

**Catedra Calculatoare**

**Tema Proiectului:**

**Implementarea procesorului EMMA-2**

**Student:** Cățănaș-Kelemen Kaj-Victor  
 (Gr.30234, anul III)  
  
**Profesor îndrumător**: Mocan Cristi

*Data:**03.01.2021*

**Cuprins**

1. Rezumat 4

2. Introducere 5

2.1 Context 5

2.2 Obiectivele proiectului 6

Arhitectura RISC 7

2.3 Sumar 7

3. Fundamentare teoretica 7

3.1 Mediul Vivado Design Suite 7

3.2 Placa FPGA Nexys4 DDR 7

3.3 Procesorul EMMA-2 8

3.3.1 Arhitectura RISC 8

3.3.2 Comparare intre familiile EMMA si MIPS 8

3.4 Arhitectura Harvard 9

3.4.1 Comparare intre arhitectura Harvard si von Neuman 9

3.4.2 Schema arhitectura Harvard 9

3.5 Doua faze ale clock-ului 10

3.6 Descrierea componentelor procesorului EMMA-2 10

3.6.1 Program Counter 10

3.6.2 Unitatea de microcod 10

3.6.3 Unitatea de microcod program counter 11

3.6.4 Magistralele procesorului EMMA-2 11

3.6.5 Memoria de date 11

3.6.6 Unitatea de registrii 12

3.6.7 Unitatea Aritmetico-Logica 12

3.6.8 Generatorul de doua faze ale clock-ului 12

3.6.9 Schema generala a procesorului EMMA-2 12

3.6.10\_Memoria\_de\_microcod 13

3.6.11 Setul de instructiuni 14

4. Proiectare si Implementare 16

4.1 Schema Bloc detaliata =>anexă 16

4.2 Descrierea componentelor 16

4.2.1 Generator de clock in doua faze 16

4.2.2 Microcode Program Counter 17

4.2.3\_Unitatea\_aritmetico-logica 17

4.2.4\_Registers 17

4.2.5 Program Counter 17

4.2.6 Instruction Memory 18

4.2.7 Data memory 18

4.2.8\_Unitatea\_microcod 18

5. Rezultate experimentale 19

5.1 Detalii de implementare 19

5.2 Teste efectuate 19

5.2.1 Teste generator de doua faze de ciclu de ceas 19

5.2.2 Teste MPC 19

5.2.3 Teste Registrii 19

5.2.4 Teste ALU 20

5.2.5 Teste Program Counter 20

5.2.6 Teste Instruction Memory 20

5.2.7 Teste Data Memory 20

5.2.8 Teste unitate de microcod 20

5.2.9 Teste procesor 20

6. Concluzii 20

Bibliografie 20

Anexă 21

# 

# 1. Rezumat

Proiectul a presupus realizarea procesorului EMMA, care implementeaza operatii de tip load/store si operatii logice sau aritmetice intre registre. Acesta utilizeaza arhitectura Harvard, cu memorii separate pentru instructiuni si pentru date. Procesorul are un set de instructiuni predefinite dar versiunea descarcabila a modelului nu contine resursele din microcodul necesar pentru implementare. Exista 2 versiuni ale acestui procesor (EMMA1 si EMMA2). EMMA1 este procesorul de baza si poate executa doar operatii care necesita un singur ciclu de ceas, EMMA2 in schimb permite si operatiile de inmultire, impartire.

# 

# 2. Introducere

## 2.1 Context

Obiectivul initial care a condus la dezvoltarea multicomputerului EMMA (Elaborate Multi-Mini Associativo), a fost mecanizarea sortarii corespondentei. Nevoia unui sistem care ar putea sta la baza unei game mai largi de aplicatii a determinat studiul și dezvoltarea noului sistem, EMMA2. Procesorul este format dintr-un Program Counter(PC), 2 memorii (Instruction Memory si Data Memory), un Microcode Program Counter(MPC), un ALU, si 3 registrii (2 pentru operatorii ALU si 1 pentru stocarea rezultatului) si unitatea principala care poate fi considerata “creierul procesorului”, Unitatea Microcod.

## 2.2 Obiectivele proiectului

Obiectivul principal a fost de a implementa procesorul EMMA2 in mediul de proiectare **Xilinx Vivado Design Suite**.Fiecare componenta trebuie implementata si testata separat inainte de a fi conectata la ansamblul principal.

Prin urmare, acesta presupune descrierea arhitecturii procesorului cu ajutorul limbajului de descriere hardware **VHDL** sau **VHSIC HDL** (Very High-Speed Integrated Circuit Hardware Description Language). Este un limbaj de descriere hardware, destinat descrierii comportamentului și arhitecturii unui modul electronic logic, ale unei functiuni logice combinatorii sau secventiale. Alaturi de limbajul Verilog, este cel mai utilizat limbaj de proiectare a sistemelor electronice digitale. E una din uneltele principale pentru proiectarea circuitelor integrate moderne, aplicat cu succes in campul microprocesoarelor (DSP, acceleratoare grafice), în telecomunicatii (TV, celulare), automobile (navigație, sisteme de control al stabilitații) și altele.

Procesoarele microprogramate, spre deosebire de microprocesoarele clasice, cu structura fixa și setul de instrucțiuni impus de fabricant, permit la o viteza relativ mai redusa, obținerea unor structuri mult mai flexibile, in ceea ce priveste realizarea și modificarea setului de instructiuni, de catre utilizator, adaptate aplicației abordate.

Procesorul presupune sa fie format dintr-un Program Counter(PC), 2 memorii (Instruction Memory si Data Memory), un Microcode Program Counter(MPC), un ALU, si 3 registrii (2 pentru operatorii ALU si 1 pentru stocarea rezultatului) si Unitatea Microcode.

Unitatea Microcode(microprogramata) contine memoria de microcoduri care este adresată de un camp numit function al registrului de instrucțiuni in ciclul de ceas respective, in care este receptionata o noua instructiune din memoria de instructiuni si de contorul de programe de microcod (MPC) in ciclurile de ceas ulterioare.Fiecare adresa din memoria de microcod este un “microcode word” (un cuvant) care va fi format dintr-un sir de biti de control care vor preciza setarile celorlalte componente in functie de rolul lor pentru instructiunea curenta.

Memoria microcodurilor este divizata in mod logic in doua sectiuni, sectiunea de Microcod standard care cuprinde adresele 0-127 si secțiunea alternativa de microcod cu adresele 128-255. Sectiunea de Microcod standard permite implementarea operațiunilor aritmetice cu ciclu unic, precum adaugare, scadere, dar si operatii logice.In timp ce secțiunea de Microcod alternativ permite implementarea operațiunilor cu mai multe cicluri, cum ar fi inmultirea si divizarea.

Microprocesoarele ce utilizeaza arhitectura RISC (Recuced Instruction Set Computing)au instructiuni limitate incorporate. Dispun de o viteza reala mai mare la aceeasi viteza de tact (1 instructiune/tact) deci ne putem permite viteze mai mici, prin urmare putem ajunge la un consum de energie mai mic.

## 2.3 Sumar

In continuare vom face o scurta prezentare asupra urmatoarelor capitole ale documentatiei:

* Fundamentarea teoretica va contine descrierea fiecarei componente in parte si functionalitatea acestora, perifericele, mediul de lucru si solutia propusa prin care voi putea dezvolta aplicatia
* Proiectarea si implementarea va contine abordarea personala si deciziile alese prin care am decis sa proiectez aplicatia si sa implementez solutia gandita
* Testarea va contine testele realizate pe parcursul dezvoltarii acestui proiect si implementare concreta cu rezultatele obtinute
* La sfarsit vor aparea concluzii, idei viitoare si bibliografia/resursele de pana acum

# 3. Fundamentare teoretica

## 3.1 Mediul Vivado Design Suite

Vivado Design Suite este o aplicatie software produsa de Xilinx pentru sinteza și analiza proiectelor VHDL, inlocuind Xilinx ISE cu functii suplimentare pentru dezvoltarea unui sistem pe cip și sinteza la nivel inalt. Vivado reprezinta o rescriere fundamentala si o gandire completa a intregului flux de proiectare si dezvoltare pentru o aplicatie dorita.

## 3.2 Placa FPGA Nexys4 DDR

Implementarea acestui proiect se va realiza dupa procesul de proiectare si va include descrierea fiecarei componente in intermediul limbajului de descriere hardware: VHDL; legarea tuturor unitatiilor, simularea acestora si testarea produsului final pe placuta de dezvoltare Nexys4.

Placa Nexys 4 este o platforma completa de dezvoltare a circuitelor digitale, gata de utilizare, bazata pe cea mai recentă Artix-7 ™ Field Programmable Gate Array (FPGA) de la Xilinx. Cu FPGA mare, de mare capacitate (numarul piesei Xilinx XC7A100T-1CSG324C), memorii externe generoase si colectia de porturi USB, Ethernet si alte porturi, Nexys 4 poate gazdui proiecte variind de la circuite combinatorii introductive pana la procesoare incorporate puternice. Mai multe periferice incorporate, inclusiv un accelerometru, un senzor de temperatura, un microfon digital MEMs, un amplificator de difuzoare și o mulțime de dispozitive I/O permit Nexys 4 sa fie utilizat pentru o gama largă de modele fara a avea nevoie de alte componente.

## 3.3 Procesorul EMMA-2

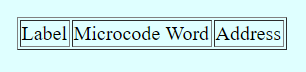
### 3.3.1 Arhitectura RISC

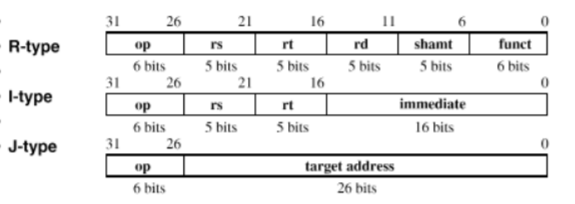
Microprocesoarele ce utilizeaza arhitectura RISC au instructiuni limitate incorporate.

De aceea este nevoie de mai putine tranzistoare pentru crearea microprocesorului. Reducerea instructiunilor salveaza si spatiu in microprocesor. Acest tip de procesor este preferat pentru scopuri stiintifice unde numarul limitat de instructiuni este necesar.

Arhitectura RISC înseamnă viteză reală mai mare la aceeași viteză de tact ( pentru că execută o instrucțiune la fiecare tact) deci ne putem permite viteze mai mici, deci consum de energie mai mic și căldură mai puțină produsă .

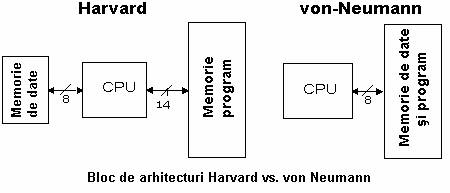
### 3.3.2 Comparare intre familiile EMMA si MIPS





Formatul instrunctiunilor principale din memoria de instrunctiuni va fi asemanator in ambele cazuri divizate pe 3 categorii in functie instrunctiunea specificat.Tipul R va fi folosit pentru pentru operații aritmetice / logice in care vom dispune de 3 adrese de registrii. Tipul I implica operații aritmetice/logice cu o valoare “Immediate” specifica, lucru cu memoria, dar și ramificari conditionate.Iar Tipul J va fi folosit pentru salturi neconditionate. Spre deosebire de aceasta utilizare procesorul Emma2 datorita utilizarii unei arhitecturi microprogramate care foloseste o memorie de microcod, va avea si un alt format instrunctiune care va fi aplicat fiecarui cuvant de microcod. Acesta va contine campul Label care poate fi tratat ca si un sir de caractere ce va indica mai usor tipul operatiei pentru care este utilizata instructiunea.Microcode-word va contine sirul de 32 de biti care va contine toate semnalele de control generate pentru operatia respective si campul Address va contine adresa urmatoare la care va fi trimis contorul MPC in cadrul memoriei de microcod.

## 3.4 Arhitectura Harvard



### 3.4.1 Comparare intre arhitectura Harvard si von Neuman

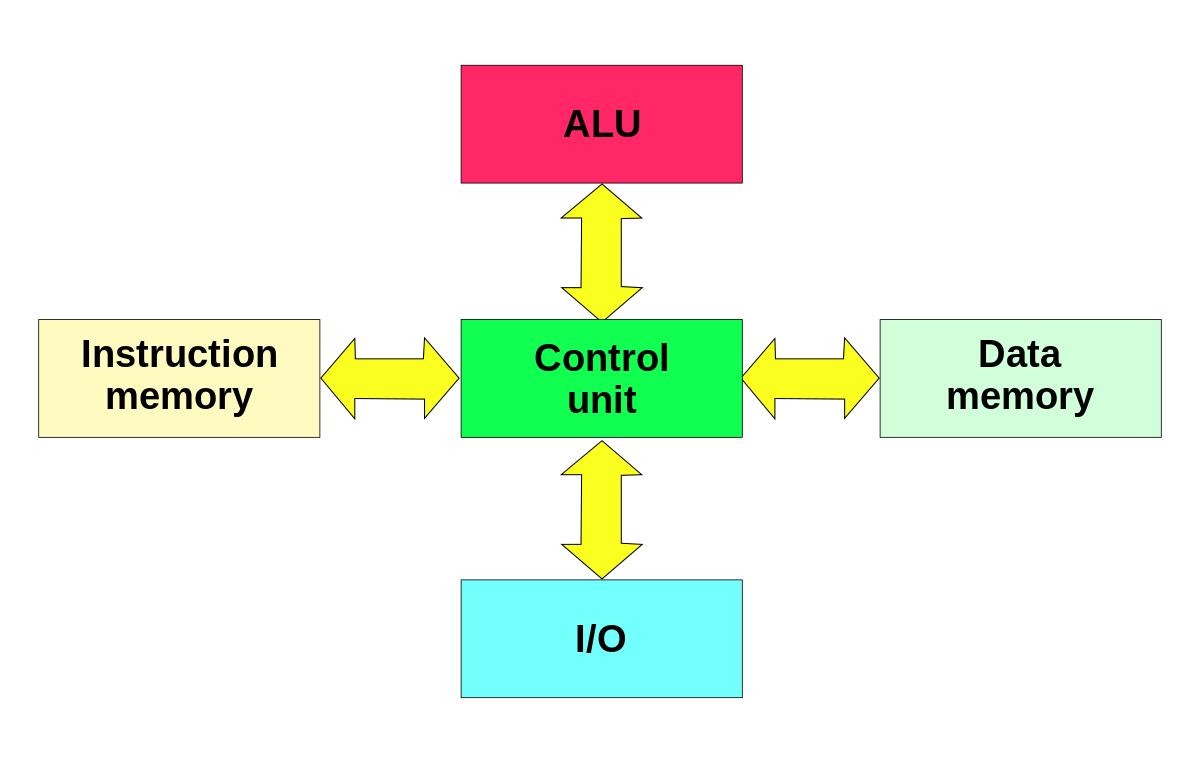
Procesorul EMMA-2 dispune de o arhitectura Harvard care este o arhitectura a calculatoarelor caracterizata prin stocarea separata a instrucțiunilor și a datelor. Numele acestei arhitecturi provine de la sistemul de calcul Harvard Mark I, care stoca instructiunile pe 24 biti pe o banda perforata, iar datele in contoare electromecanice ce permiteau 23 de cifre. Datorita lungimii cuvintelor, a tehnologiei de implementare și a structurii memoriei de adresare diferite, în cadrul acestei arhitecturi nu este necesar ca cele doua tipuri de memorie (Program si Data) sa dispună de aceleasi caracteristici. De regula memoria pentru instructiuni are o capacitate mai mare decat cea de date. De exemplu, microcontrolerele PIC au un cuvânt de date de 8 biti și o lungime a instructiunii de 12, 14, 16 sau 32 biți.

In functie de necesarul de memorie, instrucțiunile pot fi stocate de exemplu intr-o memorie de tip ROM (“read-only memory”), in timp ce datele se afla intr-o memorie de tip RAM.(“random-access memory”).Aceasta arhitectura contrasteaza cu arhitectura von Neumann, unde instructiunile și datele programului utilizeaza aceeasi memorie si aceleasi surse ca cai de date.

Spre deosebire de Von Neumann, arhitectura Harvard descrie un sistem de calcul unde memoria de date și memoria de program sunt separate. Asta inseamna ca procesoarele vor dispune de doua magistrale de date și de adrese: una pentru instrucțiuni și una pentru date.

Avantajul fata de arhitectura Von Neumann este ca aceste magistrale nu trebuie sa corespunda in dimensiune, deci putem avea un procesor pe 8 biti care adreseaza mai mult de 256 de octeti de memorie. Un alt avantaj este ca memoria de program poate fi facuta non-volatila, deci odata scris un program acesta nu mai trebuie reincarcat la fiecare pornire noua.

### 3.4.2 Schema arhitectura Harvard



## 3.5 Doua faze ale clock-ului

EMMA functioneaza pe un ceas cu doua faze. In ciclurile de ceas in care sunt active, fiecare unitate isi executa actiunile interne in prima faza a ceasului și trimite un rezultat impachetat/complet in a doua faza de tact. Ca exemplu unitatea de microcod citeste memoria de microcod in prima faza și trimite campurile de microcod corespunzatoare catre alte unitati, daca acestea sunt active (rising’edge,diferite de zero), in a doua faza.

## 3.6 Descrierea componentelor procesorului EMMA-2

### 3.6.1 Program Counter

Unitatea Program Counter dispune de o iesire care este conectata intern direct la una din propriile sale intrari. si totodata conectata extern la una din iesirile sale. De asemenea contorul de program dispune de inca 2 intrari (valoarea +1 si iesirea din BUS2) conectate intern la un multiplexor, a carui iesire formeaza cealalta intrare in sumator. Trei dintre bitii microcodului Program Counter controleaza aceste intrari; combinatiile adecvate ale acestor biti permit PC-ului sa fie incrementat cu 1 sau cu o valoare specifica in cazul salturilor conditionate.De asemenea putem obtine si combinatia specifica saltului neconditiat pentru o operatie de tip “jump”. Ultimul bit de microcod controleaza de fapt iesirea catre memoria de instructiuni; cealalta iesire este conectata inapoi la intrarea sumatorului, este activată cand valoarea din PC este actualizata.

### 

### 3.6.2 Unitatea de Microcod

Unitatea de Microcod conține memoria de microcoduri care este adresată de campul “function” al instructiunii noi preluate de catre registrul IR in ciclul de ceas in care este recepționata o noua instructiune din memoria de instructiuni și de Contorul de programe pentru Unitatea de Microcod (MPC) in ciclurile de ceas ulterioare. Unitatea de microcod are 2 campuri de control rezervate pentru microcod, unul pentru controlul iesirilor sale (BUS1, BUS2 și MPC) și unul pentru controlul executiei instructiunilor conditionate.

Unitatea de microcod dispune de 2 iesiri de date diferite, campul Address sau Literal (Immediate) al instructiunii din registrul IR, care poate fi trimis la BUS1 sau BUS2 și campul de adresa al cuvântului de microcod curent (CW) care poate fi trimis către Microprogram Counter. Bitii din al doilea camp selecteaza codul de conditie ALU (CC0 sau CC1) și determina daca microcodul destinat Program Counter-ului, Memoriilor, Registrelor și unitatii ALU este trimis, in cazul in care conditia este realizata cu success sau esec. Aceasta facilitate poate fi utilizata pentru a implementa instructiuni de salt conditionat, precum BNEG, BNEZ care sunt asemanatoare operatiei de branch.

### 3.6.3 Unitatea de Microcod Program Counter

Unitatea Microcode Program Counter are un comportament similar cu Program Counter. Acestea conțin registrul de baza impreuna cu un sumator care primește una dintre intrarile sale din registru de la propria sa iesire si dispune de inca o intare la sumator care este conectata la un multiplexor.Asadar intrarea finala de la sumator din partea multiplexorului poate fi valoarea ‘1’ sau o valoare preluata din Unitatea principala de Microcod. Microcodul este utilizat pentru a actiona ca contorul memoriei de microcod.

### 3.6.4 Magistralele procesorului EMMA-2

Magistralele (BUS 1, respectiv BUS 2) au mai multe semnale de intrare, dar vor primi date doar de la o singura intare intr-o faza de clock.Intrarile care nu primesc date de intrare sunt setate la zero.Acest aspect se poate realiza prin analizarea microcodului primit. Intrarile ar trebui conectate la o singura poarta OR care in final ne va determina valoarea finala care va fi plasata pe magistrala BUS.

### 3.6.5 Memoria de date

Memoria de Date are propriul set de registre incorporate: MAR-cu ajutorul acestuia putem accesa adresa specifica din memoria de date, este utilizat atat la citire cat si la scriere. De asemenea avem registrul MBR care este un registru buffer si ne indica valoarea concreta pe care dorim sa o scriem in memorie sau pe care vrem sa o scriem. Intrarile trimise catre magistrale sunt trimise catre toate iesirile, dar utilizate numai la unitațile de receptie daca bitul de control corespunzator al unitatii de receptie este setat la 1.Unitatea de memorie dispune de un bit de control Read/Write (0=Read, 1=Write). Pentru o operație de citire (Read / Write = 0), adresa trimisa de la BUS 2 este copiata in registrul MAR doar la urmatoarea faza de ceas, memoria va fi citita si rezultatul copiat in registrul MBR. In urmatoarea faza de ceas 1, valoarea din registrul MBR este trimisă la BUS 1. Pentru o operatie de scriere (Read / Write= 1), adresa trimisa de la BUS 2 este copiata in MAR, iar adresa trimisa de la BUS1 copiata in MBR, astfel valoarea va fi scrisa in memorie.

### 3.6.6 Unitatea de registrii

Unitatea de Registrii conține 16 registrii pentru a putea fi folositi in program. Primul registru, R0 este setat in permanenta la valoarea 0. Valorile de intrare sunt receptionate de la ALU, iar cele doua semnale de iesire sunt conectate la BUS1, respectiv BUS 2. Cand Unitatea de Microcod trimite o comanda de microcod catre unitatea de Registre, adauga sursele și destinatiile corespunzatoare extrase din instrucțiunea preluata de IR-registrul de instructiuni. Intr-o instrucțiune ADD RD RS1 RS2, valoarea din registrul RS1 este trimisa la BUS1 și cea din RS2 la BUS2.Unitatea de Registrii are o intrare (de la ALU) si de asemenea 2 iesiri, una la BUS1 și una la BUS2, fiecare controlata in parte de un bit de control din microcodul receptionat.

### 

### 3.6.7 Unitatea Aritmetico-Logica

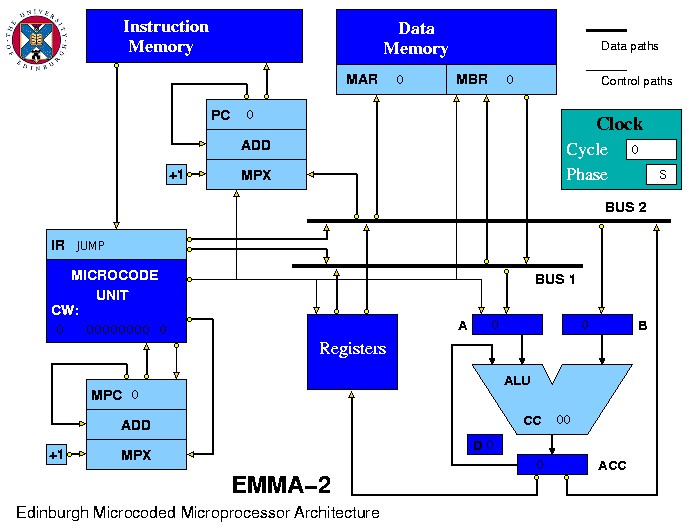
Unitatea aritmetica logica are 2 registre de intrare, A și B, care preiau date de la calea de date BUS 1, respectiv BUS 2 si un registru acumulator ACC. Aceste registre se comporta diferit fata de cele din EMMA-1 prin faptul ca nu sunt setate automat la zero daca bitul lor de microcod nu este setat la primirea unei noi comenzi. De asemenea functionarea algoritmului de divizare implica folosirea numerelor doar pe 16 biti, iar rezultatul unei operatii de inmultire nu poate depasi 32 de biti, ALU isi verifica valorile de intrare și oprește simularea daca acestea sunt sn afara intervalului specificat. Unitatea aritmetica logica are 2 campuri de control pentru microcod, unul pentru selectarea operatiei alese si unul pentru controlul intrarilor sale (BUS1 si BUS2) si iesirile sale (Registers si BUS2).

### 3.6.8 Generatorul de doua faze ale clock-ului

Dupa cum am mentionat anterior, EMMA functioneaza pe un ceas cu doua faze. In ciclurile de ceas in care sunt active, fiecare unitate isi executa actiunile interne in prima faza a ceasului și trimite un rezultat impachetat/complet in a doua faza de tact. Din acest motiv, la sectiunea de simulare vom implementa un generator de 2 faze ale clock-ului prin crearea unui process in care vom alterna valorile tact-ului intre valoarea logica ‘0’ si ‘1’ cu un timp minim de asteptare intre schimbari pentru a crea un sistem eficient si corespunzator aplicatiei la prelucrarea instructiunilor.

### 

### 3.6.9 Schema generala a procesorului EMMA-2



### 

### **3.6.10 Format de microcod**

## 

## 3.6.11 Setul de instructiuni

**JUMP Literal**

Jump primeste un operator imediat si il incarca in PC ca o adresa de instructiuni. Valoarea din PC e apoi trimisa catre memoria de instructiuni. Microcodul pentru aceasta intructiune comprima un singur cuvant si astfel nu e necesara o intrare in tabelul de jump. Este preincarcat in aplicatie la locatia de memorie 0 al microcodului.

**JREG RS**

JREG foloseste un singur registru sursa(RS), ale carui continut va fi citit si incarcat in PC. Valoarea in PC va fi apoi trimisa memoriei de instructiuni.

**BEQZ Literal**

Literalul BEQZ ia un operand literal; daca CC0=0 va fi adaugat la PC, iar daca CC0=1, PC va fi incrementat(cu +1). Valoarea din PC va fi apoi trimisa catre memoria de instructiuni.

**BNEG Literal**

BNEG ia un operand literal, care, daca CC=1, va fi adaugat lui PC, iar daca CC=0, se va incrementa doar PC(cu +1). Valoarea din PC va fi trimisa mai departe catre memoria de instructiuni.

**LD RD Address**

LD trimite campul de adresa al instructiunii catre MAR, acceseaza memoria de date, si incarca valoarea din MBR in registrul destinatie(RD), prin ALU. PC e incrementat si noua valoare PC e trimisa memoriei de instructiuni. Intrarea in tabela de salt si microcodul pentru aceasta instructiune sunt preincarcate in aplicatie.

**LDL RD Literal**

LDL este necesar pentru a incarca valoarea din campul literalului in registrul destinatie(RD), prin ALU. PC trebuie incrementat, iar valoarea incrementata este trimisa catre memoria de instructiuni.

**LDX RD Address(RS)**

LDX este necesar pentru a utiliza ALU ca sa adaugam valoarea campului Address valorii stocate in registrul sursa(RS), pentru a trimite rezultatul catre MAR pentru a accesa memoria de date si pentru incaarcarea valorii din MBR in registrul destinatie(RD), prin ALU. PC va fi incrementat iar noua valoare va fi trimisa memoriei de instructiuni.

**ST Address RS**

ST este necesar pentru a citi valoarea din RS si a o trimite catre MBR in acelasi timp in care campul Address al instructiunii e trimis catre MAR si scris in memoria de date. PC e incrementat si noua valoare e trimisa catre memoria de instructiuni.

**STX Address(RS2) RS1**

STX este necesar pentru a folosi ALU cu scopul de a adauga valoarea campului Address valorii stocate in registrul sursa RS2, apoi trimite aceasta adresa catre MAR in acelasi timp in care valoarea din registrul sursa RS1 e trimisa catre MBR si scrisa in memoria de date; PC este incrementat si trimis catre memoria de instructiuni.

**ADD RD RS1 RS2**

ADD citeste valorile RS1 si RS2(chiar simultan), le trimite catre ALU impreuna cu codul functiei ALU(=0 pentru ADD) apoi scrie rezultatul in RD; PC e incrementat si trimis catre memoria de instructiuni.

**SUB, AND, OR si XOR**

SUB, AND, OR si XOR functioneaza intr-un mod asemanator, folosind codul specific al fiecarei functii ALU(1, 2, 3 si respectiv 4).

**ADDL RD RS Literal**

ADDL citeste valoarea din RS, o trimite catre ALU impreuna cu valoarea literalului si codul functiei ALU(=0 pentru ADD) si apoi scrie rezultatul in RD; PC este incrementat si trimis catre memoria de instructiuni.

**SUBL, ANDL, ORL si XORL**

SUBL, ANDL, ORL si XORL functioneaza intr-un mod asemanator, folosind codul specific al fiecarei functii ALU(1, 2, 3 si respectiv 4).

# 4. Proiectare si Implementare

Proiectul a fost realizat in mediul de proiectare si dezvoltare numit Vivado(versiunea 2020.1)

Schema bloc a procesorului a fost luata de pe pagina web a procesorului EMMA2 de la

Universitatea Edinburgh.

## 4.1 Schema Bloc detaliata =>[anexă](#anexa)

## 4.2 Descrierea componentelor

In continuare vom prezenta componentele fundamentale ce tin de arhitectura procesorului EMMA-2, functionarea acestora si rolul lor in functionarea corecta a unitatii principale.

### 

### 4.2.1 Generator de clock in doua faze

Fiecare unitate care este activata prin semnalul Enable sau semnalele corespunzatoare provenite de la instructiunea microcod isi va executa actiunile interne in prima faza a ceasului și trimite un rezultat impachetat/complet in a doua faza de tact. Din acest motiv, la sectiunea de simulare vom implementa un process destinate unui generator de 2 faze ale clock-ului prin alternanta valorilor de tact-ului intre valoarile logice ‘0’ si ‘1’ cu un timp minim de asteptare intre schimbari pentru a asigura un system eficient si sincronizare corecta intre instructiuni.

### 

### 4.2.2 Microcode Program Counter

Unitatea de Microcod Program Counter conține memoria de microcoduri care este adresată de campul “function” al instructiunii noi preluate de catre registrul IR in ciclul de ceas in care este recepționata o noua instructiune din memoria de instructiuni și de Contorul de programe pentru Unitatea de Microcod (MPC) in ciclurile de ceas ulterioare. Unitatea de microcod are 2 campuri de control rezervate pentru microcod, unul pentru controlul iesirilor sale (BUS1, BUS2 și MPC) și unul pentru controlul executiei instructiunilor conditionate.

Unitatea de microcod dispune de 2 iesiri de date diferite, campul Address sau Literal (Immediate) al instructiunii din registrul IR, care poate fi trimis la BUS1 sau BUS2 și campul de adresa al cuvântului de microcod curent (CW) care poate fi trimis către Microprogram Counter.

.

### 4.2.3\_Unitatea\_aritmetico-logica

Unitatea aritmetica logica are 2 registre de intrare, A și B, care preiau date de la calea de date BUS 1, respectiv BUS 2 si un registru acumulator ACC. Aceste registre se comporta diferit fata de cele din EMMA-1 prin faptul ca nu sunt setate automat la zero daca bitul lor de microcod nu este setat la primirea unei noi comenzi. De asemenea functionarea algoritmului de divizare implica folosirea numerelor doar pe 16 biti, iar rezultatul unei operatii de inmultire nu poate depasi 32 de biti, ALU isi verifica valorile de intrare și oprește simularea daca acestea sunt sn afara intervalului specificat. Unitatea aritmetica logica are 2 campuri de control pentru microcod, unul pentru selectarea operatiei alese si unul pentru controlul intrarilor sale (BUS1 si BUS2) si iesirile sale (Registers si BUS2).

### 4.2.4\_Registers

Unitatea de Registrii conține 16 registrii pentru a putea fi folositi in program. Primul registru, R0 este setat in permanenta la valoarea 0. Valorile de intrare sunt receptionate de la ALU, iar cele doua semnale de iesire sunt conectate la BUS1, respectiv BUS 2. Cand Unitatea de Microcod trimite o comanda de microcod catre unitatea de Registre, adauga sursele și destinatiile corespunzatoare extrase din instrucțiunea preluata de IR-registrul de instructiuni. Intr-o instrucțiune ADD RD RS1 RS2, valoarea din registrul RS1 este trimisa la BUS1 și cea din RS2 la BUS2.Unitatea de Registrii are o intrare (de la ALU) si de asemenea 2 iesiri, una la BUS1 și una la BUS2, fiecare controlata in parte de un bit de control din microcodul receptionat.

### 4.2.5 Program Counter

Unitatea Program Counter dispune de o iesire care este conectata intern direct la una din propriile sale intrari. si totodata conectata extern la una din iesirile sale. De asemenea contorul de program dispune de inca 2 intrari (valoarea +1 si iesirea din BUS2) conectate intern la un multiplexor, a carui iesire formeaza cealalta intrare in sumator. Trei dintre bitii microcodului Program Counter controleaza aceste intrari; combinatiile adecvate ale acestor biti permit PC-ului sa fie incrementat cu 1 sau cu o valoare specifica in cazul salturilor conditionate.De asemenea putem obtine si combinatia specifica saltului neconditiat pentru o operatie de tip “jump”. Ultimul bit de microcod controleaza de fapt iesirea catre memoria de instructiuni; cealalta iesire este conectata inapoi la intrarea sumatorului, este activată cand valoarea din PC este actualizata.

### 4.2.6 Instruction Memory

Este o memorie de tip ROM. Memoriile de tip ROM sunt o varianta particulara de stocare folosita in calculatoare, ele permitand doar operatii de citire in regimul obisnuit de utilizare. In aceasta memorie sunt stocate instructiunile pe care le va executa procesorul.

Este considerata ca o memoria de instrucțiuni ce va conține un program descris prin cod de asamblare care foloseste instructiuni de adaugare, schimbare, testare si ramificare cu ajutorul registriilor declarati in intermediul programului.

Aceasta memorie de instructiuni utilizeaza unitatea Program Counter drept un registru pentru adresarea instructiunii corespunzatoare care urmeaza apoi sa fie incarcata in Registrul de Instructiuni.

### 4.2.7 Data memory

Memoria de Date are propriul set de registre incorporate: MAR-cu ajutorul acestuia putem accesa adresa specifica din memoria de date, este utilizat atat la citire cat si la scriere. De asemenea avem registrul MBR care este un registru buffer si ne indica valoarea concreta pe care dorim sa o scriem in memorie sau pe care vrem sa o scriem. Intrarile trimise catre magistrale sunt trimise catre toate iesirile, dar utilizate numai la unitațile de receptie daca bitul de control corespunzator al unitatii de receptie este setat la 1.Unitatea de memorie dispune de un bit de control Read/Write (0=Read, 1=Write). Pentru o operație de citire (Read / Write = 0), adresa trimisa de la BUS 2 este copiata in registrul MAR doar la urmatoarea faza de ceas, memoria va fi citita si rezultatul copiat in registrul MBR. In urmatoarea faza de ceas 1, valoarea din registrul MBR este trimisă la BUS 1. Pentru o operatie de scriere (Read / Write= 1), adresa trimisa de la BUS 2 este copiata in MAR, iar adresa trimisa de la BUS1 copiata in MBR, astfel valoarea va fi scrisa in memorie.

### 4.2.8\_Unitatea\_microcod

Unitatea de Microcod conține memoria de microcoduri care este adresată de câmpul “**function**” al instructiunii noi preluate de IR în ciclul de ceas în care este recepționată o nouă instrucțiune din memoria de instrucțiuni și de Contorul de programe pentru Unitatea de Microcod (MPC) în ciclurile de ceas ulterioare. (Informatie eronata preluata din Microcod poate cauza ca ambele să apară simultan; acestaa eroare este detectata automat.)Unitatea de microcod are două câmpuri de control pentru microcod, unul pentru controlul ieșirilor sale (către BUS1, BUS2 și MPC) și unul pentru controlul executiei instrucțiunilor condiționate.

Unitatea de microcod are două ieșiri de date diferite, câmpul Adress sau Literal (Immediate) al instrucțiunii din IR, care poate fi trimis la BUS1 sau BUS2 și câmpul de adresă al cuvântului de microcod curent (CW) care poate fi trimis către Microprogram Counter.

Biții din cel de-al doilea câmp selecteaza codul de condiție ALU (~ CC0 sau CC1) și determina daca microcodul destinat Contorului de program, Memoriilor, Registrelor și ALU este trimis, in cazul in care condiția este îndeplinită sau in cazul in care condiția nu este îndeplinită. Această facilitate poate fi utilizată pentru a implementa instrucțiuni de salt conditionat. (BNEG, BNEZ).

**Emma2**

In cadrul acesteia se leaga toate componentele de mai sus.

# 5. Rezultate experimentale

## 5.1 Detalii de implementare

Proiectul a fost implementat prin mediul de dezvoltare Vivado, unde s-a efectuat si simularea.In continuare se vor prezenta componentele implementate ale procesorului EMMA-2:

## 5.2 Teste efectuate

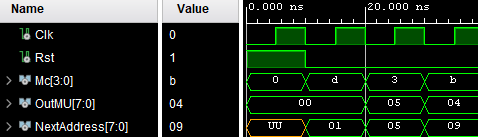
### 5.2.1 Teste generator de doua faze de ciclu de ceas



Putem observa ca Clock-ul alterneaza intre fazele crescatoare de rising’edge si fazele descrescatoare de falling’edge, astfel semnalul Clk isi schimba valoarea logica la fiecare 5ns. (Interval de timp impus de proiectant)

Astfel va alterna intre valorile ‘0’ si ‘1’.

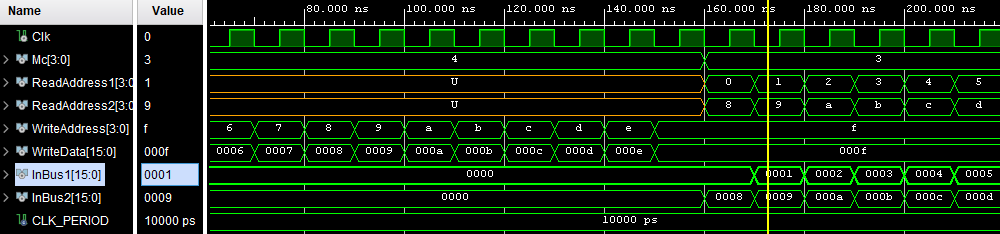
### 5.2.2 Teste MPC



La simularea MPC-ului am testat scenariile diferite care pot aparea la instructiuni distincte. In primul caz am setat semnalul Mc de microcod astfel ca adresa urmatoare sa se incrementeze: NextAddress <- NextAddress +1. Apoi am setat Mc astfel incat adresa urmatoare sa ia valoarea primita de la Unitatea microcod: NextAddress <- 5 ca si la un salt neconditionat. Iar la final am testat cazul cu saltul neconditionat in care

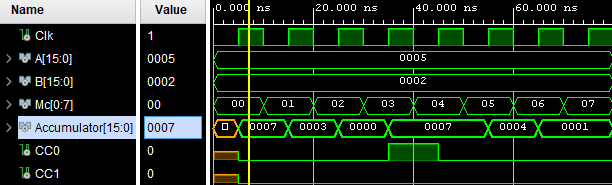
NextAddress<- NextAddress + 4; (5+4=9)

### 5.2.3 Teste Registrii



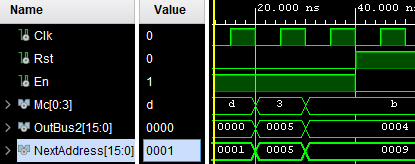
Simularea unitatii de Registrii s-a efectuat prin testarea etapei de scriere (fiecare registru primeste valoarea adresei) si etapa de citire in care citim in paralel valoarea de la registrul i si valoarea de la registrul i+8, astfel verificam mai rapid valorile scrise si corectitudinea functionarii.

### 5.2.4 Teste ALU



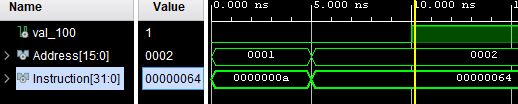
Simularea unitatii aritmetica-logica a presupus introducerea a dor registrii: A=5,B=2. Si a efectua pe rand toate operatiilede baza disponibile pe rand. (ex. 5+2=7, 5-2=3, etc.)

### 5.2.5 Teste Program Counter



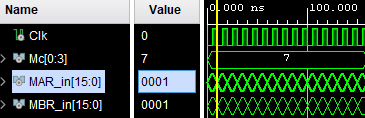
Simularea PC-lui s-a efectuat similar ca la MPC. Testam incrementarea: NextAddress <- NextAddress +1. Apoi am setat Mc, astfel incat adresa urmatoare sa ia valoarea primita de la Unitatea microcod: NextAddress <- 5 ca si la un salt neconditionat. Iar la final am testat cazul cu saltul neconditionat in care NextAddress<- NextAddress + 4; (5+4=9)

### 5.2.6 Teste Instruction Memory



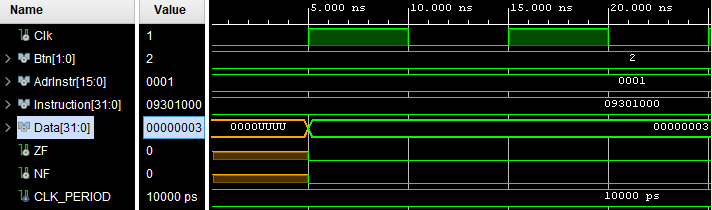
Simularea memoriei de instructiuni a fost efectuata prin detectarea unei valori egale cu 100, accesam valoarea din memorie cu ajutorul adresei, iar daca continutul la locatia respectiva este valoarea x”0064”, atunci semnalam acest aspect.

### 5.2.7 Teste Data Memory



Simularea Memorie de date implica procesul scierii in memorie realizat intr-un singur tact, si citirea totodata citirea datelor din memorie care se realizeaza in 2 faze de ceas.

### 5.2.8 Teste procesor



Testarea procesorului s-a realizat pe operatii single-cycle, precum primul exemplu cu adunare simpla.In acest caz avem in setul de registrii declarati 2 registrii cu valorile: $1=1, $2=2.Ca rezultat al adunarii am atribuit valoarea din registrul accumulator la semnalul Data, astfel indicand ca la adunarea celor doi registrii, rezultatul operatiei este egal cu 3.

# 6. Concluzii

Proiectul a presupus realizarea procesorului EMMA2, care implementeaza operatii de tip load/store si operatii logice sau aritmetice intre registre.EMMA2 se diferentiaza fata de alte procesoare prin arhitectura sa unica si modul de operare prin accesarea unei memorii de microcod pentru a determina control absolut asupra celorlalte componente.Aceasta gandire unica de a crea o arhitectura functionala pe baza de microcod mi se pare de mare folos pentru a da comenzi mai usor unui processor de performanta inalta ca acesta.

**Manual de utilizare (executia instructiunilor)**

Pentru a testa aplicatia, utilizatorul va trebui sa deschida proiectul prin mediul de proiectare Vivado, unde va putea viziona in detaliu descrierea fiecarei componente si arhitecturi aferente realizate.Asadar, continutul proiectului cuprinde toate fisierele .vhdl pentru realizarea procesorului, precum si testarea/simularea acestuia si a componentelor sale in detaliu.

Dupa deschiderea proiectului, utilizator poate selecta simularea unei componente specifice prin selectarea acesteia din panoul ***PROJECT MANAGER -> Sources -> Simulation*** sources unde va selecta optiunea “Set as Top” pentru componenta aleasa, astfel simularea va fi pentru componenta respectiva. Dupa aceasta selectie, utilizatorul are posibilitatea de a specifica timpul de rulare al simularii de la calea ***PROJECT MANAGER -> Settings -> Simulation ->*** ***x.sim.simulate.runtime*** unde va putea introduce timpul in nanosecunde(Recomand ca acesta sa se incadreze in intervalul de 200-1000ns).In final, utilizatorul poate selecta optiunea ***Flow Navigator ->*** ***Run Simulation,*** care va declansa deschiderea unei noi ferestre in cadrul proiectului, unde va putea observa formele de unda specifice (waveform) prin care poate verifica functionarea corecta.Pentru o analiza mai detaliata utilizatorul poate selecta optiunile Zoom Fit/Zoom In/Zoom Out, pentru a observa valori concrete pentru un termen specific.

Aceeasi procedura poate fi aplicata si la simularea integului processor, selectand fisierul “emma2\_proc\_tb” si pornind simularea in acest caz.Semnalele din tabelul aflat pe partea stanga a formelor de unda vor indica valorile specifice la momentele precizate de catre utilizator.

Pentru oprirea simularii se va selecta optiunea ***File->Close Simulation***.

# Bibliografie

# 

1. HASE Project

Institute for Computing Systems Architecture, School of Informatics, University of Edinburgh.

<http://www.icsa.inf.ed.ac.uk/cgi-bin/hase/emma-m.pl?arch2-t.html,arch2-f.html,menu2.html>

1. Roland N. Ibbett – A Simulation for Microcoding Exercises School of Informatics, University of Edinburgh. [http://delivery.acm.org/10.1145/1280000/1275601/a23-](http://delivery.acm.org/10.1145/1280000/1275601/a23-ibbett.pdf?key1=1275601&key2=1098694721&coll=GUIDE&dl=GUIDE&CFID=89237186&CFTOKEN=67716322)

[ibbett.pdf?key1=1275601&key2=1098694721&coll=GUIDE&dl=GUIDE&CFID=89237186&C FTOKEN=67716322](http://delivery.acm.org/10.1145/1280000/1275601/a23-ibbett.pdf?key1=1275601&key2=1098694721&coll=GUIDE&dl=GUIDE&CFID=89237186&CFTOKEN=67716322)

1. Harvard Architecture

<http://en.wikipedia.org/wiki/Harvard_architecture>

# Anexă

**Diagram, timeline

Description automatically generated**

**Diagram

Description automatically generated**

**Code**

***Registers:***

**library IEEE;**

**use IEEE.STD\_LOGIC\_1164.ALL;**

**use IEEE.STD\_LOGIC\_ARITH.ALL;**

**use IEEE.STD\_LOGIC\_UNSIGNED.ALL;**

**entity Registers is**

**Port ( Clk : in std\_logic;**

**Mc : in std\_logic\_vector (0 to 3);**

**ReadAddress1 : in std\_logic\_vector (3 downto 0);**

**ReadAddress2 : in std\_logic\_vector (3 downto 0);**

**WriteAddress : in std\_logic\_vector (3 downto 0);**

**WriteData : in std\_logic\_vector (15 downto 0);**

**InBus1 : out std\_logic\_vector (15 downto 0);**

**InBus2 : out std\_logic\_vector (15 downto 0));**

**end Registers;**

**architecture Behavioral of Registers is**

**type reg\_array is array(0 to 15) of std\_logic\_vector(15 downto 0);**

**signal reg\_file : reg\_array:=(**

**--X"0000",**

**X"0001", X"0002",**

**--X"0003",**

**--X"0004",X"0005",X"0006",X"0007",**

**--X"0008",X"0009",X"000A",X"000B",**

**--X"000C",X"000D",X"000E",X"000F",**

**others => X"0000");**

**signal RegWrite: STD\_LOGIC\_VECTOR(0 to 1):="00";**

**signal Output1, Output2: STD\_LOGIC:='0';**

**begin**

**RegWrite <= Mc(0 to 1);**

**Output1 <= Mc(2);**

**Output2 <= Mc(3);**

**--read from reg**

**InBus1 <= reg\_file(conv\_integer(ReadAddress1)) when Output1='1' else (others => '0'); --rs**

**InBus2 <= reg\_file(conv\_integer(ReadAddress2)) when Output2='1' else (others => '0'); --rt**

**--write in reg**

**process(Clk)**

**begin**

**if rising\_edge(Clk) and RegWrite="01" then**

**reg\_file(conv\_integer(WriteAddress)) <= WriteData;**

**end if;**

**end process;**

**end Behavioral;**

**library IEEE;**

**use IEEE.STD\_LOGIC\_1164.ALL;**

**use IEEE.STD\_LOGIC\_UNSIGNED.ALL;**

**use IEEE.NUMERIC\_STD.ALL;**

**entity Registers\_tb is**

**-- Port ( );**

**end Registers\_tb;**

**architecture Behavioral of Registers\_tb is**

**component Registers is**

**Port ( Clk : in std\_logic;**

**Mc : in std\_logic\_vector (0 to 3);**

**ReadAddress1 : in std\_logic\_vector (3 downto 0);**

**ReadAddress2 : in std\_logic\_vector (3 downto 0);**

**WriteAddress : in std\_logic\_vector (3 downto 0);**

**WriteData : in std\_logic\_vector (15 downto 0);**

**InBus1 : out std\_logic\_vector (15 downto 0);**

**InBus2 : out std\_logic\_vector (15 downto 0));**

**end component;**

**signal Clk : STD\_LOGIC := '0';**

**signal Mc : STD\_LOGIC\_VECTOR (3 downto 0);**

**signal ReadAddress1 : STD\_LOGIC\_VECTOR (3 downto 0);**

**signal ReadAddress2 : STD\_LOGIC\_VECTOR (3 downto 0);**

**signal WriteAddress : STD\_LOGIC\_VECTOR (3 downto 0);**

**signal WriteData : STD\_LOGIC\_VECTOR (15 downto 0);**

**signal InBus1 : STD\_LOGIC\_VECTOR (15 downto 0);**

**signal InBus2: STD\_LOGIC\_VECTOR (15 downto 0);**

**constant CLK\_PERIOD : TIME := 10 ns;**

**begin**

**DUT: Registers port map (Clk => Clk, Mc => Mc, ReadAddress1 => ReadAddress1,**

**ReadAddress2 => ReadAddress2, WriteAddress => WriteAddress,**

**WriteData => WriteData, InBus1 => InBus1, InBus2 => InBus2);**

**gen\_clk: process**

**begin**

**Clk <= '0';**

**wait for (CLK\_PERIOD/2);**

**Clk <= '1';**

**wait for (CLK\_PERIOD/2);**

**end process gen\_clk;**

**gen\_vect\_test: process**

**begin**

**Mc <= "0100";**

**for i in 0 to 15 loop**

**WriteAddress <= STD\_LOGIC\_VECTOR(to\_unsigned(i, WriteAddress'length));**

**WriteData <= STD\_LOGIC\_VECTOR(to\_unsigned(i, WriteData'length));**

**wait for CLK\_PERIOD;**

**end loop;**

**Mc <= "0011";**

**for i in 0 to 7 loop**

**ReadAddress1 <= std\_logic\_vector(to\_unsigned(i, ReadAddress1'length));**

**ReadAddress2 <= std\_logic\_vector(to\_unsigned(i+8, ReadAddress2'length));**

**wait for CLK\_PERIOD;**

**end loop;**

**wait;**

**end process;**

**end Behavioral;**

***Data Memory:***

**library IEEE;**

**use IEEE.STD\_LOGIC\_1164.ALL;**

**use IEEE.STD\_LOGIC\_UNSIGNED.ALL;**

**entity Data\_memory is**

**Port ( Clk : in STD\_LOGIC;**

**Mc: in STD\_LOGIC\_VECTOR(0 to 3);**

**MAR\_in : in STD\_LOGIC\_VECTOR(15 downto 0);**

**MBR\_in : in STD\_LOGIC\_VECTOR(15 downto 0);**

**MBR\_out : out STD\_LOGIC\_VECTOR(15 downto 0));**

**end Data\_memory;**

**architecture Behavioral of Data\_memory is**

**signal MAR\_input, MBR\_input, MBR\_output : STD\_LOGIC:='0';**

**signal Read\_Write : STD\_LOGIC:='0';**

**signal MBR\_int : STD\_LOGIC\_VECTOR(15 downto 0) := (others => '0');**

**type mem\_type is array (0 to 31) of STD\_LOGIC\_VECTOR(15 downto 0);**

**signal MEM : mem\_type := (**

**-- X"0000", X"0001", X"0002", X"0003",**

**-- X"0004", X"0005", X"0006", X"0007",**

**-- X"0008", X"0009", X"000A", X"000B",**

**-- X"000C", X"000D", X"000E", X"000F",**

**-- X"0010", X"0011", X"0012", X"0013",**

**-- X"0014", X"0015", X"0016", X"0017",**

**-- X"0018", X"0019", X"001A", X"001B",**

**-- X"001C", X"001D", X"001E", X"001F",**

**others => X"0000");**

**begin**

**MAR\_input <= Mc(0);**

**MBR\_input <= Mc(1);**

**Read\_Write <= Mc(2);**

**MBR\_output <= Mc(3);**

**MBR\_out <= MEM(conv\_integer(MAR\_in(5 downto 0)));**

**-- Data Memory --**

**process(clk)**

**begin**

**if rising\_edge(clk) then**

**if (Read\_Write = '1' and MAR\_input = '1' and MBR\_input='1') then --write**

**MEM(conv\_integer(MAR\_in(5 downto 0))) <= MBR\_in;**

**elsif (Read\_Write = '0' and MAR\_input = '1') then --read phase 0**

**MBR\_int <= MEM(conv\_integer(MAR\_in(5 downto 0)));**

**end if;**

**elsif falling\_edge(clk) then --read phase 1**

**if Read\_Write = '0' and MBR\_output='1' then**

**MBR\_out <= MBR\_int;**

**end if;**

**end if;**

**end process;**

**end Behavioral;**

**library IEEE;**

**use IEEE.STD\_LOGIC\_1164.ALL;**

**use IEEE.STD\_LOGIC\_UNSIGNED.ALL;**

**use IEEE.STD\_LOGIC\_ARITH.ALL;**

**entity DataMemory\_tb is**

**-- Port ( );**

**end DataMemory\_tb;**

**architecture Behavioral of DataMemory\_tb is**

**component Data\_memory is**

**Port ( Clk : in STD\_LOGIC;**

**Mc: in STD\_LOGIC\_VECTOR(0 to 3);**

**MAR\_in : in STD\_LOGIC\_VECTOR(15 downto 0);**

**MBR\_in : in STD\_LOGIC\_VECTOR(15 downto 0);**

**MBR\_out : out STD\_LOGIC\_VECTOR(15 downto 0));**

**end component;**

**signal Clk : STD\_LOGIC:='0';**

**signal Mc: STD\_LOGIC\_VECTOR(0 to 3):= (others =>'0');**

**signal MAR\_in : STD\_LOGIC\_VECTOR(15 downto 0):= (others =>'0');**

**signal MBR\_in : STD\_LOGIC\_VECTOR(15 downto 0):= (others =>'0');**

**signal MBR\_out : STD\_LOGIC\_VECTOR(15 downto 0):= (others =>'0');**

**constant CLK\_PERIOD : TIME := 10 ns;**

**begin**

**DUT: Data\_memory port map (Clk=>Clk, Mc => Mc, MAR\_in => MAR\_in,**

**MBR\_in => MBR\_in, MBR\_out =>MBR\_out);**

**gen\_clk: process**

**begin**

**Clk <= '0';**

**wait for (CLK\_PERIOD/2);**

**Clk <= '1';**

**wait for (CLK\_PERIOD/2);**

**end process gen\_clk;**

**gen\_vect\_test: process**

**begin**

**--scriere**

**Mc <= "0111";**

**for i in 0 to 15 loop**

**MAR\_in <= conv\_std\_logic\_vector(i,16);**

**MBR\_in <= conv\_std\_logic\_vector(i,16);**

**wait for CLK\_PERIOD;**

**end loop;**

**--citire**

**Mc <= "1001";**

**for i in 0 to 15 loop**

**MAR\_in <= conv\_std\_logic\_vector(i,16);**

**wait for CLK\_PERIOD;**

**end loop;**

**wait;**

**end process;**

**end Behavioral;**

***Instruction Memory***

**library IEEE;**

**use IEEE.STD\_LOGIC\_1164.ALL;**

**use IEEE.STD\_LOGIC\_UNSIGNED.ALL;**

**entity Instr\_memory is**

**Port ( Address: in STD\_LOGIC\_VECTOR(15 downto 0);**

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

**end Instr\_memory;**

**architecture Behavioral of Instr\_memory is**

**-- Memorie ROM**

**type tROM is array (0 to 255) of STD\_LOGIC\_VECTOR (31 downto 0);**

**signal ROM : tROM := (**

**-- PROGRAM DE TEST**

**b"00001001\_0011\_0001\_0010\_0000\_0000\_0000", --add $3,$1,$2**

**b"00001001\_0011\_0000\_0001\_0000\_0000\_0000", --add $4,$1,$0**

**--b"00000000\_0000\_0000\_0001\_0000\_0000\_0101", --jump 4**

**-- X"0000\_000A",**

**-- X"0000\_0064",**

**--X"0000\_03E8",**

**--X"0000\_2710",**

**others => X"0000\_0000");**

**begin**

**Instruction <= ROM(conv\_integer(Address));**

**end Behavioral;**

**library IEEE;**

**use IEEE.STD\_LOGIC\_1164.ALL;**

**use IEEE.STD\_LOGIC\_UNSIGNED.ALL;**

**use IEEE.STD\_LOGIC\_ARITH.ALL;**

**entity Instr\_Memory\_tb is**

**-- Port ( );**

**end Instr\_Memory\_tb;**

**architecture Behavioral of Instr\_Memory\_tb is**

**component Instr\_memory is**

**Port ( Address: in STD\_LOGIC\_VECTOR(15 downto 0);**

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

**end component;**

**signal val\_100 : STD\_LOGIC:= '0';**

**signal Address : STD\_LOGIC\_VECTOR(15 downto 0):= (others =>'0');**

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

**begin**

**DUT: Instr\_Memory port map (Address => Address, Instruction => Instruction);**

**gen\_vect\_test: process**

**begin**

**Address <= conv\_std\_logic\_vector(1,16);**

**wait for 5 ns;**

**if (Instruction = x"0000\_0064") then**

**val\_100<='1';**

**end if;**

**Address <= conv\_std\_logic\_vector(2,16);**

**wait for 5 ns;**

**if (Instruction = x"0000\_0064") then**

**val\_100<='1';**

**end if;**

**wait;**

**end process;**

**end Behavioral;**

***Program Counter***

**library IEEE;**

**use IEEE.STD\_LOGIC\_1164.ALL;**

**use IEEE.STD\_LOGIC\_UNSIGNED.ALL;**

**entity Program\_counter is**

**Port ( Clk : in STD\_LOGIC;**

**Rst : in STD\_LOGIC;**

**En : in STD\_LOGIC;**

**Mc: in STD\_LOGIC\_VECTOR(0 to 3);**

**OutBus2: in STD\_LOGIC\_VECTOR(15 downto 0);**

**NextAddress: out STD\_LOGIC\_VECTOR(15 downto 0));**

**end Program\_counter;**

**architecture Behavioral of Program\_counter is**

**signal PC : STD\_LOGIC\_VECTOR(15 downto 0) := (others => '0');**

**signal NextAddr: STD\_LOGIC\_VECTOR(15 downto 0):= (others => '0');**

**begin**

**-- Output2**

**NextAddress <= NextAddr when Mc(3)='1';**

**-- Program Counter**

**process(Clk)**

**begin**

**if rising\_edge(Clk) then**

**if Rst = '1' then**

**PC <= (others => '0');**

**elsif En = '1' then**

**PC <= NextAddr;**

**end if;**

**end if;**

**end process;**

**-- Adder**

**process(Mc)**

**begin**

**case Mc(0 to 2) is**

**when "110" => NextAddr <= PC + 1; -- next instruction**

**when "101" => NextAddr <= PC + OutBus2; -- branch**

**when "001" => NextAddr <= OutBus2; --jump**

**when others => NextAddr <= PC + 1;**

**end case;**

**end process;**

**end Behavioral;**

**library IEEE;**

**use IEEE.STD\_LOGIC\_1164.ALL;**

**use IEEE.STD\_LOGIC\_UNSIGNED.ALL;**

**use IEEE.STD\_LOGIC\_ARITH.ALL;**

**entity Program\_counter\_tb is**

**-- Port ( );**

**end Program\_counter\_tb;**

**architecture Behavioral of Program\_counter\_tb is**

**component Program\_counter is**

**Port ( Clk : in STD\_LOGIC;**

**Rst : in STD\_LOGIC;**

**En : in STD\_LOGIC;**

**Mc: in STD\_LOGIC\_VECTOR(0 to 3);**

**OutBus2: in STD\_LOGIC\_VECTOR(15 downto 0);**

**NextAddress: out STD\_LOGIC\_VECTOR(15 downto 0));**

**end component;**

**signal Clk,Rst,En : STD\_LOGIC:='0';**

**signal Mc: STD\_LOGIC\_VECTOR(0 to 3):= (others =>'0');**

**signal OutBus2 : STD\_LOGIC\_VECTOR(15 downto 0):= (others =>'0');**

**signal NextAddress : STD\_LOGIC\_VECTOR(15 downto 0):= (others =>'0');**

**constant CLK\_PERIOD : TIME := 10 ns;**

**begin**

**DUT: Program\_counter port map (Clk=>Clk, Rst=>Rst, En=>En, Mc => Mc, OutBus2 => OutBus2,**

**NextAddress => NextAddress);**

**gen\_clk: process**

**begin**

**Clk <= '0';**

**wait for (CLK\_PERIOD/2);**

**Clk <= '1';**

**wait for (CLK\_PERIOD/2);**

**end process gen\_clk;**

**gen\_vect\_test: process**

**begin**

**Rst<='1'; En<='0';**

**wait for CLK\_PERIOD;**

**Rst<='0'; En<='1';**

**Mc<="1101";**

**wait for CLK\_PERIOD;**

**Mc<="0011";**

**OutBus2 <= conv\_std\_logic\_vector(5,16);**

**wait for CLK\_PERIOD;**

**Mc<="1011";**

**OutBus2 <= conv\_std\_logic\_vector(4,16);**

**wait for CLK\_PERIOD;**

**Rst<='1'; En<='0';**

**wait;**

**end process;**

**end Behavioral;**

***Microcode Program Counter:***

**library IEEE;**

**use IEEE.STD\_LOGIC\_1164.ALL;**

**use IEEE.STD\_LOGIC\_UNSIGNED.ALL;**

**entity Microprogram\_counter is**

**Port ( Clk : in STD\_LOGIC;**

**Rst : in STD\_LOGIC;**

**Mc: in STD\_LOGIC\_VECTOR(0 to 3);**

**OutMU: in STD\_LOGIC\_VECTOR(7 downto 0);**

**NextAddress: out STD\_LOGIC\_VECTOR(7 downto 0));**

**end Microprogram\_counter;**

**architecture Behavioral of Microprogram\_counter is**

**signal MPC : STD\_LOGIC\_VECTOR(7 downto 0) := (others => '0');**

**signal NextAddr: STD\_LOGIC\_VECTOR(7 downto 0);**

**begin**

**--Output2**

**NextAddress <= NextAddr when Mc(3)='1';**

**-- Program Counter**

**process(Clk)**

**begin**

**if rising\_edge(Clk) then**

**if Rst = '1' then**

**MPC <= (others => '0');**

**else**

**MPC <= NextAddr;**

**end if;**

**end if;**

**end process;**

**-- Adder**

**process(Mc)**

**begin**

**case Mc(0 to 2) is**

**when "110" => NextAddr <= MPC + 1; -- next instruction**

**when "101" => NextAddr <= MPC + OutMU; -- branch**

**when "001" => NextAddr <= OutMU; --jump**

**when others => NextAddr <= MPC + 1;**

**end case;**

**end process;**

**end Behavioral;**

**library IEEE;**

**use IEEE.STD\_LOGIC\_1164.ALL;**

**use IEEE.STD\_LOGIC\_UNSIGNED.ALL;**

**use IEEE.STD\_LOGIC\_ARITH.ALL;**

**entity Microprogram\_counter\_tb is**

**-- Port ( );**

**end Microprogram\_counter\_tb;**

**architecture Behavioral of Microprogram\_counter\_tb is**

**component Microprogram\_counter is**

**Port ( Clk : in STD\_LOGIC;**

**Rst : in STD\_LOGIC;**

**Mc: in STD\_LOGIC\_VECTOR(0 to 3);**

**OutMU: in STD\_LOGIC\_VECTOR(7 downto 0);**

**NextAddress: out STD\_LOGIC\_VECTOR(7 downto 0));**

**end component;**

**signal Clk,Rst : STD\_LOGIC:='0';**

**signal Mc: STD\_LOGIC\_VECTOR(0 to 3):= (others =>'0');**

**signal OutMU : STD\_LOGIC\_VECTOR(7 downto 0):= (others =>'0');**

**signal NextAddress : STD\_LOGIC\_VECTOR(7 downto 0):= (others =>'0');**

**constant CLK\_PERIOD : TIME := 10 ns;**

**begin**

**DUT: Microprogram\_counter port map (Clk=>Clk, Rst=>Rst, Mc => Mc, OutMU => OutMU,**

**NextAddress => NextAddress);**

**gen\_clk: process**

**begin**

**Clk <= '0';**

**wait for (CLK\_PERIOD/2);**

**Clk <= '1';**

**wait for (CLK\_PERIOD/2);**

**end process gen\_clk;**

**gen\_vect\_test: process**

**begin**

**Rst<='1';**

**wait for CLK\_PERIOD;**

**Rst<='0';**

**Mc<="1101";**

**wait for CLK\_PERIOD;**

**Mc<="0011";**

**OutMU <= conv\_std\_logic\_vector(5,8);**

**wait for CLK\_PERIOD;**

**Mc<="1011";**

**OutMU <= conv\_std\_logic\_vector(4,8);**

**wait for CLK\_PERIOD;**

**Rst<='1';**

**wait;**

**end process;**

**end Behavioral;**

***MICROCODE UNIT:***

**library IEEE;**

**use IEEE.STD\_LOGIC\_1164.ALL;**

**USE ieee.numeric\_std.ALL;**

**entity Microcode\_Unit is**

**Port ( IR : in STD\_LOGIC\_VECTOR (31 downto 0);**

**Mc : in STD\_LOGIC\_VECTOR (0 to 7);**

**AdrMC : in STD\_LOGIC\_VECTOR (7 downto 0);**

**AdrD : out STD\_LOGIC\_VECTOR (3 downto 0);**

**AdrSA : out STD\_LOGIC\_VECTOR (3 downto 0);**

**AdrSB : out STD\_LOGIC\_VECTOR (3 downto 0);**

**InBus1 : out STD\_LOGIC\_VECTOR (15 downto 0);**

**InBus2 : out STD\_LOGIC\_VECTOR (15 downto 0);**

**InMPC : out STD\_LOGIC\_VECTOR (7 downto 0);**

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

**end Microcode\_Unit;**

**architecture Behavioral of Microcode\_Unit is**

**-- MICROCODE MEMORY --**

**type mem\_type is array (0 to 255) of STD\_LOGIC\_VECTOR(31 downto 0);**

**signal Mc\_memory : mem\_type := (**

**x"400B\_0000", --0 JUMP**

**x"0000\_0000", --1 JREG**

**x"0000\_0000", --2 BEQZ**

**x"1030\_0000", --3 BNEG**

**x"0000\_0000", --4 LD**

**x"0000\_0000", --5 LDL**

**x"0000\_0000", --6 LDX**

**x"0000\_0000", --7 ST**

**x"0000\_0000", --8 STX**

**x"000D\_3000", --9 ADD**

**x"0000\_0000", --10 ADDL**

**x"000D\_3001", --11 SUB**

**others => X"0000\_0000");**

**type op is (JUMP, JREG, BEQZ, BNEG, LD, LDL, LDX, ST, STX , ADD, ADDL,**

**SUB, SUBL, ANDD, ANDL, ORR, ORL, XORR, XORL, SLLS, SLLL, SRLS, SRLL,**

**SRAA, SRAL, MUL, MULL, DIV, DIVL, OP1, OP2, STOP);**

**signal stare : op := JUMP;**

**signal toBus1,toBus2,toMPC : STD\_LOGIC :='0';**

**signal Immediate : STD\_LOGIC\_VECTOR(15 downto 0):= (others =>'0');**

**signal Address : STD\_LOGIC\_VECTOR(7 downto 0):= (others =>'0');**

**begin**

**AdrD <= IR(23 downto 20);**

**AdrSA <= IR(19 downto 16);**

**AdrSB <= IR(15 downto 12);**

**Immediate <= IR(15 downto 0);**

**toBus1 <= Mc(1);**

**toBus2 <= Mc(2);**

**toMPC <= Mc(3);**

**InBus1 <= Immediate when toBus1='1' else (others=>'0');**

**InBus2 <= Immediate when toBus2='1' else (others=>'0');**

**InMPC <= Address when toMPC='1' else (others=>'0');**

**Operation:process(IR)**

**begin**

**case IR(31 downto 24) is**

**when "00000000" => stare <= JUMP;**

**when "00000001" => stare <= JREG;**

**when "00000010" => stare <= BEQZ;**

**when "00000011" => stare <= BNEG;**

**when "00000100" => stare <= LD;**

**when "00000101" => stare <= LDL;**

**when "00000110" => stare <= LDX;**

**when "00000111" => stare <= ST;**

**when "00001000" => stare <= STX;**

**when "00001001" => stare <= ADD;**

**when "00001010" => stare <= ADDL;**

**when "00001011" => stare <= SUB;**

**when "00001100" => stare <= SUBL;**

**when "00001101" => stare <= ANDD;**

**when "00001110" => stare <= ANDL;**

**when "00001111" => stare <= ORR;**

**when "00010000" => stare <= ORL;**

**when "00010001" => stare <= XORR;**

**when "00010010" => stare <= XORL;**

**when "00010011" => stare <= SLLS;**

**when "00010100" => stare <= SLLL;**

**when "00010101" => stare <= SRLS;**

**when "00010110" => stare <= SRLL;**

**when "00010111" => stare <= SRAA;**

**when "00011000" => stare <= SRAL;**

**when "00011001" => stare <= MUL;**

**when "00011010" => stare <= MULL;**

**when "00011011" => stare <= DIV;**

**when "00011100" => stare <= DIVL;**

**when "00011101" => stare <= OP1;**

**when "00011110" => stare <= OP2;**

**when "00011111" => stare <= STOP;**

**when others => stare <= ADD;**

**end case;**

**end process Operation;**

**process(stare)**

**begin**

**case stare is**

**when JUMP => Micro <= Mc\_memory(0);**

**when JREG => Micro <= Mc\_memory(1);**

**when BEQZ => Micro <= Mc\_memory(2);**

**when BNEG => Micro <= Mc\_memory(3);**

**when LD => Micro <= Mc\_memory(4); Address<=x"38";**

**when LDL => Micro <= Mc\_memory(5);**

**when LDX => Micro <= Mc\_memory(6);**

**when ST => Micro <= Mc\_memory(7);**

**when STX => Micro <= Mc\_memory(8);**

**when ADD => Micro <= Mc\_memory(9);**

**when ADDL => Micro <= Mc\_memory(10);**

**when SUB => Micro <= Mc\_memory(11);**

**when SUBL => Micro <= Mc\_memory(12);**

**when ANDD => Micro <= Mc\_memory(13);**

**when ANDL => Micro <= Mc\_memory(14);**

**when ORR => Micro <= Mc\_memory(15);**

**when ORL => Micro <= Mc\_memory(16);**

**when XORR => Micro <= Mc\_memory(17);**

**when XORL => Micro <= Mc\_memory(18);**

**when SLLS => Micro <= Mc\_memory(19);**

**when SLLL => Micro <= Mc\_memory(20);**

**when SRLS => Micro <= Mc\_memory(21);**

**when SRLL => Micro <= Mc\_memory(22);**

**when SRAA => Micro <= Mc\_memory(23);**

**when SRAL => Micro <= Mc\_memory(24);**

**when MUL => Micro <= Mc\_memory(25);**

**when MULL => Micro <= Mc\_memory(26);**

**when DIV => Micro <= Mc\_memory(27);**

**when DIVL => Micro <= Mc\_memory(28);**

**when OP1 => Micro <= Mc\_memory(29);**

**when OP2 => Micro <= Mc\_memory(30);**

**when STOP => Micro <= Mc\_memory(31);**

**when others => Micro <= Mc\_memory(to\_integer(unsigned(AdrMc)));**

**end case;**

**end process;**

**end Behavioral;**

***ALU:***

**library IEEE;**

**use IEEE.STD\_LOGIC\_1164.ALL;**

**use IEEE.STD\_LOGIC\_UNSIGNED.ALL;**

**use IEEE.numeric\_std.ALL;**

**entity ALU is**

**Port ( Clk: in STD\_LOGIC;**

**A : in STD\_LOGIC\_VECTOR(15 downto 0);**

**B : in STD\_LOGIC\_VECTOR(15 downto 0);**

**Mc: in std\_logic\_vector(0 to 7);**

**Accumulator : out STD\_LOGIC\_VECTOR(15 downto 0);**

**CC0 : out STD\_LOGIC;**

**CC1 : out STD\_LOGIC);**

**end ALU;**

**architecture Behavioral of ALU is**

**signal func: STD\_LOGIC\_VECTOR(0 to 2);**

**signal ALUCtrl : STD\_LOGIC\_VECTOR(2 downto 0);**

**signal ALUResAux : STD\_LOGIC\_VECTOR(15 downto 0);**

**begin**

**func <= Mc(5 to 7);**

**-- ALU**

**process(Clk)**

**begin**

**if(rising\_edge(Clk)) then**

**case func is**

**when "000" => -- ADD**

**ALUResAux <= A + B;**

**when "001" => -- SUB**

**ALUResAux <= A - B;**

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

**ALUResAux <= A and B;**

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

**ALUResAux <= A or B;**

**when "100" => -- XOR**

**ALUResAux <= A xor B;**

**when "101" => -- SLL**

**ALUResAux <= B(14 downto 0) & "0";**

**when "110" => -- SRL**

**ALUResAux <= "0" & B(15 downto 1);**

**when "111" => -- SRA**

**ALUResAux <= B(0) & B(15 downto 1);**

**when others =>**

**ALUResAux <= (others=>'0');**

**end case;**

**-- rez = 0**

**case ALUResAux is**

**when X"0000" => CC0 <= '1';**

**when others => CC0 <= '0';**

**end case;**

**-- rez < 0**

**case ALUResAux(15) is**

**when '1' => CC1 <= '1';**

**when others => CC1 <= '0';**

**end case;**

**end if;**

**end process;**

**-- ALU rezultat**

**Accumulator <= ALUResAux;**

**end Behavioral;**

**library IEEE;**

**use IEEE.STD\_LOGIC\_1164.ALL;**

**use IEEE.STD\_LOGIC\_UNSIGNED.ALL;**

**use IEEE.STD\_LOGIC\_ARITH.ALL;**

**entity ALU\_tb is**

**-- Port ( );**

**end ALU\_tb;**

**architecture Behavioral of ALU\_tb is**

**component ALU is**

**Port ( Clk : in STD\_LOGIC;**

**A : in STD\_LOGIC\_VECTOR(15 downto 0);**

**B : in STD\_LOGIC\_VECTOR(15 downto 0);**

**Mc: in std\_logic\_vector(0 to 7);**

**Accumulator : out STD\_LOGIC\_VECTOR(15 downto 0);**

**CC0 : out STD\_LOGIC;**

**CC1 : out STD\_LOGIC);**

**end component;**

**signal Clk : STD\_LOGIC:='0';**

**signal A,B: STD\_LOGIC\_VECTOR(15 downto 0):= (others =>'0');**

**signal Mc : STD\_LOGIC\_VECTOR(0 to 7):= (others =>'0');**

**signal Accumulator : STD\_LOGIC\_VECTOR(15 downto 0):= (others =>'0');**

**signal CC0,CC1 : STD\_LOGIC:='0';**

**constant CLK\_PERIOD : TIME := 10 ns;**

**begin**

**DUT: ALU port map (Clk=>Clk, A => A, B => B, Mc => Mc, Accumulator => Accumulator, CC0=>CC0, CC1 => CC1);**

**gen\_clk: process**

**begin**

**Clk <= '0';**

**wait for (CLK\_PERIOD/2);**

**Clk <= '1';**

**wait for (CLK\_PERIOD/2);**

**end process gen\_clk;**

**gen\_vect\_test: process**

**begin**

**A <= conv\_std\_logic\_vector(5,16);**

**B <= conv\_std\_logic\_vector(2,16);**

**for i in 0 to 7 loop**

**Mc <= conv\_std\_logic\_vector(i,8);**

**wait for CLK\_PERIOD;**

**end loop;**

**wait;**

**end process;**

**end Behavioral;**

***MPG:***

**library IEEE;**

**use IEEE.STD\_LOGIC\_1164.ALL;**

**use IEEE.STD\_LOGIC\_UNSIGNED.ALL;**

**entity MPG is**

**Port ( en : out STD\_LOGIC;**

**input : in STD\_LOGIC;**

**clock : in STD\_LOGIC);**

**end MPG;**

**architecture Behavioral of MPG is**

**signal count\_int : STD\_LOGIC\_VECTOR (15 downto 0) := (others => '0');**

**signal Q1 : STD\_LOGIC := '0';**

**signal Q2 : STD\_LOGIC := '0';**

**signal Q3 : STD\_LOGIC := '0';**

**begin**

**en <= Q2 AND (not Q3);**

**process (clock)**

**begin**

**if clock='1' and clock'event then**

**count\_int <= count\_int + 1;**

**end if;**

**end process;**

**process (clock)**

**begin**

**if clock'event and clock='1' then**

**if count\_int(15 downto 0) = "1111111111111111" then**

**Q1 <= input;**

**end if;**

**end if;**

**end process;**

**process (clock)**

**begin**

**if clock'event and clock='1' then**

**Q2 <= Q1;**

**Q3 <= Q2;**

**end if;**

**end process;**

**end Behavioral;**

***EMMA\_2:***

**library IEEE;**

**use IEEE.STD\_LOGIC\_1164.ALL;**

**entity emma2\_proc is**

**Port ( Clk : in STD\_LOGIC;**

**Btn : in STD\_LOGIC\_VECTOR (1 downto 0);**

**AdrInstr : out STD\_LOGIC\_VECTOR (15 downto 0);**

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

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

**ZF : out STD\_LOGIC;**

**NF : out STD\_LOGIC);**

**end emma2\_proc;**

**architecture Behavioral of emma2\_proc is**

**component MPG is**

**Port ( en : out STD\_LOGIC;**

**input : in STD\_LOGIC;**

**clock : in STD\_LOGIC);**

**end component;**

**component ALU is**

**Port ( Clk: in STD\_LOGIC;**

**A : in STD\_LOGIC\_VECTOR(15 downto 0);**

**B : in STD\_LOGIC\_VECTOR(15 downto 0);**

**Mc: in std\_logic\_vector(0 to 7);**

**Accumulator : out STD\_LOGIC\_VECTOR(15 downto 0);**

**CC0 : out STD\_LOGIC;**

**CC1 : out STD\_LOGIC);**

**end component;**

**component Data\_memory is**

**Port ( Clk : in STD\_LOGIC;**

**Mc: in STD\_LOGIC\_VECTOR(0 to 3);**

**MAR\_in : in STD\_LOGIC\_VECTOR(15 downto 0);**

**MBR\_in : in STD\_LOGIC\_VECTOR(15 downto 0);**

**MBR\_out : out STD\_LOGIC\_VECTOR(15 downto 0));**

**end component;**

**component Instr\_memory is**

**Port ( Address: in STD\_LOGIC\_VECTOR(15 downto 0);**

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

**end component;**

**component Registers is**

**Port ( Clk : in std\_logic;**

**Mc : in std\_logic\_vector (0 to 3);**

**ReadAddress1 : in std\_logic\_vector (3 downto 0);**

**ReadAddress2 : in std\_logic\_vector (3 downto 0);**

**WriteAddress : in std\_logic\_vector (3 downto 0);**

**WriteData : in std\_logic\_vector (15 downto 0);**

**InBus1 : out std\_logic\_vector (15 downto 0);**

**InBus2 : out std\_logic\_vector (15 downto 0));**

**end component;**

**component Program\_counter is**

**Port ( Clk : in STD\_LOGIC;**

**Rst : in STD\_LOGIC;**

**En : in STD\_LOGIC;**

**Mc: in STD\_LOGIC\_VECTOR(0 to 3);**

**OutBus2: in STD\_LOGIC\_VECTOR(15 downto 0);**

**NextAddress: out STD\_LOGIC\_VECTOR(15 downto 0));**

**end component;**

**component Microprogram\_counter is**

**Port ( Clk : in STD\_LOGIC;**

**Rst : in STD\_LOGIC;**

**Mc: in STD\_LOGIC\_VECTOR(0 to 3);**

**OutMU: in STD\_LOGIC\_VECTOR(7 downto 0);**

**NextAddress: out STD\_LOGIC\_VECTOR(7 downto 0));**

**end component;**

**component Microcode\_Unit is**

**Port ( IR : in STD\_LOGIC\_VECTOR (31 downto 0);**

**Mc : in STD\_LOGIC\_VECTOR (0 to 7);**

**AdrMC : in STD\_LOGIC\_VECTOR (7 downto 0);**

**AdrD : out STD\_LOGIC\_VECTOR (3 downto 0);**

**AdrSA : out STD\_LOGIC\_VECTOR (3 downto 0);**

**AdrSB : out STD\_LOGIC\_VECTOR (3 downto 0);**

**InBus1 : out STD\_LOGIC\_VECTOR (15 downto 0);**

**InBus2 : out STD\_LOGIC\_VECTOR (15 downto 0);**

**InMPC : out STD\_LOGIC\_VECTOR (7 downto 0);**

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

**end component;**

**--MPG**

**signal en,rst: STD\_LOGIC:='0';**

**--ALU**

**signal ACC: STD\_LOGIC\_VECTOR (15 downto 0):= (others=>'0');**

**signal CC0,CC1: STD\_LOGIC:='0';**

**--MU**

**signal InMPC: STD\_LOGIC\_VECTOR (7 downto 0):= (others=>'0');**

**signal InBus1\_1,InBus2\_1: STD\_LOGIC\_VECTOR (15 downto 0):= (others=>'0');**

**signal Mc\_out: STD\_LOGIC\_VECTOR (0 to 31):= (others=>'0');**

**--REG**

**signal RA1,RA2,WA: STD\_LOGIC\_VECTOR (3 downto 0):= (others=>'0');**

**signal InBus1\_2,InBus2\_2: STD\_LOGIC\_VECTOR (15 downto 0):= (others=>'0');**

**--MPC**

**signal AdrMc: STD\_LOGIC\_VECTOR (7 downto 0):= (others=>'0');**

**--IM**

**signal Address: STD\_LOGIC\_VECTOR (15 downto 0):= (others=>'0');**

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

**--DATA**

**signal InBus1\_3: STD\_LOGIC\_VECTOR (15 downto 0):= (others=>'0');**

**--BUS**

**signal Bus1,Bus2: STD\_LOGIC\_VECTOR (15 downto 0):= (others=>'0');**

**begin**

**MPG1: MPG port map(en=>en, input=>btn(0), clock=>Clk);**

**MPG2: MPG port map(en=>rst, input=>btn(1), clock=>Clk);**

**MU: Microcode\_Unit port map( IR => Instr, AdrMc => AdrMc, Mc => Mc\_out(0 to 7), AdrD => WA, AdrSA =>RA1,**

**AdrSB => RA2, InBus1 => InBus1\_1, InBus2 => InBus2\_1, InMPC => InMPC, Micro => Mc\_out);**

**MPC: Microprogram\_counter port map( Clk => Clk, Rst => rst, Mc => Mc\_out(8 to 11),**

**OutMU => inMPC, NextAddress => AdrMc);**

**PC: Program\_counter port map( Clk => Clk, Rst => rst, En => en, Mc => Mc\_out(12 to 15),**

**OutBus2 => Bus2, NextAddress => Address);**

**REG: Registers port map ( Clk => Clk, Mc => Mc\_out(16 to 19), ReadAddress1 => RA1, ReadAddress2 => RA2,**

**WriteAddress => WA, WriteData => ACC, InBus1 => InBus1\_2, InBus2 => InBus2\_2);**

**DM: Data\_memory port map ( Clk => Clk, Mc => Mc\_out(20 to 23),**

**MAR\_in => Bus2, MBR\_in => Bus1, MBR\_out => InBus1\_3);**

**IM: Instr\_memory port map( Address => Address, Instruction => Instr);**

**ALU1: ALU port map( Clk => Clk, A => Bus1, B => Bus2, Mc => Mc\_out(24 to 31),**

**Accumulator => ACC, CC0 => CC0, CC1 => CC1);**

**-- BUS1 --**

**B1: process(InBus1\_1,InBus1\_2,InBus1\_3)**

**begin**

**if(Mc\_out(1)='1') then**

**Bus1 <= InBus1\_1;**

**elsif(Mc\_out(18)='1') then**

**Bus1 <= InBus1\_2;**

**elsif(Mc\_out(23)='1') then**

**Bus1 <= InBus1\_3;**

**end if;**

**end process;**

**-- BUS2 --**

**B2: process(InBus2\_1,InBus2\_2,ACC)**

**begin**

**if(Mc\_out(2)='1') then**

**Bus2 <= InBus2\_1;**

**elsif(Mc\_out(19)='1') then**

**Bus2 <= InBus2\_2;**

**elsif(Mc\_out(27)='1') then**

**Bus2 <= ACC;**

**end if;**

**end process;**

**--Output**

**AdrInstr <= Address;**

**Instruction <= Instr;**

**Data <= x"0000" & ACC ;**

**ZF <= CC0;**

**NF <= CC1;**

**end Behavioral;**

***EMMA\_2\_TESTBENCH:***

**library IEEE;**

**use IEEE.STD\_LOGIC\_1164.ALL;**

**entity emma2\_proc\_tb is**

**-- Port ( );**

**end emma2\_proc\_tb;**

**architecture Behavioral of emma2\_proc\_tb is**

**component emma2\_proc is**

**Port ( Clk : in STD\_LOGIC;**

**Btn : in STD\_LOGIC\_VECTOR (1 downto 0);**

**AdrInstr : out STD\_LOGIC\_VECTOR (15 downto 0);**

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

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

**ZF : out STD\_LOGIC;**

**NF : out STD\_LOGIC);**

**end component;**

**signal Clk : STD\_LOGIC:='0';**

**signal Btn : STD\_LOGIC\_VECTOR (1 downto 0):=(others=>'0');**

**signal AdrInstr : STD\_LOGIC\_VECTOR (15 downto 0):=(others=>'0');**

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

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

**signal ZF,NF : STD\_LOGIC:='0';**

**constant CLK\_PERIOD : TIME := 10 ns;**

**begin**

**DUT: emma2\_proc port map (Clk=>Clk, Btn => Btn, AdrInstr => AdrInstr,Instruction => Instruction,**

**Data =>Data, ZF => ZF, NF => NF);**

**gen\_clk: process**

**begin**

**Clk <= '0';**

**wait for (CLK\_PERIOD/2);**

**Clk <= '1';**

**wait for (CLK\_PERIOD/2);**

**end process gen\_clk;**

**gen\_vect\_test: process**

**begin**

**Btn<="10";**

**wait for 100 ns;**

**Btn<="01";**

**wait;**

**end process;**

**end Behavioral;**