOGIC\_VECTOR(31 downto 0);

            reg\_write\_enable : in STD\_LOGIC;

            read\_data1 : out STD\_LOGIC\_VECTOR(31 downto 0);

            read\_data2 : out STD\_LOGIC\_VECTOR(31 downto 0)

        );

    end component;

    component **ALU** is

    port (

        op : in STD\_LOGIC\_VECTOR(2 downto 0);

        a : in STD\_LOGIC\_VECTOR(31 downto 0);

        b : in STD\_LOGIC\_VECTOR(31 downto 0);

        result : out STD\_LOGIC\_VECTOR(31 downto 0);

        zero : out STD\_LOGIC

    );

    end component;

    component **DataMemory**

        port (

            clk : in STD\_LOGIC;

            address : in STD\_LOGIC\_VECTOR(31 downto 0);

            write\_data : in STD\_LOGIC\_VECTOR(31 downto 0);

            mem\_write\_enable : in STD\_LOGIC;

            read\_data : out STD\_LOGIC\_VECTOR(31 downto 0);

            mem\_read\_enable : in STD\_LOGIC);

    end component;

Instanțierea componentelor, spre deosebire de declararea lor și a semnalelor interne, va avea loc în corpul begin al arhitecturii în paralel cu procesul care rulează logica procesorului.

*-- Instantiate Instruction Memory*

**instruction\_memory** : InstructionMemory port map (clk, temp\_pc, temp\_instruction, instructions\_memory);

*-- Instantiate Registers*

**registers\_unit** : Registers port map (clk, temp\_rs1, temp\_rs2, temp\_rd, temp\_rd\_write\_data, temp\_rd\_write\_enable, temp\_rs1\_data, temp\_rs2\_data);

*-- Instantiate ALU*

**alu\_unit** : ALU port map (temp\_alu\_op, temp\_alu\_a, temp\_alu\_b, temp\_alu\_result, temp\_alu\_zero);

*-- Instantiate Data Memory*

**data\_memory** : DataMemory port map (clk, temp\_mem\_address, temp\_mem\_write\_data, temp\_mem\_write\_enable, temp\_mem\_read\_data, temp\_mem\_read\_enable);

Procesul care rulează logica procesorului se poate împarți în următoarele stagii:

-aducerea instrucțiunii curente din memoria instructiuniilor

-decodarea instrucțiunii

-execuția instrucțiunii

-salvarea datelor

-setarea noului program counter

Inițial, procesul se afla în stadiu de așteptare până când semnalul clk va fi pe un front urcător. Când condiția este îndeplinită semnalul pc va primi valoarea semnalului temp\_pc care este inițializat cu 0 reprezentând prima adresa din memoria cu instrucțiuni. Procesul este din nou în stadiu de așteptare până componenta memoriei cu instrucțiuni rezolvă de efectuat procesul de a aduce din memoria ei instrucțiunea aflată la adresa dată de pc. De asemenea în acest stagiu va fi setat pe ‚0’ logic semnalul de rd\_write\_enable pentru a opri scrierea dateleor registrilor în memorie.

    wait on clk until **rising\_edge**(clk);

    pc <= temp\_pc;

    rd\_write\_enable <= '0';

    temp\_rd\_write\_enable <= '0';

wait on temp\_instruction;

Următoarea etapă este decodarea instuctiunii din semnalul temp\_instruction pentru a determina ce tip de instrucțiune este și ce ramură trebuie executat mai departe. În semnalul temp\_opcode va fi salvat tipul instrucțiunii care indică pe ce ramură a structurii if va continua procesul. Înainte de a intra pe o ramură a if-ului, toate instructiuniile au o parte comună care se poate decoda înainte de a continua pe ramurile if-ului. Această parte constă în aflarea primului registru sursă notat cu rs1, aflarea registrului destinație notat cu rd și aflarea valorii funct3 care este o extensie pentru opcode. Procesul va aștepta până la următorul front crescător al semnalului clk pentru a salva atribuirile de valori. Pentru decodarea instructiuniilor să folosit logica din figura 4.10.

Figura 4.10 –A blue and white rectangular box with white text

Description automatically generated Tabel decodare instructiuni (sursa: [4] figura 2.5)

instruction <= temp\_instruction;

        opcode <= temp\_instruction(6 downto 0);

        temp\_opcode <= temp\_instruction(6 downto 0);

        rs1 <= temp\_instruction(19 downto 15);

        temp\_rs1 <= temp\_instruction(19 downto 15);

        funct3 <= temp\_instruction(14 downto 12);

        temp\_funct3 <= temp\_instruction(14 downto 12);

        rd <= temp\_instruction(11 downto 7);

        temp\_rd <= temp\_instruction(11 downto 7);

        wait until **rising\_edge**(clk);

În funcție de valoarea semnalului opcode procesul va intra pe ramura asociată cu tipul intructiunii unde are loc a doua parte din decodarea instrucțiunii pentru a afla valoarea celui de-al doilea operand. Pentru instructiuniile de tip R valoarea celui de-al doilea operand vor fi datele din al doilea registru sursă, iar pentru instructiuniile de tip I valoarea va fi o constantă sau o adresă.

În execuția instrucțiunilor, indiferent de tipul acestora vor folosi o structură de switch pentru a alege ce tip de operație se va executa. Tipul operației este dat de temp\_funct pentru intructiuniile de tip R, respectiv teemp\_funct3 pentru instructiuniile de tip I. După executarea operației valoarea rezultată va fi salvată în registrul destinație rd și pc incrementat cu 1 pentru a trece la următoarea instrucțiune din cod. Procesul va fi în stare de așteptare până semnalul temp\_pc va fi modificat, această așteptare blochează programul din a rula tot procesul de la început cu aceeași instrucțiune.

if temp\_opcode = "0110011" then *-- R-type instruction*

*-- decode the remaining fields of instruction*

                funct7 <= temp\_instruction(31 downto 25);

                temp\_funct7 <= temp\_instruction(31 downto 25);

                rs2 <= temp\_instruction(24 downto 20);

                temp\_rs2 <= temp\_instruction(24 downto 20);

                wait on temp\_rs2\_data;

                alu\_a <= temp\_rs1\_data;

                temp\_alu\_a <= temp\_rs1\_data;

                rs1\_data <= temp\_rs1\_data;

*-- second input for ALU is data from rs2*

                rs2\_data <= temp\_rs2\_data;

                alu\_b <= temp\_rs2\_data;

                temp\_alu\_b <= temp\_rs2\_data;

*-- concatenate funct3 with funct7 for case structure*

                funct <= temp\_funct3 & temp\_funct7;

                temp\_funct <= temp\_funct3 & temp\_funct7;

                wait on temp\_funct, temp\_alu\_b;

                case temp\_funct is

                    when "0000000000" => alu\_op <= "000"; temp\_alu\_op <= "000";   *-- add*

                    when "0000100000" => alu\_op <= "001"; temp\_alu\_op <= "001";  *-- sub*

                    when others => alu\_op <= "UUU";

                end case;

                wait on temp\_alu\_result;

                alu\_result <= temp\_alu\_result;

                alu\_zero <= temp\_alu\_zero;

                rd\_write\_enable <= '1';

                temp\_rd\_write\_enable <= '1';

                rd\_write\_data <= temp\_alu\_result;

                temp\_rd\_write\_data <= temp\_alu\_result;

                wait on temp\_rd\_write\_data;

*--set next pc*

                temp\_pc <= std\_logic\_vector(unsigned(temp\_pc) + 1);

        elsif temp\_opcode = "0010011" then *-- I-type instruction for algorithmic opperations*

*-- decode the remaining fields of instruction*

                temp\_alu\_b <= X"00000" & temp\_instruction(31 downto 20); *-- second input for ALU is a constant*

                alu\_b <= X"00000" & temp\_instruction(31 downto 20);

                alu\_a <= temp\_rs1\_data;

                temp\_alu\_a <= temp\_rs1\_data;

                rs1\_data <= temp\_rs1\_data;

                case temp\_funct3 is

                    when "000" => alu\_op <= "000"; temp\_alu\_op <= "000";

                    when others => alu\_op <= "UUU"; temp\_alu\_op <= "UUU";

                end case;

                wait on temp\_alu\_result;

                alu\_result <= temp\_alu\_result;

                alu\_zero <= temp\_alu\_zero;

                rd\_write\_enable <= '1';

                temp\_rd\_write\_enable <= '1';

                rd\_write\_data <= temp\_alu\_result;

                temp\_rd\_write\_data <= temp\_alu\_result;

                wait on temp\_rd\_write\_data;

*--set next pc*

                temp\_pc <= std\_logic\_vector(unsigned(temp\_pc) + 1);

        end if;

        wait on temp\_pc;

**TESTBENCH**

La fel ca arhitectura, codul de testare este mult mai mare decât a celorlalte componenet, de aceea voi evidenția doar partea importantă din test. Toatea semnalele din entitatea procesorului vor avea un echivalent pentru testare care este compus din numele original al semnalului și prefixul „tb\_”.

tb\_clk <= not(tb\_clk) after 10ns;

    tb\_instructions\_memory(0) <= X"00600113"; *--addi x2, x0, 6*

    tb\_instructions\_memory(1) <= X"00400193"; *--addi x3, x0, 4*

    tb\_instructions\_memory(2) <= X"00218233"; *--add x4, x2, x3*

**uut** : RISC\_V\_Processor port map(

    tb\_clk,

*-- instruction*

    tb\_instructions\_memory, tb\_pc, tb\_instruction,

*-- registers*

    tb\_rs1, tb\_rs2, tb\_rd, tb\_rd\_write\_data, tb\_rd\_write\_enable, tb\_rs1\_data, tb\_rs2\_data,

*-- ALU*

    tb\_alu\_op, tb\_alu\_a, tb\_alu\_b, tb\_alu\_result, tb\_alu\_zero,

*-- mem*

    tb\_mem\_address, tb\_mem\_write\_data, tb\_mem\_write\_enable, tb\_mem\_read\_data, tb\_mem\_read\_enable,

*-- extra decode*

    tb\_funct7, tb\_funct3, tb\_opcode, tb\_funct

    );

Acest test va genera următoarea formă de unde:

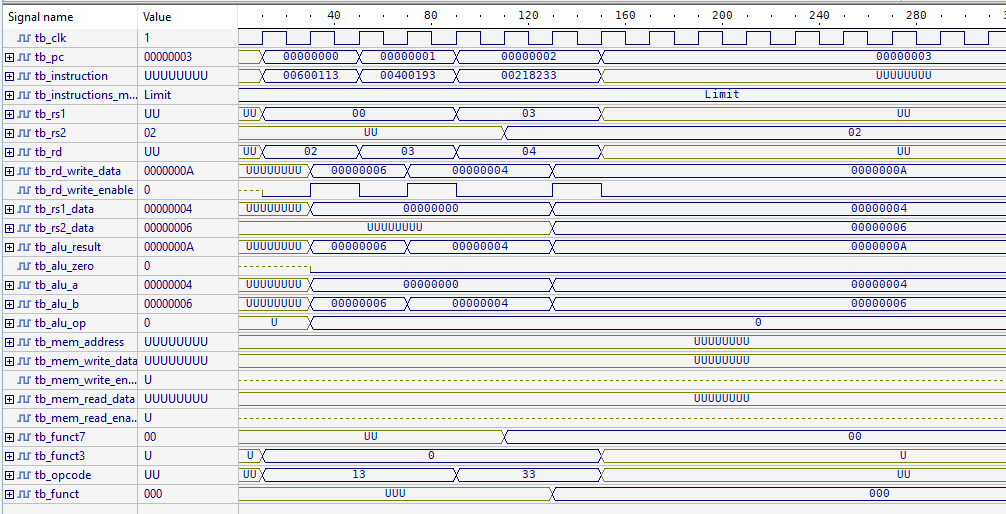


Figura 4.11 – Formă de unde generate in urma rulării testului pentru procesor

Forma de unde pentru procesor, după cum se poate observa este mai stufoasă comparativ cu formele de unde individuale pe componente. Semnalele pentru memoria de date, tb\_mem\_address, tb\_mem\_write\_data, tb\_mem\_write\_enable, tb\_mem\_read\_data și tb\_mem\_read\_enable sunt nedefinite din cauză că în timpul testului nu rulează instrucțiuni de salvare în memoria de date. În test sunt executate 3 instrucțiuni, de aceea de la valoarea lui tb\_pc egală cu 3 ar însemnă aducerea din memorie a 4a instrucțiune dar oferă răspunsul nedefinit deoarece a 4a instrucțiune lipsește. Cea mai importantă parte din toate semnalele este urmărirea semnalelor alocate unității aritmetico-logice, adică semnalele cu prefixul „tb\_alu\_”.

Pentru prima instrucțiune, la momentul de 30ns, operandul a primește valoarea aflata în registrul x0 adică 0 deoarece registrul x0 are mereu valoarea 0 și nu poat fi modificată. În schimb, din cauză că operația executată este o adunare imediată, operandul b va primi valoarea 6 care este aflata din decodarea instrucțiunii, rezultatul final fiind 6 (0+6=6). Rezultatul este salvat în registrul x2, acest registru este selectat prin semnalul tb\_rd și salvarea este realizată când semnalul tb\_rd\_write\_enable are valoarea ‚1’ logic. A doua instrucțiune este similară cu prima cu observația că al doilea operand primește valoarea constantă 4 decodată din instrucțiune. Astfel rezultatul final va fi 4 urmând a fi salvat în registrul x3 când semnalul tb\_rd\_write\_enable are valoarea ‚1’ logic. Din cauză că primele 2 instrucțiuni au fost de adunare directă cu constantă, semnalul tb\_rs2\_data a rămas nedefinit deoarece nu se folosește registrul sursa 2 în execuția instructiuniilor. A treia instrucțiune începe de la 90ns și efectuează operația de adunare intre registrii x2 și x3, rezultatul fiind salvat în registrul x4. Din cauză că răspunsul de la unitatea aritmetică-logică este aproape instant se pot urmări ușor operațiile pe forma de unde. Această instrucțiune fiind ultima nu va mai schimba valorile semnalelor pentru unitatea aritmetică-logică. Semnalul alu\_a are valoarea 4, semnalul alu\_b are valoarea 6, iar rezultatul urmat din adunarea acestora este A care convertit în baza 10 este 10.

**5. MANUAL DE UTILIZARE**

Pentru a rula și testa funcționarea procesorului este necesară instalarea unui mediu de programare care suportă limbajul de descriere VHDL. Proiectul a fost realizat folosind produsul de la Aldec numit Active-HDL. După descărcarea și instalarea programului, la deschiderea acestuia va arata o pagină goală deoarece nu este deschis un spațiu de lucru.

Deschiderea spațiului de lucru se face selectând „file” din bara ferestrei, iar din meniul deschis se va selecta „Open workspace..”.

A screenshot of a computer

Description automatically generated

Figura 5.1 – Locație opțiune meniu „Open Workspace..”

A screenshot of a computer

Description automatically generated După apăsarea pe „Open workspace..”, va deschide o fereastră în care trebuie să căutați fișierul .aws din folderul proiectului, după selectarea fișierului apăsați pe butonul de „Open” al ferestrei.

Figura 5.2 – Exemplu locatie fisier cu extensia .aws

Pentru a testa procesorul sau componentele individuale, în panoul din partea stangă, apăsați pe „+” pentru a extinde lista de entități incluse în fișierul .vhd, faceți click dreapta pe entitatea cu numele „tb\_arch” și din meniu selectați „Set as Top-Level”.

A screenshot of a computer

Description automatically generated

Figura 5.3 – Locatie optiune meniu „Set as Top-Level”

Pentru a putea genera formele de undă, va trebuie deschis un fișier waveform prin „file-> new -> waveform”.

A screenshot of a computer

Description automatically generated

Figura 5.3 – Locație opțiune creare fișier „Waveform”

A screenshot of a computer

Description automatically generated După deschiderea fișierului untitled.awc, se va apasă butonul de „Initialize simulation” sau tasta F6 care va arată o fereastră în care se cere compilarea proiectului, în această fereastră apăsați butonul „Yes”.

Figura 5.4 – Locație buton „Initialize Simulation (F6)”

A screenshot of a computer

Description automatically generated În acest moment simularea a fost initializata și în căsuța din stânga pe tab-ul structure se vor afla toate semnalele din entitatea aleasă. Din această căsuță puteți trage cu mouse-ul, semnalele pe care doriți să le simulați, pe fișierul untitled.awc.

Figura 5.5 – Locație fereastra cu semnale

A screenshot of a computer

Description automatically generatedDupă alegerea tuturor semnalelor dorite, în dreapta butonul de inițializare se va seta în căsuța alăturată pentru cât timp va fi executată simulare. După selectarea timpului se va apăsa butonul de „run for”

Figura 5.6 – Locație buton „Run For (F5)”

Rularea va genera formele de unde cu semnalele alese astfel:

A screenshot of a computer

Description automatically generated

Figura 5.7 – Exemplu forma de unde

**6. CONCLUZIE**

În cadrul acestui proiect, mi-am propus să implementez procesorul RISC-V pentru a dobândi cunoștințe de programare hardware, fiind unul dintre motivele principale care m-au condus la alegerea acestei facultăți.

Am reușit să construiesc bazele procesorului RISC-V cu un set redus de instrucțiuni. Procesorul este proiectat să suporte execuția a mai multor instrucțiuni consecutive. Pe bazele create se pot adaugă cu ușurință mai multe instrucțiuni din setul de baza a procesorului sau customizate.

Chiar dacă proiectul implementează versiunea de bază a procesorului, au existat unele idei pe care nu am reușit să le implementez. Una din ideile pe care nu am reușit să le implementez este adăugarea unei memorii cache între memoria de date și procesor pentru a micșora timpul de execuție a instrucțiunilor care necesită accesarea memoriei de date.

Prin acest proiect, am reușit să dobândesc cunoștințe de bază despre cum funcționează procesoarele și cum se dezvoltă programe folosind VHDL. Aceste cunoștințe mă vor ajuta pentru viitor în a dezvolta aplicații pentru chipuri FPGA în situații în care condițiile de optimizare a unei aplicații sunt mai stricte și microprocesoarele nu pot îndeplini acele condiții. Un exemplu este întâlnit la matricele mari de leduri care controlează fiecare led în paralel, de aceea un sistem FPGA este mai optim din cauza paralelismului oferit de VHDL

În concluzie, proiectul a îndeplinit în mare parte obiectivele propuse. Această implementare oferă o bază solidă pentru dezvoltarea ulterioară pentru a fi îmbunătățită din punct de vedere a timpului de execuție și pentru adăugarea de noi instrucțiuni.

**BIBLIOGRAFIE**

[1] “RISC-V,” Wikipedia, Jan. 13, 2021. \*\*\*https://en.wikipedia.org/wiki/RISC-V

[2] Wikipedia Contributors, “Assembly language,” Wikipedia, Sep. 25, 2019. \*\*\*https://en.wikipedia.org/wiki/Assembly\_language

[3] J. Bartlett, “Programming from the Ground Up .”

[4] D. A. Patterson and J. L. Hennessy, Computer Organization and Design RISC-V Edition: The Hardware Software Interface .

[5] “VHDL,” Wikipedia, Aug. 13, 2020. \*\*\*https://en.wikipedia.org/wiki/VHDL

[6] B. Mealy and F. Tappero, Free Range VHDL.

[7] D. L. Perry, VHDL: Programming by Example.

