Friday, November 21, 2025

Unitate de management pentru controlul memoriei cache pentru multi-core

Student : Barna Răzvan

Structura Sistemelor de Calcul

Universitatea Tehnica din Cluj-Napoca

11 Octombrie 2025

Friday, November 21, 2025

Cuprins

1. Introducere

1.1 Context

Scopul proiectul este proiectarea unei unități de management pentru controlul memoriei cache, într-un context multi-core, cu implementarea unui protocol de coerență de tip snoopy. Arhitectura procesoarelor multi-core reprezintă un pas major în evoluția sistemelor de calcul, fiind una dintre principalele direcții de dezvoltare în domeniul hardware modern.

Totuși, această abordare nu este lipsită de provocări, principala problemă fiind menținerea sincronizării datelor între memoria cache a fiecărui nucleu și memoria principală. Lipsa unei

coerențe eficiente poate duce la inconsistențe și comportamente neașteptate în execuția programelor paralele.

1.2 Obiective

Întregul proiect va fi proiectat in VHDL și inclus într-un proiect Xilinx Vivado. El va conține 2 procesoare MIPS single-cycle , fiecare acționând ca un nucleu, având propria sa memorie cache de nivel 1.Intre aceste memorii cache și memoria principala va fi proiectata o magistrala de date, unde fiecare core va face una sau mai multe instrucțiuni în funcție de ce comenzi transmite cealaltă unitate de calcul.

Pe această magistrală de cache se va implementa un protocol de coerență de tip Snoopy, care va asigura consistența datelor între cele două nuclee.

1.2.1 Tipurile de protocoale de tip Snoopy

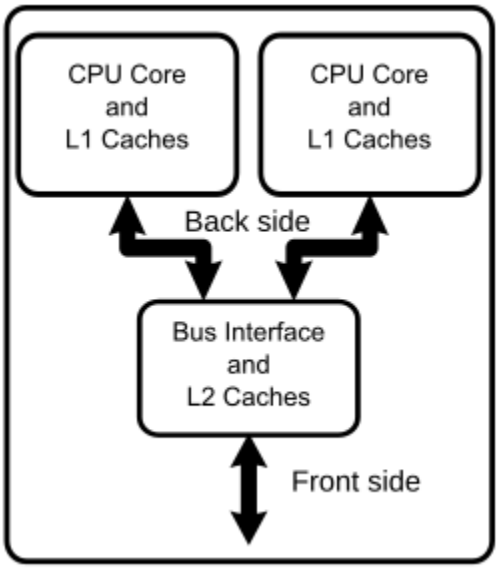
- MSI (Modified, Shared, Invalid) : asigură coerența prin invalidări și transferuri între cache-uri.
- MESI : Extinde MSI adăugând starea Exclusive, care indică faptul că linia este doar într-un cache și este curate.
- MOESI : Adaugă starea Owned, care permite unui cache să partajeze o linie modificată cu altele, fără a o scrie imediat în memorie.
- MESIF (Modified, Exclusive, Shared, Invalid, Forword) : Adaugă starea Forward, care desemnează un cache responsabil cu furnizarea datelor către altele, optimizând răspunsurile la cereri de citire.

2. Cercetare Bibliografică

2.1 Ce este și din ce este format un core?

CPU este Unitatea Centrala de Procesare, reprezentând creierul unui calculator. Un nucleu CPU este o singură unitate de procesare din cadrul procesorului care poate executa instrucțiuni. Cu cât procesorul are mai multe core-uri , cu atât mai multe sarcini poate executa în mod paralel.

Fiecare core poate citi, decoda și executa instrucțiuni în mod independent ,exact ca un procesor complet, doar că împărțind același cip cu alte nuclee. Ele sunt formate din : unitatea de control, unitatea aritmetica logica, registre, memoria cache de nivel 1.



Figură 1: Diagrama unui procesor dual-core generic cu cache-uri locale de nivel 1 ale CPU și o memorie cache partajată de nivel 2.

2.2 Ce reprezintă tehnica multi-core?

Tehnica multi-core reprezintă o metodă de proiectare a procesoarelor în care mai multe nuclee de procesare sunt integrate pe același cip.

Scopul principal este creșterea performanței de procesare fără a mări foarte mult frecvența de ceas si executarea paralelă a mai multor sarcini sau fire de execuție. Pentru o lungă perioadă de timp, toată atenția a fost asupra creșterii frecvenței procesorului cu un singur nucleu. Până la un anumit punct a funcționat , însă creșterea frecvenței este direct proporționala cu consumul de energie electrică. Astfel, creșterea frecvenței nu a mai fost rentabilă , iar tehnica multi-core reprezintă principalul indicator de performanță.

2.3 În ce constă protocolul Snoopy?

Protocolul Snoopy este un mecanism folosit în sisteme multi-core pentru a asigura coerența cache-urilor. Atunci când mai multe nuclee au copii ale aceleași date în cache-urile lor,

protocolul Snoopy se asigură că toate copiile rămân consistente.De exemplu, în cazul în care 2 core-uri citesc aceleași date din memoria principala, iar una dorește modificarea lor, celălalt nucleu nu ar putea vedea noile valori ale datelor , folosindu-le pe cele vechi. Acest exemplu reprezintă unul clasic de tipul : incoerența datelor Datorită acestui protocol , în sistemele multi-core cu un număr limitat de nuclee , aceste nesincronizări dispar.

Între memoriile cache ale nucleelor și memoria principală va fi proiectată o magistrală de date comună, prin care fiecare nucleu poate executa una sau mai multe instrucțiuni în funcție de comenzile primite de la celelalte unități de calcul. Pe această magistrală se implementează protocolul Snoopy, care permite ca fiecare cache să observe activitatea celorlalte cache-uri și să reacționeze corespunzător, asigurând coerența datelor.

3. Planificare

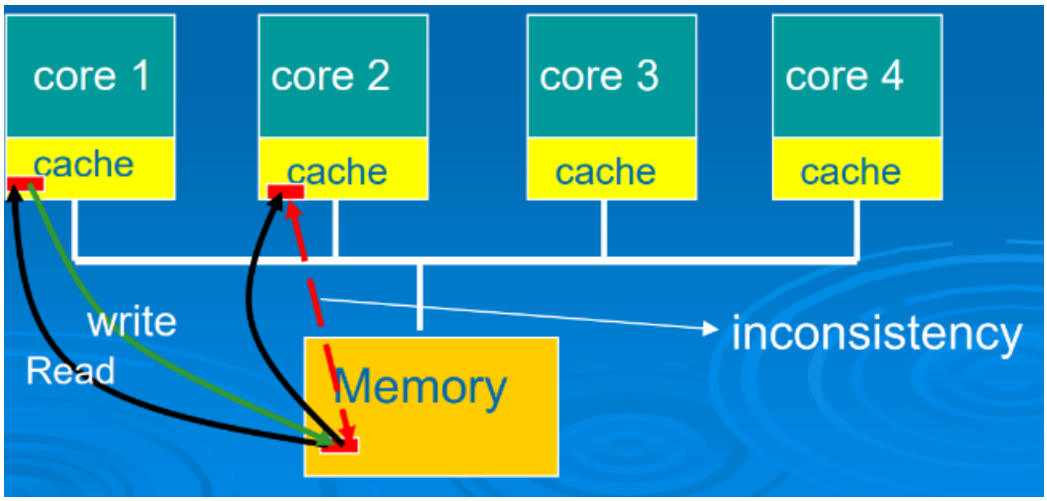
- Săptămâna 1 (29 septembrie) – Alegerea temei de proiect și planificare
- Săptămâna 3 (13 octombrie) – Studiu bibliografic , planificare si începerea documentației
- Săptămâna 5(27 octombrie) – Proiectarea și implementarea magistralei de date de tip FIFO, legarea procesoarelor de tip Mips single-cycle la magistrală și organizarea memoriei partajate
- Săptămâna 7(10 noiembrie) – Sinteză și implementare UC pentru Cache Controller-ului și organizarea memoriilor cache a core-urilor(procedura MSI)
- Săptămâna 9(24 noiembrie) - Interconectare componente, testare, implementare
- Săptămâna 11(8 decembrie) - Interconectare componente, testare, rezultate experimentale ,implementare
- Săptămâna 13(12 ianuarie) – Prezentare Proiect

4. Analiză

Etapa de analiză are ca Analiza reprezintă etapa în care se înțeleg și se definesc cerințele sistemului multi-core propus, definirea funcționalităților acestora și descrierea comportamentului unităților în timpul execuției. În proiectul de față, accentul este pus pe implementarea coerenței memoriei cache folosind protocolul Snoopy MSI, într-un context multi-core cu doi nucleu MIPS single-cycle.

4.1 Scopul proiectului

Proiectul final va putea realiza sincronizarea datelor folosite de către cele două core-uri. În plus , este posibilă tehnica de “write-back” , care are loc în timpul citirii unei date invalide. Aceasta se deosebește față de “write-through” prin faptul că nu scrie mereu în memoria cache de nivel doi , cea comună , ci doar când ambele nuclee detin aceleași date. Tehnica “wb” este mai avantajoasă , deoarece accesul la o memorie mai îndepărtată de procesor este mai rară , deci viteza de transfer a datelor crește.



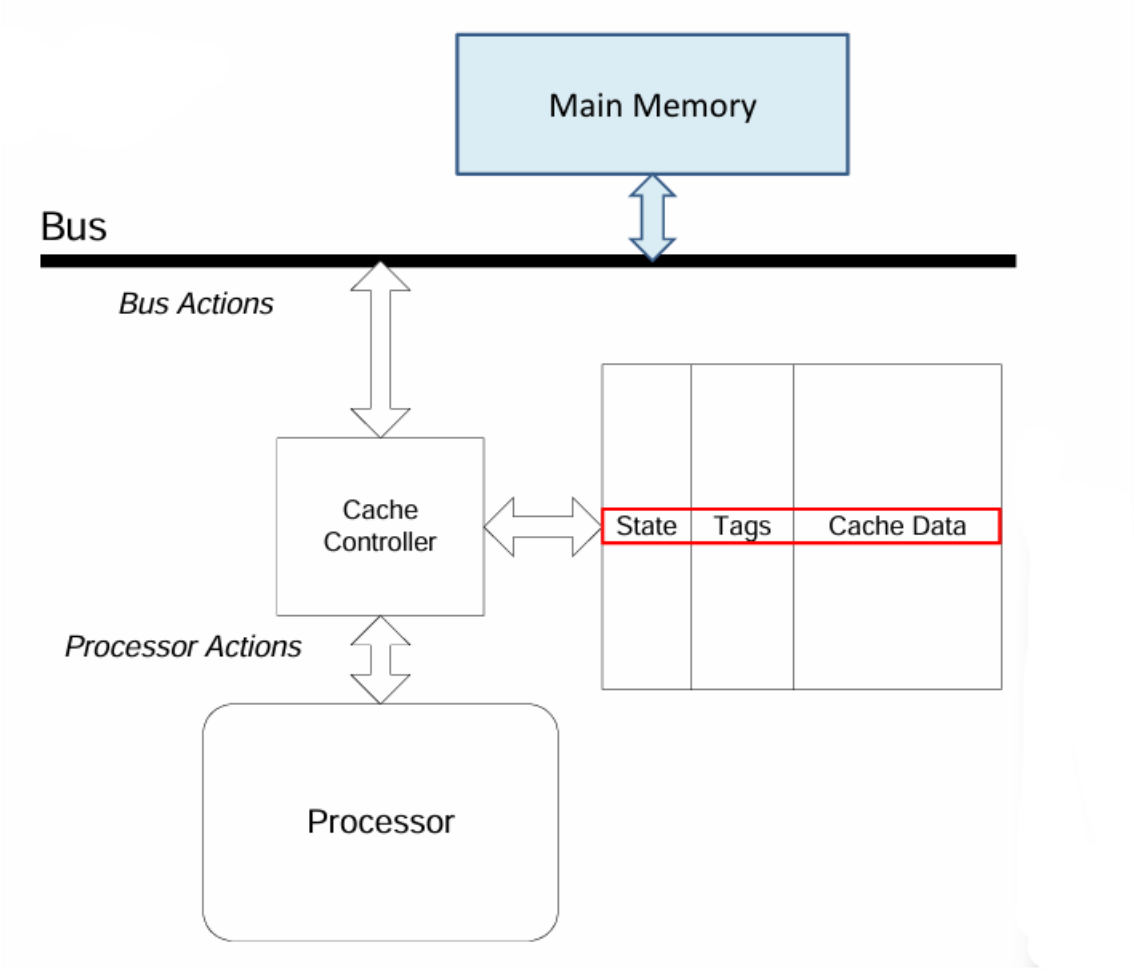
Figură 2: Schema unui procesor multicore si legătura cu memoria principala

4.2 Analiza detaliata a protocolului Snoopy

Acest protocol reprezintă soluția sincronizării datelor în cazul instrucțiunilor de citire sau de scriere, un real avantaj în procesarea paralelă ,însă complică arhitectura magistralei de date si a nucleelor de tipul “single cycle” și este nevoie de mai multe cicluri de ceas pentru întreg procesul de sincronizare.

„Ascultarea” de pe magistrală a protocolului este implementat printr-un tabel ,reprezentat ca o matrice în care sunt stocate toate variabilele din memoria cache a core-urilor. Aici sunt stocate: id-ul core-urlui , tipul instrucțiunii care a declanșat ascultarea ,

datele pentru identificarea adresei și variabila propriu-zisă. Datele trimise de nucleu ajung pe magistrală , unde protocolul le identifică și le trimite la acest tabel pentru a le vedea starea actuală și pentru a le-o modifica daca este cazul. Totodată , scrierea în memoria comună este condiționată de această unitate de analiză a adreselor.



Figură 3: Arhitectura completă a magistralei de date

În cazul în care instrucțiunea nu este una care interacționează cu memoria cache , atunci pe magistrala nu va fi trimis nimic. Față de imaginea de mai sus ,soluția propusă de mine vine cu o componentă în plus , inclusă în cache controller . În procesor nu am niciun fel de date despre starea actuala și este trimisă mai departe orice fel de instrucțiune de citire sau scriere.

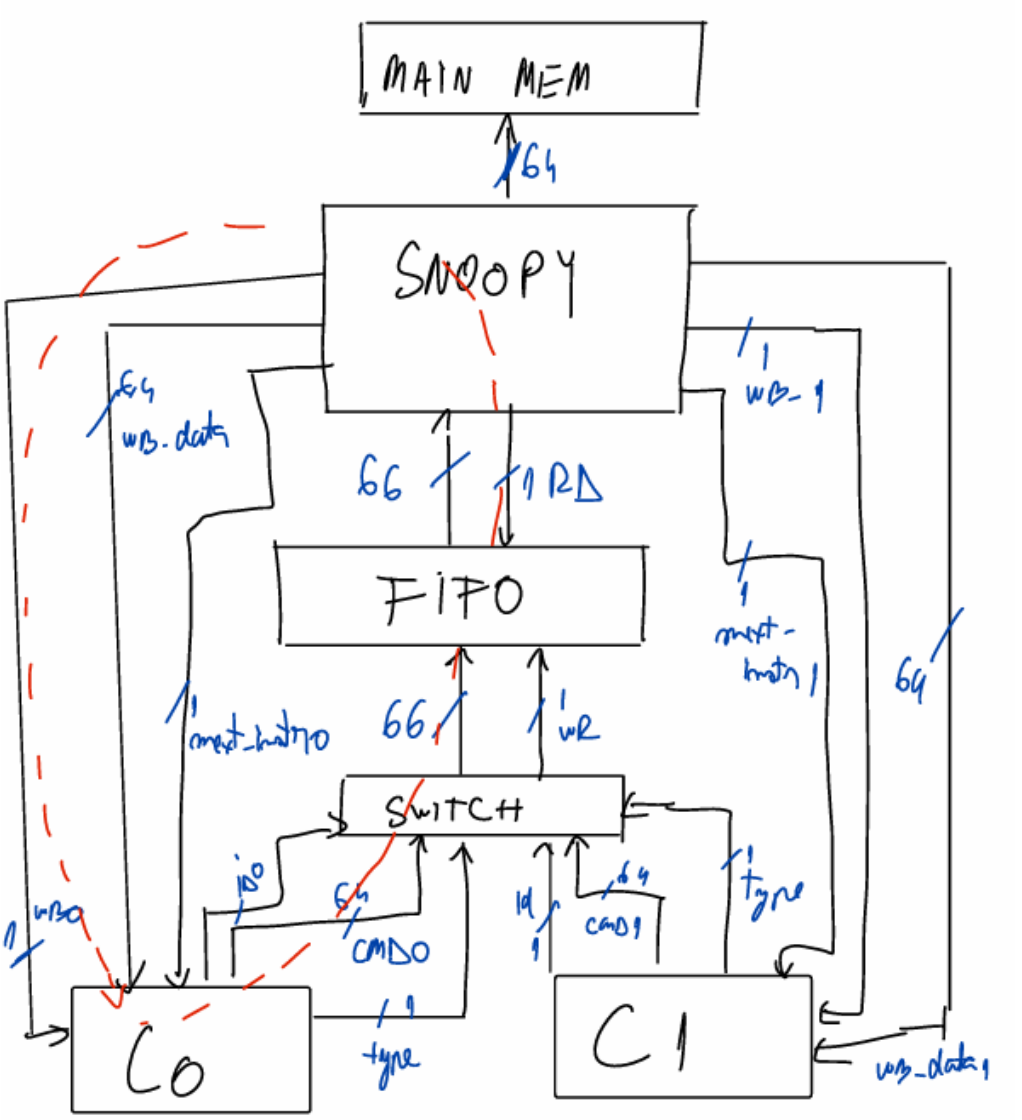
Protocolul Snoopy implementat este format din mai mulți pași:

1. Trimiterea instrucțiunii într-un switch.
2. Transferul mai departe pe magistrala(scrierea într-o coadă de tip FIFO).
3. Transferul în cache controller.

4. Transferul în componenta “getFull_line_state” , cu scopul de a extrage întreaga linie din tabel unde este salvată și starea curentă.
5. Trimiterea înapoi în cache controller a instrucțiunii cu starea actuală.
6. Transferul în tabel pentru scriere/citire a datelor și modificarea stărilor.
7. (în cazul la wb) Scrierea în memoria principală și transferul înapoi în memoria cache a core-ului.

Din această secvență putem deduce calea critică , care are loc la o scriere de tip write back.

4.3 Schema completă a proiectului



Figură 4: Schema completă

Notă: linia roșie reprezintă calea critică a arhitecturii.

5. Design

5.1 Design-ul memoriei

Memoria privată pentru fiecare core reprezintă memoria cache de nivel 1 (L1) , unde fiecare va stoca propriile date. Memoria comună reprezintă cache-ul de nivel 2 (L2) , de unde nucleele vor citi date , le vor prelucra si după le vor scrie înapoi în caz de write back.

Legătura dintre cele două tipuri de memorii este realizată cu ajutorul tehnicii Direct Mapped Cache , unde fiecare linie de adresă , pe lângă variabilă , conține și un câmp de tag , unul de index , iar altul de offset. În total vor fi 64 de biți , 32 de biți pentru date și 32 pentru codificarea adreselor.

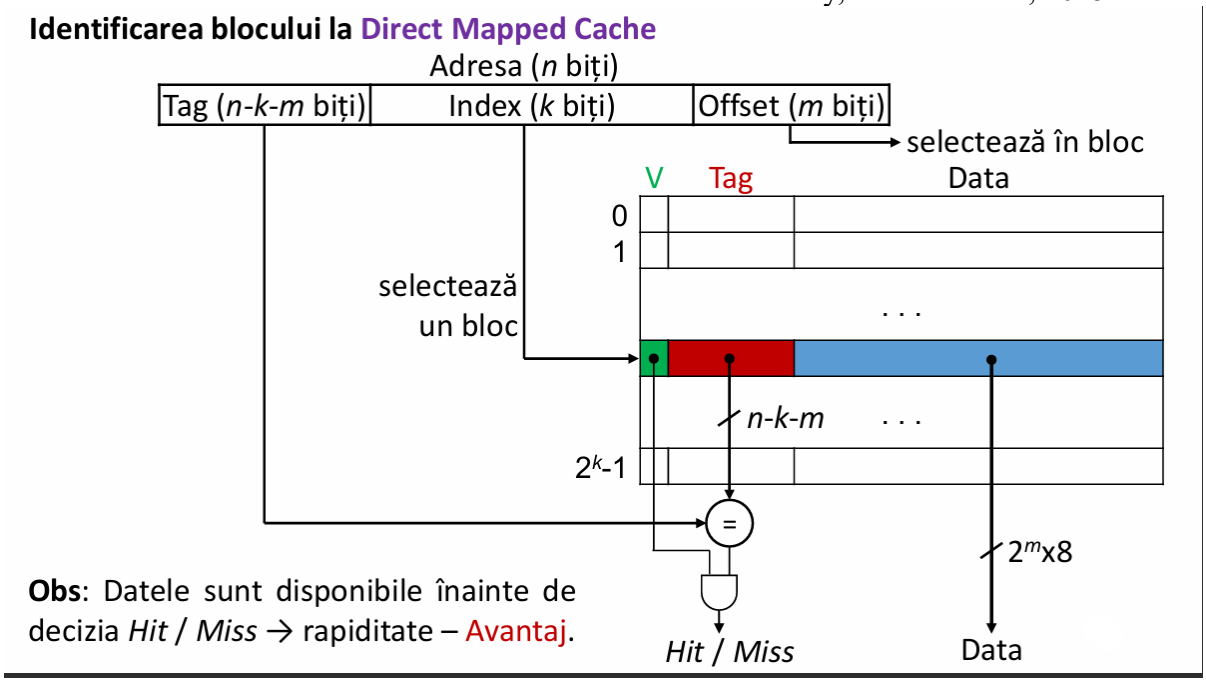
Organizarea adresei este:

- Data : 32 de biți – datele efective
- Offset : 4 biți – pentru selectarea cuvântului din blocul de date
- Index : 6 biți – pentru selectarea liniei din memorie
- Tag : 22 de biți (32 – 4 – 6) – pentru verificarea originii datelor

5.1.1 Memoria partajată (principală)

Pe lângă cei 64 de biți , fiecare linie de memorie are în plus încă un bit , pe poziție de MSB , care reprezintă validitatea linie. În cazul în care sunt transmise date de la cache controller , iar linia este validă , este actualizată valoarea de la linia respectivă , de la cuvântul respectiv.

O linie poate stoca 16 cuvinte , fiecare 32 de biți(512) , 32 de biți pentru adresa si 1 bit pentru validitate , un total de 545 de biți , cu 64 de linii.



Figură 5: Schema tehnicii Direct Mapped Cache

5.2.2 Memoria privată(cache L1)

Memoria cache este formata din cei 64 de biți, cu un total de 64 de linii . Ea poate primi ca intrare de date rezultatul ce provine de la unitatea aritmetică logică sau de la cache controller , în cazul unui write back. Intern , este calculată linia unde trebuie să se suprascrie datele.

5.2.3 Tabelul de stări

Acest tabel reprezintă tot un tip de memorie , deoarece este o matrice care ascultă și scrie fiecare mesaj trimis pe magistrală. Datele parcurg această componentă de 2 ori:

1. Prima dată când se caută starea actuală pentru a știi protocolul snoopy ce sa facă , în ce caz se află.
2. Când protocolul trimite cereri pentru modificarea stărilor și a datelor stocate dacă este cazul.

5.3 Cache Controller (Unitatea de Control)

O variabilă de date este considerată invalidă atunci când unul dintre nucleele procesorului rescrie valoarea acelei date în propria memorie cache, iar restul core-urilor încă dețin valoarea veche. În această situație, cache controller-ul trimite o cerere pe magistrala de date pentru a notifica celelalte core-uri despre modificare. Conform protocolului Snoopy, scrierea efectivă în memoria partajată are loc doar la write-back, adică atunci când linia de cache cu valoarea modificată trebuie scrisă în memoria principală pentru a sincroniza datele. Restul nucleelor își actualizează starea liniei de cache corespunzător (Invalid sau Shared), în funcție de tipul accesului.

Pentru gestionarea acestui proces, este folosită tehnica MSI, care definește trei stări principale:

- M – Modified
- S – Shared
- I – Invalid

Avantajul protocolului MSI este simplitatea și eficiența sa în sisteme cu un număr redus de core-uri. Totuși, un dezavantaj apare în situațiile în care o linie de memorie partajată este prezentă doar într-un nucleu; în acest caz, linia trece mai întâi în starea S, iar ulterior, după ce se modifică și memoria principală, ajunge în starea M, necesitând două tranziții de stare. Această problemă poate fi optimizată prin introducerea unei stări suplimentare Exclusive (E), rezultând protocolul MESI.

a. Starea Modified (M)

Această stare indică faptul că în memoria cache a nucleului curent există cea mai recentă valoare a variabilei, obținută după scriere. În restul core-urilor, linia de cache se află în starea Invalid (I). După write-back, linia de cache a nucleului cititor va fi marcată ca Shared (S), iar nucleul care a făcut write-back își poate păstra sau actualiza starea în funcție de acces.

b. Starea Shared (S)

Starea S indică faptul că valoarea variabilei este identică în mai multe memorii cache și în memoria partajată. Dacă două sau mai multe nuclee împart aceeași linie de date, toate vor avea starea Shared (S) pentru acea linie. În cazul în care unul dintre ele încearcă să scrie, toate stările de S sunt eliminate și se aplică tranziția către M pentru nucleul care scrie, iar celelalte devin I.

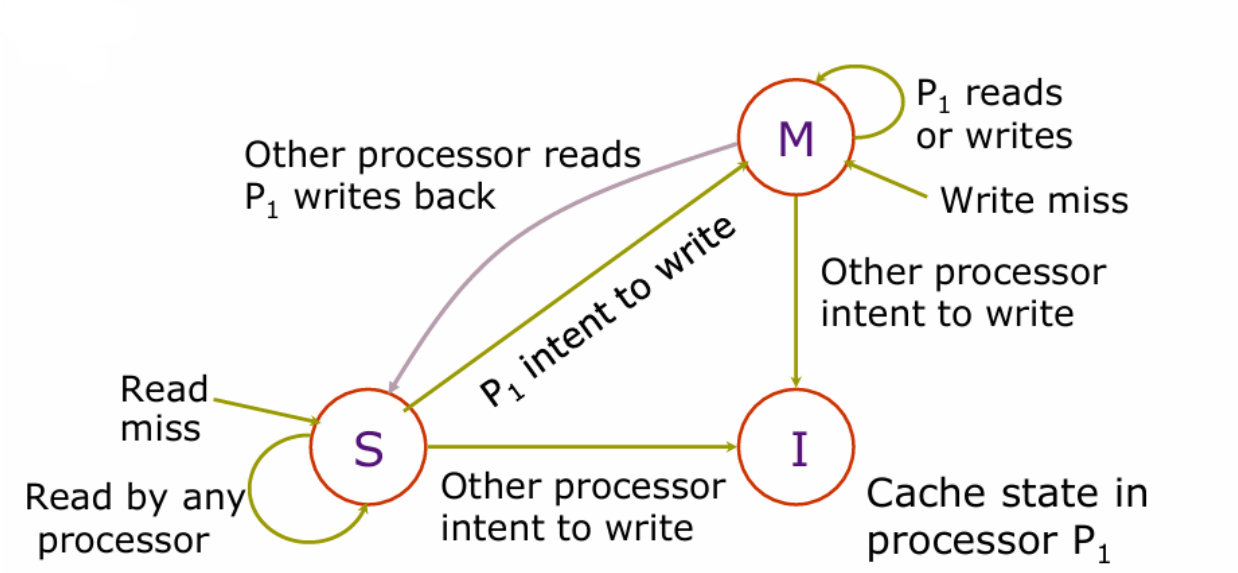
c. Starea Invalidate (I)

Această stare semnalează faptul că valoarea din memoria cache a nucleului nu este validă și nu poate fi utilizată pentru operații. Ea apare în urma unei cereri de remote read sau remote write, atunci când alt core trimite o cerere de citire sau scriere pe magistrala de date. Linia este marcată

ca invalidă până când nucleul respectiv primește o copie actualizată sau efectuează un nou acces. Scrierea în memoria principală are loc doar în acest caz , când valoarea de la adresa

	Action and Next State						
Current State	*Processor Read*	*Processor Write*	*Eviction*		*Cache Read*	*Cache Read&M*	*Cache Upgrade*
I	*Cache Read* Acquire Copy → S	*Cache Read&M* Acquire Copy → M			No Action → I	No Action → I	No Action → I
S	No Action → S	*Cache Upgrade* → M	No Action → I		No Action → S	*Invalidate Frame* → I	*Invalidate Frame* → I
M	No Action → M	No Action → M	*Cache Write back* → I		Memory inhibit; Supply data; → S	*Invalidate Frame;* Memory inhibit; Supply data; → I	

Figură 6: Tabel de tranziții a stărilor



Figură 7: Diagrama de stări a cache controller-ului

Friday, November 21, 2025

5.4 Nucleul de tip Mips Single-Cycle

Ca nucleu al procesorului este utilizată arhitectura MIPS cu ciclu unic, proiect realizat anterior în cadrul disciplinei *Arhitectura Calculatoarelor*. Acest tip de procesor are la bază un model de execuție simplificat, în care fiecare instrucțiune este complet procesată într-un singur ciclu de ceas. Spre deosebire de arhitecturile pipelined, în care etapele instrucțiunilor se suprapun, nucleul MIPS single-cycle parcurge toate fazele de execuție — de la preluarea instrucțiunii până la scrierea rezultatului — în mod secvențial, dar în cadrul aceluiași impuls de ceas.

Arhitectura este compusă din cinci blocuri funcționale principale: Instruction Fetch (IF), Register File (REGFILE), Unitatea Aritmetică și Logică (ALU), Unitatea de Control (UC) și Memoria de Date (MEM). Blocul IF are rolul de a prelua instrucțiunea din memoria de instrucțiuni pe baza adresei din registrul Program Counter (PC). Unitatea UC decodifică instrucțiunea și generează semnalele de control necesare celorlalte componente. REGFILE furnizează operanzii necesari execuției, iar ALU efectuează operațiile aritmetice și logice specificate. În cazul instrucțiunilor de tip load/store, blocul MEM asigură accesul la memoria de date. Toate aceste operații sunt realizate în cadrul unui singur ciclu de ceas, ceea ce conferă arhitecturii un comportament predictibil și o implementare hardware relativ simplă, deși cu un timp de ciclu mai lung comparativ cu arhitecturile pipelined.

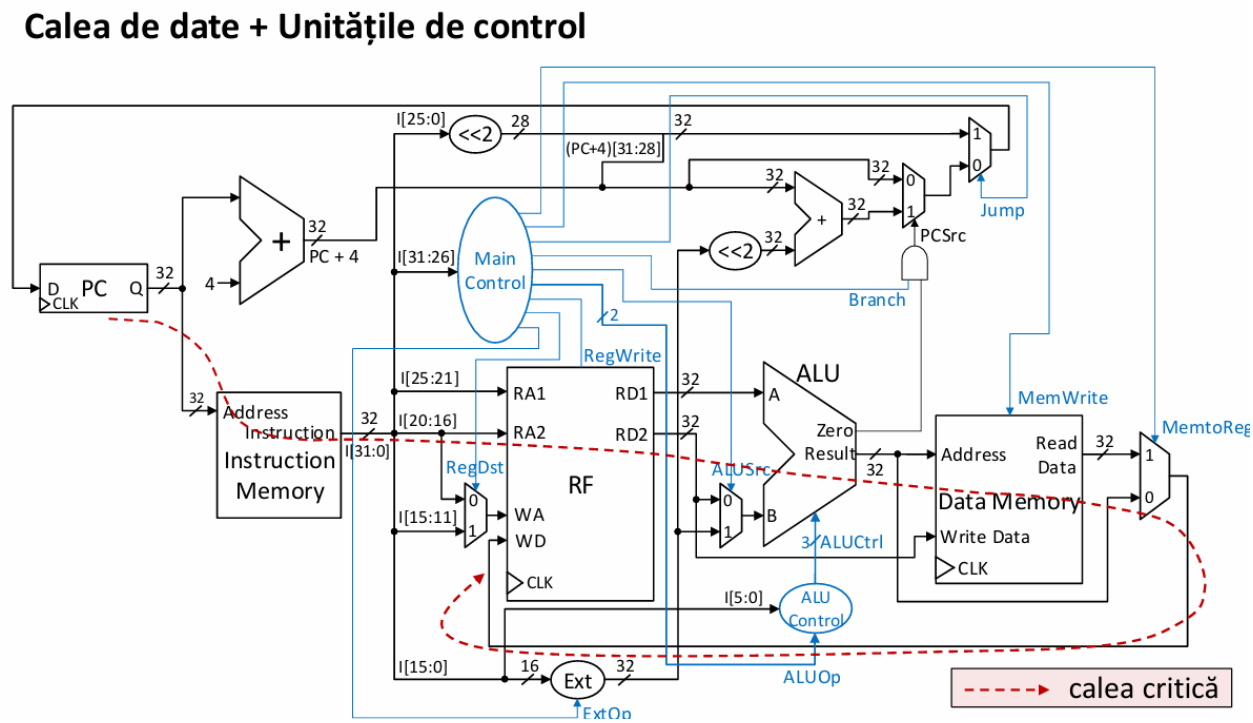
Totuși , singura modificare esențială care va exista în această componentă deja existentă este modificarea mărimii câmpului de adrese din memoria cache , pentru a implementa protocolul Snoopy , cu tehnica MSI.

Semnificații: & - AND, | - OR, ^ - XOR, l - logic, a - aritmetic, v - cu variabila

	Instrucțiune	Opcode *Instr[31-26]*	Reg Dst	ExtOp	ALUSrc	Branch	BGTZ	Jump	BNE	MemWrite	MemtoReg	RegWrite	ALUOp[1:0]	function *Instr[5-0]*	ALUCtrl[2:0]
1.	ADD	000000	1	X	0	0	0	0	0	0	0	1	CodR	000001	+ 001
2.	SUB	000000	1	X	0	0	0	0	0	0	0	1	CodR	000011	- 010
3.	SLL	000000	1	X	0	0	0	0	0	0	0	1	CodR	000100	<< 011
4.	SRL	000000	1	X	0	0	0	0	0	0	0	1	CodR	000101	>> 100
5.	AND	000000	1	X	0	0	0	0	0	0	0	1	CodR	000110	AND 101
6.	OR	000000	1	X	0	0	0	0	0	0	0	1	CodR	000111	OR 110
7.	XOR	000000	1	X	0	0	0	0	0	0	0	1	CodR	001000	XOR 000
8	SLT	000000	1	X	0	0	0	0	0	0	0	1	CodR	001101	< 111
9.	ADDi	100000	0	1	1	0	0	0	0	0	0	1	Cod+	X	+ 001
10.	LW	100001	0	1	1	0	0	0	0	0	1	1	Cod+	X	+ 001
11.	SW	100010	0	1	1	0	0	0	0	1	X	0	Cod+	X	+ 001
12.	BEQ	100100	X	1	0	1	0	0	0	0	X	0	Cod-	X	- 010
13.	BGTZ	100101	X	1	0	0	1	0	0	0	X	0	Cod-	X	- 010
14.	BNE	100011	X	1	0	0	0	0	1	0	X	0	Cod-	X	- 010
15.	Jump	111110	X	X	X	X	0	1	0	0	X	0	XX	X	XXX

Figură 8:Codificarea Intrucțiunilor

Mai jos este prezentată întreaga schemă a unui nucleu , un mips single-cycle , fără modificările implementate de mine pentru acest proiect.



Figură 9: Arhitectura procesorului Mips Single Cycle

6. Implementare

În această parte vor fi descrise în detaliu componentele implementate . În cazul nucleelor , descrierea arhitecturală este aceeași în proporție mare , fiind și menționând doar modificările sau adăugările făcute în unitatea de control și memoria cache privată.

Codul VHDL ce descrie principalele componente ale unității aritmetice și logice poate fi consultat în cele ce urmează.

Friday, November 21, 2025

6.1 Mips single-cycle

În implementarea propusă , singurele modificări au loc în unitatea de control, memoria cache, instruction fetch și memoria de tip ROM, unde nu mai este rulat același program , ci este unul simplu , cu scopul e a testa toate cazurile posibile pentru protocolul Snoopy.

Fiecare core are aceeași implementare , dar în proiect codul este duplicat pentru a putea exemplifica procesarea paralelă a core-urilor.

Fiecare componentă din procesor , IF,ID,EX,MEM,WB are aceeași funcționalitate.

6.1.1 Unitatea de control

```
ENTITY UC IS
  PORT (INSTR: IN STD_LOGIC_VECTOR(5 DOWNTO 0);

REGDST,EXTOP,ALUSRC,BRANCH,JUMP,BGTZ,BNE,MEMWRITE,MEMTOREG,
REGWRITE: OUT STD_LOGIC;
      READWRITECC,LW_SWINSTR : OUT STD_LOGIC;
      ALUOP : OUT STD_LOGIC_VECTOR(1 DOWNTO 0));
END UC;
```

Semnale adăugate sunt :

- lw_swInstr : un semnal de ieșire ,care este trimis la memoria privată , cu scopul de a atenționa memoria că este o instrucțiune de scriere sau de citire. În caz afirmativ , este permisă transferul mai departe de adresă la cache controller. Acest semnal este ‘1’ în acest caz.
- readWriteCC : un semnal de ieșire, cu scopul de a transmite la memorie și mai departe, tipul de instrucțiune . Acest semnal este ‘1’ când are loc sw (scriere) și 0 când are loc lw(citire).

6.1.2 Instruction Fetch

```
entity IFetch0 is
  Port (rst,clk,en,jump, PcSrc,next_instr_core0 : in std_logic;
        jumpAddress, branchAddress: in std_logic_vector( 31 downto 0);
        instruction, pc4 : out std_logic_vector( 31 downto 0) );
end IFetch0;

if en ='1' and next_instr_core0 = '1' then
```

Singura modificare adusă la această componentă o reprezintă adăugarea semnalului “instr_core0” sau “instr_core1” pentru celălalt core.Este un semnal de intrare , care provine de la cache controller . Când tabelul returnează adresa cu stările si datele modificate , semnifică faptul că instrucțiunea s-a terminat de procesat și poate fi procesată următoarea din memoria Rom a procesorului . Acest semnal completează condiția pusă de “en” pentru următoarea instrucțiune.

6.1.2 Memoria internă

```
entity Mem0 is
  Port (memWrite,En,clk,wb,readWriteCC,lw_swInstr: in std_logic;
        AluRes,Rd2: in std_logic_vector(31 downto 0);
        data_fromCC : in std_logic_vector(63 downto 0);
        send_data_to_bus,line_debug : out std_logic_vector(63 downto 0) ;
        useCC: out std_logic;
        memData,aluResOut: out std_logic_vector(31 downto 0));
end Mem0;

signal line_indexCc, address, address_aux : integer range 0 to 63 := 0;
signal found : std_logic :='0';
signal alu_res_aux : std_logic_vector(5 downto 0);

signal ucc_aux : std_logic :='0';
begin

alu_res_aux <= aluRes(7 downto 2);

address_aux <= to_integer(unsigned(AluRes(7 downto 2))) mod 64;
```

În componenta memoriei interne sunt adăugate următoarele semnale de control:

- Wb : semnal de intrare , care provine de la cache controller , în cazul unei scrieri de tip write back. “Data_fromCC” reprezintă adresa noua suprascrisă din memoria principală. De menționat faptul că tag, index, offsest rămân aceleași , iar căutarea în memorie are loc după acestea.
- Found : un flag care marchează găsirea liniei pentru write back.
- useCC : semnal de ieșire , care este transmis mai departe la switch ca să marcheze faptul că nucleul dorește să folosească cache controller-ul.
- Send_data_tobus: semnal de ieșire . Transferă întreaga linie unde are loc un lw sau sw.Semnalul line_debug este același , folosit pentru debug.

```
find_line: process(clk)
      begin
      if rising_edge(clk) then
         if wb = '1' then
          for i in 0 to 63 loop
         if m(i) = data_fromCC(63 downto 32) and (not
(data_fromCC = X"0000000000000000")) then
            line_indexCc <= i;
            exit;
        end if;
      end loop;
      end if;
      end if;
      end process;
address <= address_aux when wb = '0' else line_indexCc;
```

Proces cu scopul de a căuta index-ul linie care a trimis înainte adresa în cache controller.

Ultima asignare , cea din afara procesului , are ca scop fixarea adresei din memorie . Dacă are loc write back , adresa este cea rezultată în urma căutării , altfel este cea calculată în unitatea de execuție.

În descrierea hardware prezentată mai jos ,este procesul care se ocupă cu transmiterea datelor către switch. La fiecare ciclu de ceas , ucc_aux este 0 pentru a nu influența starea viitoare. Primul if verifică cazul de write back și scrie în memorie , următorul verifică dacă are loc o scriere. În acest caz, adresa este transmisă pe magistrală , dar și scrisă în propria memorie , deoarece core-ul este sigur ca acea valoare este cea mai nouă ,cel puțin la acea perioadă de timp. Ultima parte din if verifică cazul instrucțiunii de citire și returnează din memorie prin semnalul memData , valoarea citită . Presupun că în acest caz , de citire ,daca citește și este în starea de modified sau shared , atunci rămâne valoarea din memorie , iar în cazul de invalid are loc write back.

```
process(clk)
begin
   if rising_edge(clk) then
      line_debug <= m(address);
      ucc_aux <= '0';

      if wb = '1' then
         m(address) <= data_fromCC;
         memData <= data_fromCC(31 downto 0);

      elsif en = '1' and readWriteCC = '1' and
memWrite='1' and lw_swInstr='1' then
         ucc_aux <= '1';
         send_data_to_bus <= m(address)(63
downto 32) & Rd2;
         m(address)(31 downto 0) <= Rd2;
         memData <= Rd2;

      elsif en = '1' and readWriteCC = '0' and
lw_swInstr='1' then
         ucc_aux <= '1';
         send_data_to_bus <= m(address);
         memData <= m(address)(31 downto 0);

      end if;
   end if;
end process;
```

6.2 Switch (Scheduler)

Rolul acestei componente este de a selecta pe rând câte o instrucțiune , în cazul în care ambele core-uri trimit în același ciclu de ceas cereri către cache controller.

```
  Port (
       id_0, id_1, readWrite_type0, readWrite_type1 : in std_logic;
       data_in0, data_in1 : in std_logic_vector(63 downto 0);
       useCC0, useCC1, clk : in std_logic;
       wr_fifo : out std_logic;
       data_out_toCC : out std_logic_vector(65 downto 0)
  );
    signal turn : std_logic := '0';
    signal data_out_aux : std_logic_vector(65 downto 0);
    signal last_sent : std_logic_vector(65 downto 0) := (others => '0');
    signal candidate : std_logic_vector(65 downto 0);

process(clk)
    begin
       if rising_edge(clk) then
          wr_fifo <= '0';
          if useCC0 = '1' and useCC1 = '1' then
             if turn = '0' then
                candidate <= id_0 & readWrite_type0 & data_in0;
                turn <= '1';
             else
                candidate <= id_1 & readWrite_type1 & data_in1;
                turn <= '0';
             end if;
          elsif useCC0 = '1' then
             candidate <= id_0 & readWrite_type0 & data_in0;
             turn <= '1';
          elsif useCC1 = '1' then
             candidate <= id_1 & readWrite_type1 & data_in1;
             turn <= '0';
          else
             candidate <= (others => '0');
          end if;

          if candidate /= last_sent and candidate /= X"0000000000000000" & "00" then
             data_out_aux <= candidate;
             wr_fifo <= '1';
```

- id_0, id_1, readWrite_type0, readWrite_type1 : semnale de intrare , care precizează id-ul fiecărei instrucțiuni trimise , de la fiecare core, cu scopul de a le concatena și păstra informația în această linie.
- data_in0, data_in1 : adresele care sunt transferate de la fiecare nucleu
- useCC0, useCC1 : semnalele ca selecție trimise de fiecare core , în cazul unei instrucțiuni sw sau lw.
- wr_fifo : semnal care are rolul de a anunța coada de adrese că urmează să scrie ce îi va fi transmis de către switch.
- data_out_toCC : semnal de ieșire a componentei , cu adresa aleasă . Este pe 66 de biți , MSB este id-ul , iar MSB - 1 este tipul instrucțiunii.

Rolul semnalului auxiliar “turn” este acela de a fi folosit în cazul în care core-urile trimit deodată date. Este implementată această selecție pe modelul tehnicii “Round Robin ”, deoarece selecția adresei este aliatoare , iar la următorul ciclu de ceas este transmisă cealaltă adresă ,care nu a fost transferată inițial.

Semnalele “last_sent” și “candidate” au rolul de a evita scrierii în FIFO aceleiași adrese. Primul semnal stochează semnalul atribuit la momentul de timp T-1 , candidatul trecut , iar al doilea reprezintă candidatul de la acest moment de timp T. Astfel, se verifică ca cele 2 semnale să nu fie egale și diferite de 0 . Presupun că adresa care conține doar ‘0’ nu există sau este rezervată și nu poate fi accesată.

Obs:

Este folosit principiul descrierii hardware din VHDL , acela că valoarea asignată într-un proces , a unui semnal ,este vizibilă , poate fi folosită abia următorul ciclu de ceas.

6.3 Fifo_connect_mux_cc

Componenta FIFO are rolul de a stoca toate cererile selectate de către switch și a le scrie la fiecare semnal de intrare “wr”, provenit tot de la switch . Implementarea este identică cu cea din laboratorul 4 , unde există semnalul de wr_ptr și de rd_ptr, data_out , care este valoarea citită de la “rd_ptr”. În cazul în care nu are loc citire, este returnat un semnal de înaltă impedanță ,folosindu-se un 3-state-buffer.

```
    component decoder_5to32 is
    Port (
       wr_ptr     : in  std_logic_vector(4 downto 0);
       decode_out : out std_logic_vector(31 downto 0)
    );
  end component;

gen_fifos2: for i in 0 to 31 generate
    fifo_inst1: fifo_component_2ids
       port map(
          wr       => wr,
          rst      => rst,
          clk      => clk,
          data_in =>  data_in,
          fifo_out => M(i),
          decode_out => decode_out(i)
       );
    end generate gen_fifos2;
```

De asemenea , este folosită componenta internă prezentată mai sus , pentru a stoca adresa la index-ul respectiv.

Alte semnale descrise în această componentă sunt:

- wr_inc – semnal ce crește pointerul de scriere wr_ptr. Acționează doar dacă FIFO-ul nu este plin.
- rd_inc – semnal ce crește pointerul de citire rd_ptr. Acționează doar dacă FIFO-ul nu este gol.
- new_fifo – semnal auxiliar care indică momentul în care FIFO a furnizat o nouă intrare la ieșire (data_out).Devine ‘1’ doar în ciclul de ceas în care a fost efectuată citirea.
- count_aux – contor care reține câte elemente valide există în FIFO. Ajută la generarea semnalelor full și empty.

6.4 Memoria Principala

Această memorie este actualizată doar în momentul unui write-back, atunci când un cache deține o linie în starea Modified (M) și un alt core solicită acea linie, fiind invalid Astfel, datele care ajung la această memorie provin din cache-ul unui core, ca rezultat al unei invalidări, al unei citiri remote sau al unei tranziții de stare impusă de protocolul Snoopy.

```
entity MainMem is
  Port (dataIn : in std_logic_vector(63 downto 0);
        write_enMain : in std_logic;
        clk : in std_logic;
        send_data_to_CCback: out std_logic_vector(63 downto 0));
end MainMem;
signal index  : integer := 0;
signal offset : integer range 0 to 15 := 0;
signal aux : std_logic_vector(63 downto 0) :=(others =>'0');
begin
proc_write: process(clk)
          variable start_bit : integer ;
          variable end_bit: integer ;
          begin
             if rising_edge(clk) then
               if write_enMain = '1' then
                    index  <= to_integer(unsigned(dataIn(41 downto 36)));
                    offset <= to_integer(unsigned(dataIn(35 downto 32)));
                    if M(index)(544) = '1' then
                    start_bit := offset * 32;
                    end_bit   := start_bit + 31;
                    M(index)(end_bit downto start_bit) <= dataIn(31 downto 0);
                    aux <= dataIn;
send_data_to_CCback <= aux;
```

Alte semnale descrise în această componentă sunt:

- dataIn (63:0): reprezintă datele primite din partea cache controller-ului ca rezultat al unei operații Snoopy.
- write_enMain : semnal de control care indică faptul că memoria trebuie actualizată. Activat doar în cazul în care Snoopy a declanșat un write-back.
- send_data_to_CCback: reprezintă cuvântul scris în memoria principală. Este returnat către cache controller pentru confirmarea operației sau pentru reintroducerea datelor în alt cache.

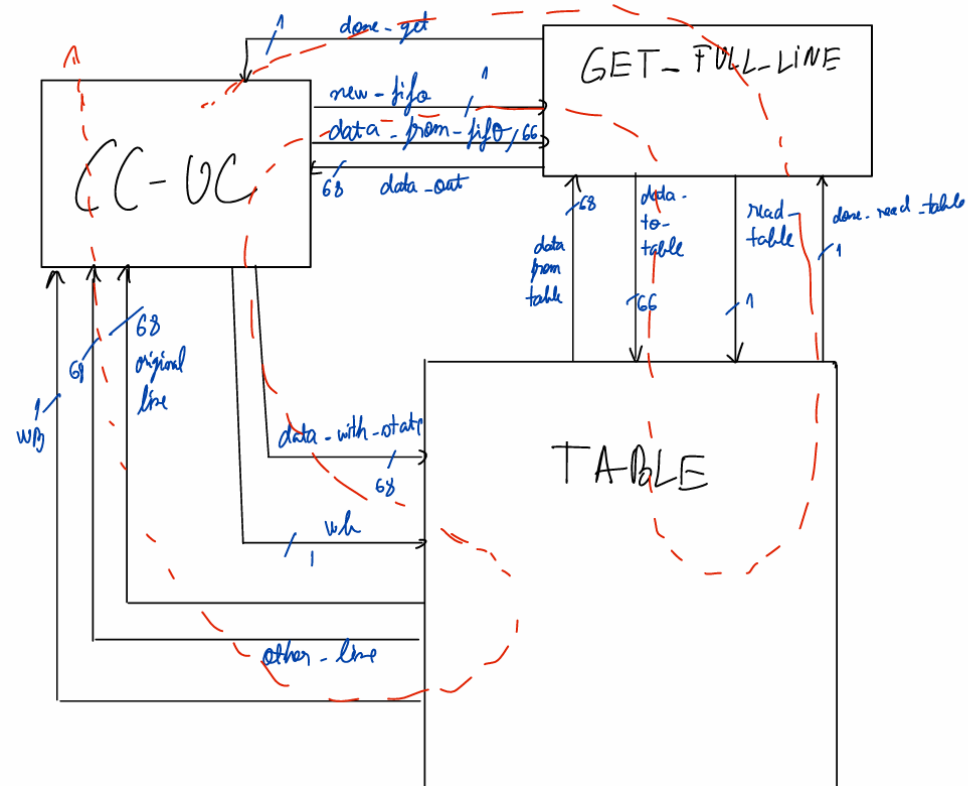
În plus , la activarea acestei componente , este extras index-ul și offset-ul din adresă și suprascrisă valoarea , la adresa exactă , dată prin formula :

```
start_bit := offset * 32;
end_bit := start_bit + 31;
M(index)(end_bit downto start_bit) <= dataIn(31 downto 0);
```

6.5 Implementarea Protocolului Snoopy

Uninatea de control pentru cache controller este implementat conform *figurii 6,* fiind respectată tranziția dintre stări. Totodată , această componente este compusă din alte 3 componente , care asigură funcționalitatea propusă. Acestea sunt:

- Cache Controller – face legătura dintre magistrala(FIFO) și unitatea de control ,reprezentând punctul de intrare în aceasta.
- Get_fullLine_with_state – primește linia de adresă din cache-ul privat al nucleului , o trimite la tabel și returnează linia de adresă cu starea curentă.
- Tabelul de stări – componentă în care sunt stocate toate adresele din cache-urile private, fiind un total de 128 de linii de adresă , și actualizează stările și datele.



Figură 10 : Schema Unității de control cu toate componentele

Notă: Linia punctată roșie reprezintă Data-Flow-ul componentei , pașii de execuție, alături de semnalele de intrare și ieșire.

6.5.1 Cache Controller (UC)

Această componentă reprezintă punctul de legătură dintre datele trimise de către nuclee și memoria principală . Totodată , este partea principală a proiectului , scopul fiind implementarea acestei unități de control.

Este cea care spune magistralei când să citească date, cea care transmite datele mai departe către componenta „get_fullLine” , către tabelul de stări sau către memoria principală. Totodată se ocupă de suprascrierea în memoria privată a core-urilor, la activarea semnalului de write back.

```
  Port( data_inFIFO : in std_logic_vector(65 downto 0);
        start : in std_logic;
        data_toCore0,data_toCore1:out std_logic_vector(63 downto 0);
        data_in_fifo_debug,DATA_FROMCC_TOTABLE_GET : out std_logic_vector(65 downto
0);
        data_fromTable_debug,data_in_fromCC_debug,data_inFIFOFromTable_debug : out
std_logic_vector(67 downto 0); --67 , scriu in daca trb ; id 1 bit , read/write type 1 bit , state
2 biti , tag 22 , index 6 , offset 4 , data 32 biti
        clk,new_fifo,lw_str_core0,lw_str_core1: in std_logic;
        original_line_debug,other_line_debug : out std_logic_vector(67 downto 0);
        wb_toCore0, wb_toCore1,
DONE,modify_state_out,DONE_GET_OUT,NEW_FIFO_OUT,WB_FOR_WB : out std_logic;
        write_enMain,next_instr_core0, next_instr_core1,rd_fifo  : out std_logic;
        line_toMain : out std_logic_vector(63 downto 0)
        );

signal next_state , current_state : MSI_STATE ;
signal data_toCore_aux,data_in_fromCC_debug_aux : std_logic_vector(67 downto 0)
:=(others =>'0');
signal wb_aux,wb_table,done_aux : std_logic :='0';
signal data_toTable,data_fromTable : std_logic_vector(67 downto 0) :=(others =>'0');

signal data_fromTable_toGet,data_out_Get : std_logic_vector(67 downto 0) :=(others
=>'0');
signal data_toTable_fromGet  : std_logic_vector(65 downto 0) :=(others =>'0');
signal done_read_table,modify_state,done_get,read_table_aux,continue_FSM : std_logic
:='0';
signal latched_line   : std_logic_vector(67 downto 0) := (others => '0');
signal latched_valid  : std_logic := '0';
```

Partea de descriere hardware a unității de control cuprinde partea de Port , unde sunt toate semnalele declarate:

- Start: reprezintă un semnal extern , de intrare .Este activ la început pentru “a porni” citirea din coadă. (am încercat să reprezinte o simulare din viața reală , ca atunci când este pornit un calculator. Totodată , citirea este făcută în mod automat, când o instrucțiune a fost procesată total de către tabel , însă la prima iterație nu a fost procesat nimic înainte. Acesta este motivul pentru care am introdus acest semnal).
- data_toCore0,data_toCore1: semnale de ieșire pentru suprascrierea valorii din memoria privată, provenite din tabel.
- data_in_fromCC : semnalul cu stare , de 68 de biți , transmis de UC către tabel ,după ce a revenit din căutarea din tabel.
- new_fifo : semnal de intrare , cu scopul de a semnala faptul că o nouă instrucțiune a fost procesată.
- lw_str_core0,lw_str_core1, _instr_core0, next_instr_core1: primele 2 semnale provin din unitatea de control din nuclee și au ca scop transmiterea mai departe a nucleelor să treacă la următoarea instrucțiune ,deoarece instrucțiunea de tip lw sau sw a fost terminată de procesat.În plus , daca nu este o instrucțiune de acest tip , adică *lw_str_core0 = ‘0’,* atunci va transmite ‘1’ pentru a trece la instrucțiunea următoare.
- wb_toCore0, wb_toCore1: semnalele de ieșire cu scopul de a transmite faptul că a avut loc un write back la core-ul care a transmis instrucțiunea.
- modify_state_out: semnal de ieșire cu scopul de a transmite tabelului faptul că în UC s-a identificat starea în care se află și trebuie modificată starea din tabel , împreună cu datele.
- rd_fifo: semnal pentru magistrală , pentru a citi cererile , câte una la fiecare perioada de ceas, exceptând prima perioadă din simulare , unde este nevoie de semnalul ‚start’.
- write_enMain : activare write în memoria partajată.
- line_toMain : valoarea variabilei ce trebuie scrisă în memorie. Aceeași valoare este partajată de ambele core-uri la acel moment de timp.
- “type MSI_state is (S, M , I);” reprezintă definirea stărilor din automatul de stări finite.

```
signal latched_line   : std_logic_vector(67 downto 0) := (others => '0');
signal latched_valid  : std_logic := '0';

process(clk)
begin
  if rising_edge(clk) then
    if done_get = '1' then
      latched_line  <= data_out_Get;
      latched_valid <= '1';
    elsif (done_aux = '1' and modify_state = '1') then
      latched_valid <= '0';
    end if;
  end if;
end process;
```

Semnalele ‘latched’ sunt adăugate în arhitectură, deoarece în timpul simulării am observat comportamente neobișnuite a FSM. Ele au ca scop întârzierea semnalelor venite din componenta “get_fullLine” cu o perioadă de ceas , pentru a putea facilita transformarea de stări din automat. Altfel, automatul vedea doar starea anterioară care nu era cea corespunzătoare.

În rest , procesele pentru tranziția dintre stări și de decodificare sunt aceleași ca într-un automat cu stări finite.

Dacă din „get_fullLine” este adusă o nouă instrucțiune , atunci procesul “state_reg” va decodifica biții de stare .Dacă se prelucrează aceeași stare , de exemplu: este cazul de scriere a unei variabile ce se află în starea S , iar din starea S va ajunge în M . În prima perioada , se va decodifica starea S , iar apoi are loc operația : current_state <= next_state.

Pe lângă aceste procese de controlare a stărilor , mai există încă 2 :

1. Proces pentru scrierea de write-back în core-ul la care îi aparține instrucțiunea.
2. Proces pentru scrierea de write-back în memoria principală.

6.5.2 Get_fullLine

Rolul acestei componente este unul simplu: acela de a căuta linia de adresă trimisă de la unul dintre core-uri în tabelul de stări. Căutarea are loc după id și după tag , index și offset. Valoarea datei nu este cea din tabel , este cea trimisă de către core.

Această componentă este activată de către cache controller printr-un semnal de enable . Acest semnal nu este altceva , decât semnalul “new fifo” , care este propagat . Astfel , de fiecare dată când este citită o nouă valoare de pe magistrala de date , este trimisă în această componentă.

Trimiterea spre UC a liniei de adresă cu stare este posibilă doar dacă tabelul confirmă găsirea liniei , prin semnalul “done_read_table”.

```
entity Get_fullLine_state is
  Port (clk,en : in std_logic;
        data_in : in std_logic_vector(65 downto 0);
        data_fromTable : in std_logic_vector(67 downto 0);
        data_toTable,DATA_FROMCC_TOTABLE_GET : out std_logic_vector(65 downto 0);
        data_out: out std_logic_vector(67 downto 0);
           done_get,read_table : out std_logic;
        done_read_table : in std_logic );
end Get_fullLine_state;
```

6.5.3 Tabelul de stări

Table_RAM implementează funcționalitatea de căutare și gestionare a liniilor de cache într-un tabel de stări. Aceasta:

1. Caută linia de cache trimisă de unul dintre core-uri, utilizând ID-ul și tag-ul.
2. Returnează linia găsită împreună cu starea sa (out_withState).

3. Permite modificarea stării liniilor pentru scriere sau citire, gestionând și invalidarea liniilor partajate.

```
 Port (data_in_fromCC : in std_logic_vector(67 downto 0);
      data_out_toCC ,data_in_fromCC_debug: out std_logic_vector(67 downto 0);
      original_line,other_line : out std_logic_vector(67 downto 0);
      out_withState: out std_logic_vector(67 downto 0);
      search_state : in std_logic_vector(65 downto 0);
      wb_fromCC,modify_state,read_table : in std_logic;
      wb_ToCC,done ,done_read: out std_logic;
      clk : in std_logic );

type matrix is array(0 to 127) of std_logic_vector(67 downto 0);
signal M : Matrix :=(
      0 => "0"&"0" &"11"& "0000" &X"9877" & "00" & "000100" & "0000" &
X"08888111",
      1 => "1"&"0" &"00"& "0000" &X"9877" & "00" & "000100" & "0000" &
X"00CCCCCC", -- celalalt care share uieste si ar deveni invalid
      others =>(others =>'0'));

signal data_out_aux : std_logic_vector( 67 downto 0) :=(others =>'0');
signal wb_to_aux : std_logic := '0';
signal done_aux : std_logic :='0';
signal found : std_logic :='0';
signal original_line_aux , other_line_aux : std_logic_vector(67 downto 0) :=(others =>'0');
```

Tabelul este format din 128 de linii, a câte 68 de biți , unde sunt stocate : id , read_type , starea(2) , tag(22), index(6) , offset(4) și data(32).

Pe lângă procesul de modificare a stărilor și de returnare a liniilor modificate (cel care are loc după căutarea linie corespunzătoare) , mai există încă unul , cu scopul de căutare , folosit de unitatea de control în prima fază.

```
process(clk)
variable index : integer range 0 to 127 := 0;
variable found_v : std_logic := '0';
begin
   if rising_edge(clk) then
      found_v := '0';
      done_read<='0';
       for i in 0 to 127 loop
           if M(i)(67) =search_state(65)and read_table = '1' and M(i)(63 downto 32) =
search_state(63 downto 32)then
              index := i;
              found_v := '1';
          exit;
          end if;
          end loop;
          if found_v = '1' then
            done_read<='1';
            out_withState<= search_state(65 downto 64) & M(index)(65 downto 32) &
search_state(31 downto 0);
         end if;
         end if;
```

7. Simulare și validare

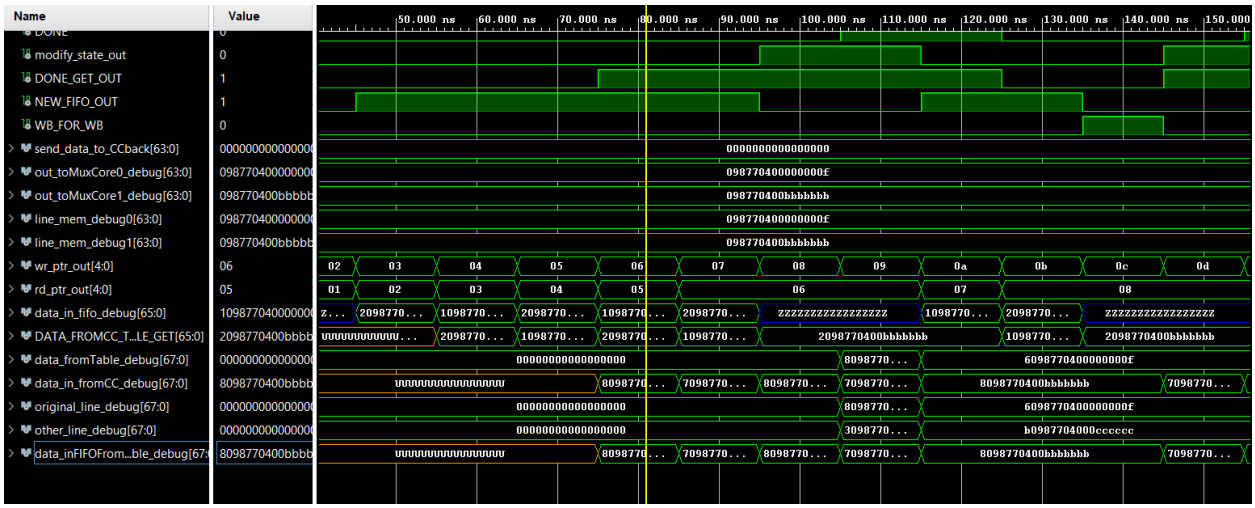
Un test de simulare care poate acoperi majoritatea cazurilor de testare pentru protocolul Snoopy este următorul :

- Core 0 adaugă în registrul 3 valoarea 15 , iar apoi dorește să o scrie.
- Core 1 în acest timp trimite o cerere de citire.

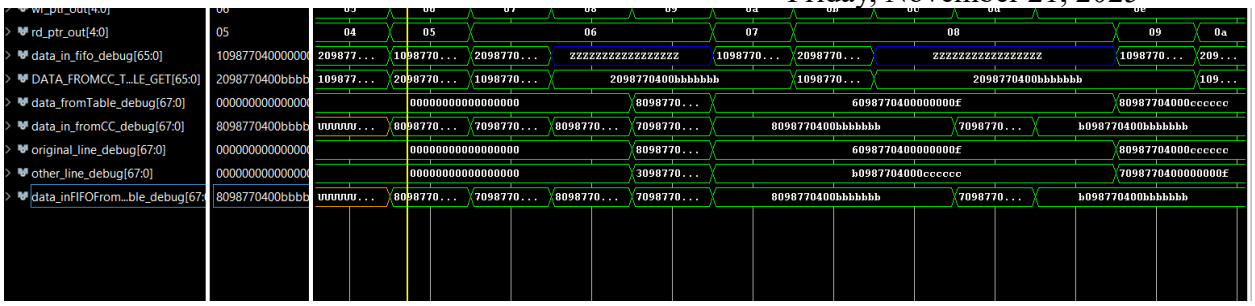
Adresele sunt aceleași (tag , index, offset) în memoria ambelor core-uri , de la adresa 0 :

- "0000" &X"9877" & "00" & "000100" & "0000" & X"0AAAAAAA" – valoare inițială în core 0
- "0000" &X"9877" & "00" & "000100" & "0000" & X"0BBBBBBB" – valoare inițială în core 1

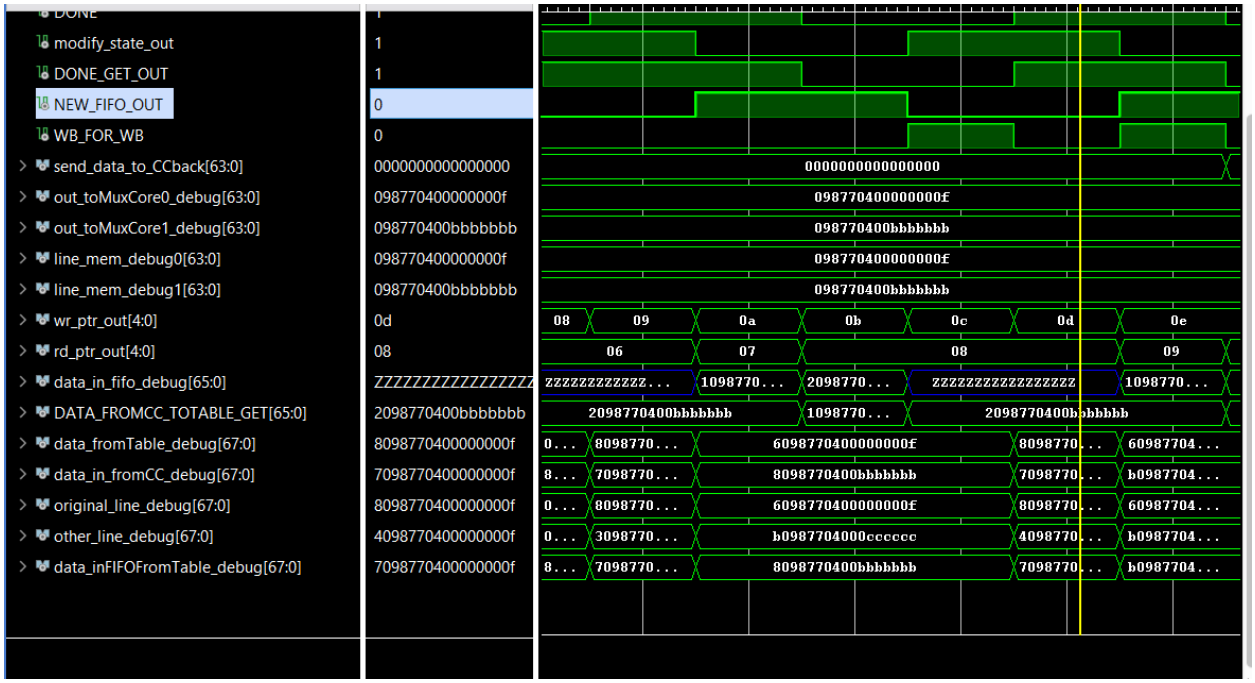
- Datele din tabelul snoopy sunt următoarele : "0"&"0" &"11"& "0000" &X"9877" & "00" & "000100" & "0000" & X"08888111"pentru core 0 și "1"&"0" &"00"& "0000" &X"9877" & "00" & "000100" & "0000" & X"00CCCCCC" pentru core1.



Inițial se observa în semnalele “out_mux” cererile pe care le trimit core-urile către switch . Ele conțin doar cei 64 de biți, cu valoarea din memorie.
Core0 trimite cu instructiunea deja executată de a adăuga 15 în registrul 3 , iar apoi trimite cerere de a-l scrie în memorie.
În semnalele “line_mem_debug” prezintă datele memoriei actuale de la fiecare core. Când are loc o scriere , se schimbă atunci . Din acest motiv este deja scris “f” foarte devreme.
În semnalul “data_in_fifo_debug” sunt afișate semnalele care sunt scrise în fifo . Primul este ales de către scheduler ca fiind adresa de la core-ul 1 pentru ca are pe MSB ‘1’ , iar următorul este ‘0’. Pe MSB-1 este tipul de instrucțiune. Se observa că nucleul0 scrie și nucleul1 citește. Într-adevăr, datorată simulării este scrisă aceeași cerere de mai multe ori , alterând cererea de la primul nucleu cu cererea de la al doilea nucleu.
În semnalul “data_fromCC_toTable” arată cum datele merg din cache controller către tabel pentru a citi starea adresei. Apoi este returnată în semnalul “data_inFIFOfromTable” câteva cicluri de ceas mai târziu.
În semnalul “data_inFIFOfromTable” MSB-2 și MSB-3 sunt stările adresei. Pentru id 0 este inițial pusă pe S , iar pentru id 1 pe I.



În această poză se observă că prima dată are loc citirea de la core1 , datele din memorie rămân la fel(original line și other line) , apoi are loc scrierea lui core0 și modifică stările și datele: datele la id 0 devine „f” , iar starea M , iar la celălalt starea devine I , invalidă. Apoi citește din nou id1 și ar avea loc un write back.



Ambele linii din memorie având aceeași dată , ‘f’, iar câteva cicluri de ceas mai târziu se va și scrie în memoria principală.

8. Concluzii

În urma proiectării și implementării mecanismelor de coerență a memoriei într-un sistem multi-core, s-a observat că gestionarea corectă a datelor partajate reprezintă una dintre cele mai dificile probleme întâlnite de orice arhitectură de calcul modernă. Comunicarea constantă dintre nuclee, necesitatea menținerii unei imagini consistente a memoriei și

detectarea conflictelor de acces evidențiază complexitatea interacțiunilor interne ce au loc într-un procesor multi-core.
Implementarea protocolului Snoopy și a componentelor asociate – controlere de cache, memoria principală și logica de tranzit între stările MSI – a demonstrat că asigurarea coerenței necesită un echilibru atent între performanță, cost hardware și simplitatea interfețelor de comunicare. De asemenea, a subliniat faptul că fiecare operație asupra unei linii de date poate avea implicații asupra întregului sistem, iar mecanismele de invalidare, partajare și write-back sunt esențiale pentru funcționarea corectă a procesorului.
Pentru dezvoltator, realizarea acestui proiect a contribuit la o înțelegere aprofundată a modului în care procesoarele moderne gestionează memoria cache, a conceptelor fundamentale de coerență și consistență, precum și a modului în care protocoalele snoopy coordonează activitatea mai multor nuclee. Totodată, implementarea practică în VHDL a permis consolidarea cunoștințelor privind descrierea hardware, sincronizarea semnalelor și structurarea unui sistem multicore la nivel arhitectural.

De asemenea , întreg proiectul poate fi găsit pe GitHub:
https://github.com/RazvanBarna/SnoopyProtocol

9. Resurse bibliografice:

- https://course.ece.cmu.edu/~ece600/lectures/lecture17.pdf
- https://passlab.github.io/CSCE513/notes/lecture23_TLP_IntroSMPSnooping.pdf
- Curs 4: Proiectarea MIPS cu ciclu unic de ceas , Curs 11 : Memorii de la materia : Arhitectura Calculatoarelor : https://users.utcluj.ro/~vcristian/AC.html
- Curs 6 SSC : SSC_Multicore6
- Laboratorul 5 SSC : FIFO https://moodle.cs.utcluj.ro/pluginfile.php/221087/mod_folder/content/0/SCS_Lab05.pdf?forcedownload=1
- https://www.tutorialspoint.com/what-are-snoopy-cache-protocols-in-computer-architecture
- https://en.wikipedia.org/wiki/Cache_coherence