Cuprins

1 INTRODUCERE

1.1 SCOPUL ȘI MOTIVAȚIA LUCRĂRII

Această implementare a unui microprocesor, bazat pe setul instrucțiuni RISC-V, s-a născut din dorința de a pune în aplicare conceptele digitale și electronice studiate. Un procesor se află la intersecția acestor ramuri de studiu, fiind astfel un bun exemplu practic.

Aceste dispozitive electronice reprezintă fundamentul tuturor științelor informatice, grație capacității computaționale intrinsece. Procesoarele, indiferent de gradul lor de specializare, au fost și rămân nucleul unei revoluții tehnologice pe care nici din pură ignoranță nu o putem omite, aceasta fiind prezentă până și în cele mai mundane aspecte ale vieții cotidiene.

Scopul acestei lucrări este de a traversa universul digital, începând din rădăcinile sale analogice, ajungând într-un final la organizarea ierarhică a numeroaselor entități digitale în a căror întregime se constituie un sistem de calcul complet funcțional.

Adesea este ușor sa ne pierdem în complicitățile ascunse printre miile de porți logice, un veritabil microcosm digital, însă prin mijloacele abstractizării și modularizării, proiectarea unui procesor devine nimic mai mult decât o modelare regulată a unui sistem descriptibil de operațiile algebrei Booleane. Pe parcursul lucrării, se va prezenta de asemenea o simplă implementare didactică a modulului de memorie cache, un component digital de o importanță deosebită, precum si problematica care cere o astfel de soluție.

Memoria cache cât și Microprocesorul vor fi realizate în limbajul de descriere hardware VHDL, acesta fiind considerat cunoscut în prealabil, entitățile urmând să fie simulate prin intermediul Vivado, soluție de design și sinteză hardware oferită de Advanced Micro Devices.

Procesorul implementat prin această lucrare nu va viza integral toate instrucțiunile setului oferit de specificația RISC-V, acesta fiind limitat la anumite instrucțiuni de calcul aritmetic de acces la nivelul memoriei inferioare. Însă, în ciuda acestor limitări, sistemul conceput este complet funcționat, existând posibilitatea extinderii capabilităților sale.

Implementarea memoriei cache prezintă o provocare, aceasta fiind factorul principal din cadrul unui microprocesor cu rol de optimizare a randamentului computațional. Prin design-ul memoriei cache avem posibilitatea de a ne familiariza în primul rând cu lacunele de performanță a procesorului. În al doilea rând, vom face contact direct cu modul de design al acesteia și conceptul de automat cu stări finite, cu ajutorul căreia se va minimiza dificultatea design-ului entităților și arhitecturilor VHDL complexe.

2 STUDIU BIBLIOGRAFIC

2.1 ARHITECTURA RISC

Înainte de realizarea unei analize asupra stadiului de dezvoltare și implementare al setului de instrucțiuni RISC-V, înțelegerea locului pe care filozofia RISC o are în disciplina arhitecturii calculatoarelor, este de o importantă deosebită.

Acronimul RISC, face referință la *reduced instruction set computer* sau calculator cu set de instrucțiuni reduse. Un microprocesor care implementează o astfel de filozofie, utilizează un set de instrucțiuni compact și puternic optimizat, garantând execuția rapidă a fiecărei instrucțiuni. Prin urmare, o caracteristică a acestei abordări, este faptul că microprocesorul va fi nevoit să execute un număr mai ridicat de instrucțiuni pentru a realiza aceleași operații efectuate de un calculator cu set de instrucțiuni complex, cunoscut și sub acronimul de *CISC*, printr-un număr observabil mai redus de instrucțiuni.

De-a lungul timpului, începând cu întemeierea arhitecturii RISC, au fost conceput mai multe seturi de instrucțiuni relevante, printre acestea enumerându-se următoarele: MIPS, ARM cât și setul care va reprezenta arhitectura procesorului implementat pe decursul acestei lucrări, RISC-V [1].

2.2 FAMILIA SETURILOR DE INSTRUCȚIUNI ARM

Seturile de instrucțiuni care aparțin familiei ARM sunt fără echivoc cele mai de succes dintre toate seturile aferente arhitecturii RISC. Acest succes este în mare parte datorat costurilor reduse de producție cât și eficienței computaționale ridicate. Dispozitivele dezvoltate în jurul microprocesoarelor ARM au un grade de utilitate ridicat, prezența acestora făcând-se simțită într-o vastă gamă de domenii. Cele mai evidente utilizări sunt reprezentate de telefoanele mobile și computerele personale, însă arhitectura ARM a reușit să se etaleze până și în domeniul computerelor de înaltă performanță, prin intermediul super computerului Fugaku.

Arhitectura ARM s-a bucurat de decenii întregi de dezvoltare și prin urmare de vaste îmbunătățiri, ajungând la un grad înalt de maturitate, lucru care-i definește utilitate contemporană [1].

2.3 SETUL DE INSTRUCȚIUNI RISC-V

Setul de instrucțiuni RISC-V reprezintă una dintre cele mai noi adiții aduse mulțimii familiilor arhitecturii RISC. Acest ISA nu funcționează pe baza unei licențe de utilizare, fiind un standard deschis, este permisă folosirea sa tuturor entităților legale sau persoanelor care doresc implementarea unui microprocesor sau a unui sistem integrat bazându-se pe acest set. Caracteristica acestui set de instrucțiuni este modul de acces la memoria de date cât

și la nivelul registrelor. Operațiile matematice se efectuează direct doar asupra datelor din registrele dorite spre acces.

De asemenea, setul de instrucțiuni este divizat în mai multe extensii. Pentru ca un microprocesor să fie considerat de tipologie RISC-V, acesta trebuie să respecte modul de funcționare al instrucțiunilor aritmetice de bază, precum sunt definite acestea de specificație. Instrucțiuni aritmetice complexe, precum înmulțirea și împărțirea numerelor, fac parte din extensia operaților asupra numerelor cu virgulă flotantă. Tot din altă extensie fac parte instrucțiunile de manipulare a biților, precum deplasările aritmetice la stânga sau dreapta.

Dacă se dorește execuția unui sistem de operare, este necesară implementarea instrucțiunilor din extensia Zicsr, aceasta acoperind registrele de control și status [1].

2.4 IMPLEMENTĂRI RISC-V

Datorită proliferării lipsite de licență cât și împărțirii setului în extensii, se poate observa un constant flux de implementări, variind de la simple exemple didactice la sisteme cu module multicip complexe.

Numeroase programe de studii care au ca scop dezvoltarea cunoștințelor despre organizarea calculatoarelor, obișnuiesc să prezinte ca suport didactic implementări succinte ale unui nucleu RISC-V [1].

Fiecare asemeni implementare prezintă ușoare diferențe arhitectural-organizatorice față de omologi săi. Aceste diferențe sunt produsul faptului că arhitectura RISC-V nu îngrădește utilizatorii săi într-o specifică topologie de organizare a modulelor care constituie în întregimea lor un microprocesor. Fiecare utilizator are astfel liber arbitru în definirea propriei organizări, atât timp cât respectă setul de instrucțiuni.

Se disting astfel două mari tipuri de microprocesoare RISC-V, ale căror implementări sunt disponibile spre analiză. Prima și cea mai comună este reprezentată de microprocesorul RISC-V SCP sau *single cycle processor*, cea de a doua purtând numele de *multi-cycle processor* sau pe scurt, MCP [1].

Procesorul single-cycle are un caracter preponderent didactic, utilizarea acestuia fiind mai puțin fezabilă în practică. În ciuda acestui fapt, acesta prezintă capabilități de calcul complete și are o oarecare utilitate computațională.

Pe de altă parte, un procesor multi-cycle se apropie mai mult de arhitectura de design a procesoarelor moderne, având un pipeline complex de execuție și implementând astfel un set mult mai vast de instrucțiuni. De asemenea, acest tip de procesor poate fi extins în ceea ce privește numărul său de nuclee. Un exemplu de procesor single-cycle, dar și unul multi-cycle simplificat sunt analizate de S. Harris, D. Harris [1]. Autorii prezintă capabilitatea limitată de execuție a unui nucleu single-cycle, dezvoltând din acesta procesorul multi-cycle. Din punct de vedere funcțional acestea sunt însă similare. Diferența apare în momentul în care se dorește execuția unor instrucțiuni care depășesc durata unei perioade a ciclului de tact.

3 FUNDAMENTARE TEORETICĂ

3.1 GESTIONAREA COMPLEXITĂȚII

Cand vine vorba de modelara unui sistem computațional de o complexitate ridicată, este de preferat să avem anumite fundamente în implementare, pe care să ne putem baza fără echivoc. În lipsa acestor principii este adesea usor să ne pierdem în complexitatea sistemului, rezultând astfel posibile erori care-și vor face simțită prezența în produsul final [1].

3.1.1 ABSTRACTIZARE

Abstractizarea este opusul specificității. Din punct de vedere conceptual, actul de abstractizare, indiferent de suportul teoretic asupra căruia este aplicat, ajută la simplificarea unei probleme a cărei complexități ar fi de altfel prea greu de tratat. Prin abstractizare, detaliile de la un anumit nivel logic al unui sistem, sunt redate sumar și considerate ca atare de către nivelele logice superioare.

Acest lucru poate fi observat într-o multitudine de domenii, de la arhitectura calculatoarelor la studiul fiziologiei medicale. De exemplu, bazându-ne pe cel din urmă domeniu enumerat, modul de funcționare a unui organism viu poate fi privit din mai multe perspective de abstractizare, începând de la interacțiunile biochimice si biomecanice de la nivelul unei celule, trecând pe urmă la modul în care aceste celule interacționează între ele formând variate țesuturi, ajungând într-un final la nivelul de abstracție al țesuturilor care împreună formează organe, fiecare nivel implicându-l direct pe precedentul său.

3.1.2 MODULARITATE

Modularizarea definește modul în care un sistem computațional va fi divizat în numeroase parți de sine stătătoare, acum numite module, fiecare cu un rol și o interfață de utilizare concis definită. Aceste module permit astfel reutilizarea entităților pe care le definesc, ne mai fiind nevoie de irosirea unei perioade mari de timp cu diverse noi implementării care sunt congruente cu un modul deja existent. Modularizarea ne permite de asemenea înlocuirea unor părți ale sistemului nostru cu altele de o eficiență mai ridicată, cât timp acestea respectă aceeași interfață pentru a permite comunicarea cu modulele adiacente.

3.1.3 IERARHIZARE

Ierarhizarea implică ordonarea într-o arhitectură a modulelor anterior definite. Arhitectura, în cazul nostru, va fi reprezentată de modul de organizare a microprocesorului ce urmează a fi dezvoltat, micro arhitectura acestuia. Organizarea ierarhică implică modularitatea dar vice-versa nu este mereu valabilă, modulele putând exista pe același nivel ierarhic, nefiind, prin urmare, subordonate unul altuia.

3.2 ABSTRACȚIA NUMERICĂ

Pentru a produce un rezultat de o oarecare utilitate, sistemele computaționale au nevoie de date. Aceste date sunt complet irelevante cât timp nu respectă un mod de reprezentare util sistemului. De asemenea, este importat de luat in considerare faptul ca datele hrănite pot avea semnificații diverse, complet obtuze una față de cealaltă.

Problema reprezentării datelor primește o importanță specială, deosebită chiar, dând naștere următoarei multitudini de întrebări, *care este modul corect de reprezentare*; *cum asigurăm coerența datelor cu analizarea acestora de către sistemul de calcul*; *cum ne asigurăm ca datele indiferent formatului lizibil uman, nu sunt ilizibile procesorului.*

Pentru a răspunde pe deplin, trebuie mai întâi să definim tipul datelor pe care microprocesorul le va accepta. Este rapid evident, din natura sistemului, că datele trebuie să fie numerice. Însă, nu la fel de evident este modul în care aceste numere vor fi reprezentate pentru a suporta toate operațiile admisibile de o unitate logică-aritmetică.

Cea mai reprezentativă caracteristică a unui sistem de numerație este numărul de simboluri unice utilizate de acesta. Numărul de simboluri poartă numele de radix și este congruent cu conceptul de bază numerică. Valoarea minimă pe care radixul unui sistem de numerație o poate lua este *1*, corespunzând unui sistem cu un singur simbol, fiecare număr conținând *n+1* simboluri față de precedentul sau *n*. Însă, trecând cu vederea această anomalie numerică, bazele care vor reprezenta suportul matematic al acestei lucrări sunt cea decimală, cea hexazecimală și cea binară. În Tabela 1 se pot observa bazele anterior menționate, însoțite de simbolurile aferente cât și de un exemplu reprezentativ.

Radix	Valori	Exemplu
Unar	1	111
Binar	0, 1	1000
Decimal	[0, 9]	10
Hexadecimal	$[0, 9] \cup [A, F]$	B4

Tabela 1: Intervalul de simboluri posibile, raportate la baza numerică

Un alt aspect important, strâns legat de radix, este numărul de simboluri *s* necesare pentru a reprezenta un număr oarecare *n* în baza *r*. Relația matematică care definește acest aspect este redată prin formula 2 [2].

$$s = log_r n \tag{1}$$

3.2.1 NUMERELE BINARE

Modul de funcționare a dispozitivelor digitale este constituit pe oscilațiile rapide ale semnalelor electrice, semnale a căror valori se identifică cu unul dintre membri faimosului cuplu binar, *0 și 1*. Prin urmare, datele vor avea o reprezentare care utilizează radixul binar.

Fiecare simbol dintr-un număr binar poarte numele de bit, un amalgam de 4 biți se numește nibble, iar o înșiruire de 2 nibble, echivalentă cu 8 biți, poartă numele de octet.

Bit-ul care corespunde celui mai mare exponent de 2 poartă numele de *msb*, iar bit-ul care corespunde celui mai mic exponent are denumirea de *lsb* [2].

Conceptul de împărțire a unui număr binar în octeți ajuta reprezentarea acestora într-o bază numerică superioară, în special cea hexazecimală. Figura 1 prezintă clar părțile constituente a unui octet și relația dintre acestea.

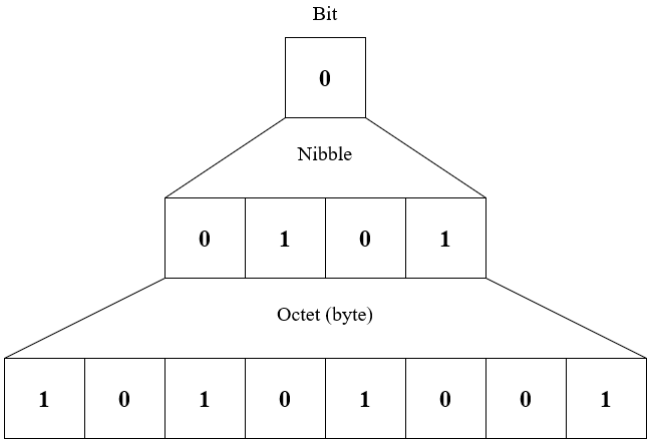

Figura 1: Modul de împărțire a unui octet, părțile sale constituente

Octetul va reprezenta unitatea fundamentală și indivizibilă pentru micro arhitectura microprocesorul dezvoltat prin această lucrare. Acesta, prin urmare, este cea mai mică entitate adresabilă cu care se va lucra. Un octet este limitat de numărul de date pe care le poate reprezenta, acestea fiind calculate prin exponentul 2^n, în cazul nostru, 2^8 sau 256 de valori.

Un lucru important de menționat este că valoarea maximă a unui număr pe *n* biți va fi mereu $2^n - 1$. Prin urmare, dacă dorim să reprezentăm o putere oarecare 2^n, vor fi necesari $n+1$ biți de date. Tabelul 2 prezintă relația dintre magnitudinea binară (numărul de biți folosiți în reprezentare) și cantitatea de date reprezentate prin plaja de valori adiacentă, ignorând existenta numerelor negative.

Biți de date	Numărul datelor	Interval valori
8	256	$0 \leq n \leq 2^8 - 1$
16	65536	$0 \leq n \leq 2^{16} - 1$
32	2^{32}	$0 \leq n \leq 2^{32} - 1$
64	2^{64}	$0 \leq n \leq 2^{64} - 1$

Tabela 2: Plaja de valori asumând numere strict pozitive, de magnitudini binare diverse

3.2.2 BAZA HEXADECIMALĂ

Datorită clarificării reprezentării datelor în radixul binar, înțelegerea bazei hexazecimale va fi cu atât mai simplă. Convertirea unui număr din binar in hexazecimal se face pe baza împărțirii acestuia în serii de *nibble*. În situația când reprezentarea binară nu are destui biți pentru a acomoda o așa diviziune a sa, se completează cu diferența de simboluri de 0

necesare. Tabela 3 prezintă valorile care vor fi utilizate în conversie cât și echivalența dintre baza binară, decimală și hexazecimală.

Nibble	Simbol	Decimal
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	A	10
1011	B	11
1100	C	12
1101	D	13
1110	E	14
1111	F	15

Tabela 3: Echivalența nibble - simbol hexazecimal

3.2.3 OPERAȚIILE MATEMATICE ȘI NUMERELE BINARE

Pentru a utiliza în mod corect reprezentările în această bază, modul în care calculele matematice sunt efectuate asupra numerelor binare necesită clarificare.

Datele nu sunt folositoare doar prin existența lor. Pentru a dobândi utilitate, acestea sunt supuse aparatului matematic, prin care se calculează diverse valori, asumând un algoritm corect, care ne oferă informații despre problema pe care dorim să o rezolvăm.

Cea mai elementară operație matematică care poate fi aplicată unui număr este adunarea.Însumarea numerelor este cu atât mai facilă cu cât numărul de simboluri folosite în reprezentarea acestora scade [2]. Spre norocul lumii digitale, radixul utilizat permite efectuarea operațiilor matematice în cele mai simple metode. Există doar 4 operații fundamentale posibile, acestea fiind prezentate în Tabela 4.

Operație	Sumă	Carry
0 + 0	0	0
0 + 1	1	0
1 + 0	1	0
1 + 1	0	1

Tabela 4: Operațiile fundamentale de însumare a numerelor binare

Dintre toate aceste operații, cea căreia îi vom oferi o importanță ridicată este $1+1 = 0$ *carry* 1. Aceasta ne obligă să adunăm o unitate biților de pe poziții superioare. Practic,

această sumă, odată ce este generalizată, ne spune că $2^n + 2^n = 2 \cdot 2^n = 2^{n+1}$, o trivialitate matematică. Diverse exemple de adunare ale numerelor binare pot fi consultate în Figura 2.

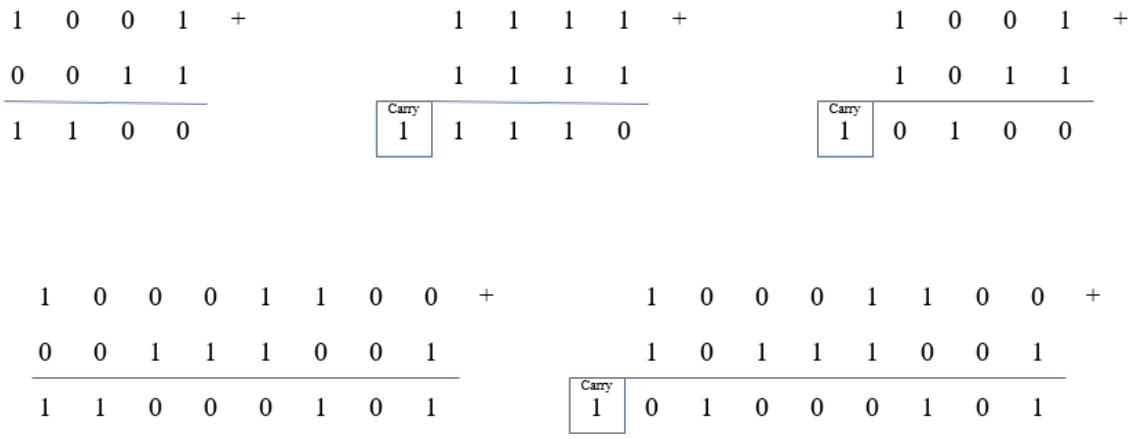

Figura 2: Exemple de adunare a numerelor binare

3.2.4 REPREZENTAREA NUMERELOR NEGATIVE

Modul în care numerele pozitive sunt adunate fiind acum clarificat, următoarea operație matematică tratată este scăderea. Aceasta poate fi vizualizată ca adunarea unui numar *a* la inversul aditiv al altui număr *b*. Această operație cere prin urmare un mod de reprezentare al numerelor binare negative, soluție care vine prin trei metode [1].

Primul mod de reprezentare este cel prin semn mărime. Acesta este intuitiv, fiind similar cu reprezentarea numerelor decimale cu semn. Bit-ul cel mai semnificativ devine acum bit-ul de semn, 1 reprezentând un număr negativă, iar 0 unul pozitiv. Tabelul 5 prezintă aplicarea acestei reprezentări asupra numerelor pe 8 biți.

Valoare binară	Semn mărime	Fără semn
00000000	0	0
00000001	1	1
00000010	2	2
...	...	...
01111110	126	126
01111111	127	127
10000000	-0	128
10000001	-1	129
10000010	-2	130
...	...	...
11111101	-125	253
11111110	-126	254
11111111	-127	255

Tabela 5: Reprezentarea prin semn mărime

Se pot astfel distinge următoarele lucruri:

- Există două reprezentări posibile pentru 0, și anume ± 0.
- Deși se acoperă tot 255 de valori numerice posibile (256 cu cel de al doilea 0), plaja de valori *signed* s-a distribuit egal numerelor negative și celor pozitive. Astfel, numerele *unsigned* semn mărime sunt cuprinse în intervalul $[-2^{n-1}+1, 2^{n-1}-1]$ unde $n > 0$

O altă medotă de reprezentare este prin complementul de 1. Conform acesteia, se inversează biții numărului binar pozitiv (biții cu valoarea 1 vor deveni 0 și viceversa) rezultând astfel inversul său aditiv. Spre exemplu, $00000001' = 11111110;\ 01010101' = 10101010$ iar, în cazul lui 0, $00000000' = 11111111$. Tabela 6 conține reprezentările numerelor de 8 biți.

Valoare binară	Semn mărime	Fără semn
00000000	0	0
00000001	1	1
...	...	...
01111110	126	126
01111111	127	127
10000000	-127	128
10000001	-126	129
10000010	-125	130
...	...	...
11111110	-1	254
11111111	-0	255

Tabela 6: Reprezentarea prin complement de 1

La fel ca în cazul reprezentării prin semn mărime, 0 dorește să respecte principiul superpoziției, diferența principală însă, precum se distinge din compararea Tabelei 5 cu Tabela 4, este faptul ca valorile negative sunt eșalonate invers.

Problema acestor reprezentări este cu atât mai vizibilă când se efectuează adunarea a două numere binare, rezultatul fiind mereu $111..1..111$, unda dintre reprezentările posibile ale lui 0. Soluția vine prin complementul de 2, complement format prin adăugarea unei unități reprezentării complementare de 1 [1]. Efectul însumării unitare este cel mai bine explicitat de Figura 3.

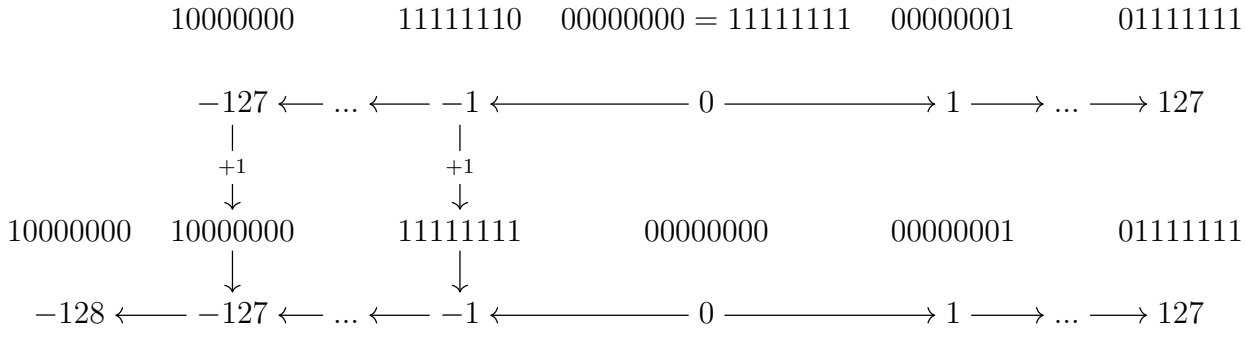

Figura 3: Efectul adunării unității asupra complementului de 1

Din această figură se observa următoarele:

- În cazul complementului de 2, nu mai există două reprezentări posibile pentru 0, locul suplimentar fiind luat de posibilitatea reprezentării unui număr negativ adițional, -128.
- Intervalul posibil de valori devine acum $[-2^{n-1}, 2^{n-1} - 1]$.

Odată cu clarificarea reprezentării numerelor binare negative, operațiile matematice fundamentale pe care microprocesorul modelat pe decursul acestei lucrări le va utiliza, pot fi acum implementate. Însă, pentru a face legătura dintre abstractul arhitectural al sistemului și datele numerice, este necesară tratarea entităților digitale fundamentale cunoscute drept porți logice.

3.3 ABSTRACȚIA LOGICĂ

Porțile logice sunt acele dispozitive care fac legătura dintre operațiile matematice abstracte definite anterior și implementarea propriu zisă a sistemului practic de calcul. În spatele unei porți logice se găsesc tranzistoarele, mici întrerupătoare electrice ale căror mărime a ajuns în zilele noastre să atingă ordinul nanometric (10^{-9} *metri*) [3]. Subtilitățile de funcționare ale unui tranzistor sunt dincolo de scopul acestei lucrări, acesta fiind considerat cutia neagră de la baza implementării entităților digitale.

O poartă logică are rolul de a implementa o funcție matematică a algebrei Booleane. Comportamentul acestora este considerat ideal, întârzierile de propagare a impulsului astfel neglijabile. Pentru analiza funcției algebrice descrise se vor utiliza tabele logice, a căror rol este de a relaționa în format tabelar semnalele de intrare cu rezultatul produs.

Un amalgam de porți logice cumulat după operația definită de o funcție algebrică formează un sistem logic de o complexitate variată. Printre acestea se număra dispozitivele de memorare, de la *flip-flop-uri* și *bistabile* la *RAM* și *ROM*, dar și cele aritmetice precum *ALU* (unitatea aritmetică logică) și *FPU* (coprocesorul pentru numere cu virgulă flotantă) [3].

3.3.1 POARTA BUFFER

Un buffer reprezintă cea mai simplă poartă logică. Rolul acesteia este de a transmite exact semnalul pe care-l primește ca intrare, adăugând însă o întârziere de propagare. Simbolul porții logice este prezentat de Figura 4.

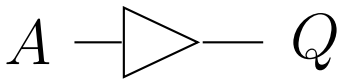

Figura 4: Simbolul porții logice Buffer

Utilitatea acesteia se poate observa în special la nivelul procesoarelor multi-cycle, buffer-ul având un rol important în organizarea modului de execuție a instrucțiunilor complexe care necesită multiple cicluri de tact.

3.3.2 POARTA NOT

De asemenea cunoscut sub numele de *inversor*, rolul acestei porți logice este de a schimba polaritatea semnalului primit.

A	Q
1	0
0	1

Tabela 7: Funcția logică NOT

Tabela 7 prezintă funcționalitatea acestei porți. Simbolul utilizat în diagramele circuitelor este prezentat prin Figura 5. Ecuația matematică a inversorului este $Q = \overline{A}$.

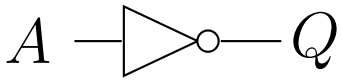

Figura 5: Simbolul porții logice NOT

3.3.3 POARTA SAU

Poarta logică *sau* reprezintă implementarea operației de disjuncție matematică [2], a cărei rezultat este 0 doar atunci când ambele intrări logice sunt 0. Tabela 8 prezintă valorile operației raportate la intrările logice. Expresia matematică a operației *sau* este $Q = A + B$.

A	B	Q
1	1	1
1	0	1
0	1	1
0	0	0

Tabela 8: Funcția logică SAU

Simbolul utilizat în diagramele circuitelor este prezentat prin Figura 6.

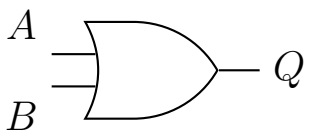

Figura 6: Simbolul porții logice SAU

3.3.4 POARTA ȘI

Poarta logică *și* reprezintă implementarea operației de conjuncție matematică [2], a cărei rezultat este 1 doar atunci când ambele intrări logice sunt 1. Tabela 9 prezintă valorile operației raportate la intrările logice. Expresia matematică a operației *și* este $Q = A \cdot B$.

A	B	Q
1	1	1
1	0	0
0	1	0
0	0	0

Tabela 9: Funcția logică ȘI

Simbolul utilizat în diagramele circuitelor este prezentat prin Figura 7.

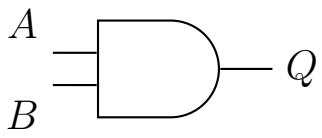

Figura 7: Simbolul porții logice ȘI

3.3.5 POARTA XOR

Poarta logică *xor*, de asemenea cunoscută ca *SAU exclusiv* setează semnalul de output pe 1 logic doar în cazul în care cel mult una dintre intrări este activă. Comportamentul acesteia este cel mai bine reprezentat de Tabela 10. Expresia matematică care descrie această poartă este $Q = A \oplus B$.

A	B	Q
1	1	0
1	0	1
0	1	1
0	0	0

Tabela 10: Funcția logică XOR

Această operație logică implică practic excluderea mutuală a semnalelor de intrare, indiferent de numărul acestora. Simbolul logic utilizat în diagramele digitale se poate regăsi în Figura 8.

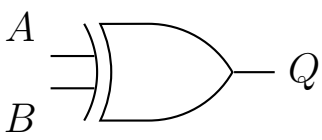

Figura 8: Simbolul porții logice XOR

3.3.6 POARTA NAND

Ca o regulă generala, totalitatea porților logice la ale căror denumire se atașează prefixul *N-* sunt varianta negată a porții logice originale, din această cauză doar poarte NAND va fi studiată, aceasta având o importanță deosebită. Poarta NAND este de asemenea cunoscută ca și poarta logică fundamentală [2]. Această denumire reiese din operațiile algebrei Booleane.

În mod firesc, semnalul de ieșire al acestei porți va fi complementarul operației *and*. La nivelul Tabelei 11 se poate observa acest lucru. Expresia matematică a acestei operații este $Q = \overline{A \cdot B}$.

A	B	Q
1	1	0
1	0	1
0	1	1
0	0	1

Tabela 11: Funcția logică NAND

Simbolul utilizat în diagramele circuitelor este prezentat prin Figura 9.

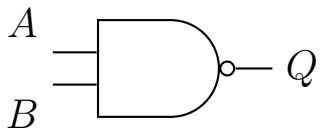

Figura 9: Simbolul porții logice NAND

3.3.7 MULTIPLEXARE

Pe lângă circuitele clasice reprezentate de porțile logice, un alt component digital, multiplexorul, merită studiat, acesta având o importanță deosebită. Prin multiplexare se înțelege selecția unui anumite intrări dintr-o lista de *n* posibilități. Selecția este dictată de un alt semnal de intrare numit selecție sau selector.

Multiplexara de asemenea are o operație opusă, aceasta purtând numele de demultiplexare. Dintr-o listă de *n* ieșiri posibile, se va alege conform semnalului selector drumul pe care-l va lua un semnal de intrare.

Atunci când se dorește utilizarea unui dispozitiv multiplexor, cel mai important parametru este reprezentat de numărul de intrări specificate de cazul de utilizare. Conform acestor *n* intrări, se va calcula utilizând expresia $\log_2 n$ biții necesari semnalului de selecție [2].

Figura 10 arată modul de reprezentare grafic al unui multiplexor cu 4 intrări pe 32 de biți și un semnal de selecție de 2 biți. Demultiplexorul este reprezentat tot prin acestă figura, însă, rotită cu 180 de grade, intrările devenind semnalele de ieșire și viceversa.

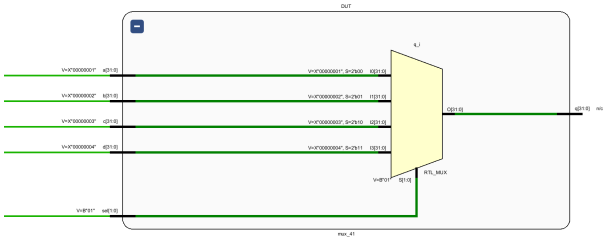

Figura 10: Diagrama logică a unui multiplexor

3.4 ABSTRACTIA ARHITECTURALĂ

Arhitectura face legătura dintre micro arhitectura sistemului și programatorul care dorește să-l utilizeze. Acest nivel de abstracție are rolul de a oferi un ghid asupra interacțiunii cu interfața definită de standardul RISC-V. Prin urmare, vor analiza detaliile arhitecturii, printre care se numără structura organizațională a registrelor, tipurile de instrucțiuni cât si modul în care octeții sunt organizați la nivelul acestora. Ne vom limita însă la extensia de bază RISC-V care include doar instrucțiunile fundamentale asupra numerelor întregi.

3.4.1 FIȘIERUL DE REGISTRE

Cel mai important detaliu micro arhitectural este fișierul de registre. Acesta este singurul dintre elementele de design a cărei respectare a tiparului impus de standardul RISC-V este imperativă.

Standardul impune un fișier de 32 de registre în cazul setului RV32I și 32 sau 64 de registre, în funcție de mărimea spațiului de adresare, pentru un procesor RV64I. Denumirea acestor registre, conform convenției, indică modul destinat de utilizare dar, nu există constrângeri la doar astfel de utilizări.

Tabela 12 ne ajută să identificăm registrele procesorului RV32I cât și utilizarea acestora. Coloana *Denumire Simbolică* prezintă modul de adresarea ale acestor registre la nivelul limbajului de asamblare [4].

3.4.2 INSTRUCȚIUNILE DE BAZĂ RISC-V

Operațile de bază executate de un procesor RV32I sunt transcrise în instrucțiuni având o mărime de 4 octeți. Aceste instrucțiuni pot fi de următoarele tipuri [4]:

- Instrucțiuni R, *Register-Register*, pentru operațiile de la registru la registru.
- Instrucțiuni I, *Immediate*, pentru operații asupra valorilor imediate.
- Instrucțiuni S, *Store*, cu rol de încărcare a datelor în memoria volatilă.
- Instrucțiuni U, *Upper Immediate*, cu rol de încărcare imediată a octeților cei mai semnificativi în registre.

Registru	Denumire simbolică	Descriere
x0	zero	Legat la valoarea 0
x1	ra	Adresa de return
x2	sp	Pointer stivă
x3	gp	Pointer global
x4	tp	Pointer thread
x5-7	t0-2	Valori temporare
x8	fp	Stocare date sau pointer cadru
x9	s1	Stocare date
x10-11	a0-1	Argumente apel funcție sau valori de return
x12-17	a2-7	Argumente apel funcție
x18-27	s2-11	Stocare date
x28-31	t3-6	Valori temporare

Tabela 12: Fișierul de registre RV32I

În cazul instrucțiunilor care utilizează valori imediate este importat să se țină cont de lățimea acestor valori. Familia de instrucțiuni U permite o lățime mai ridicată acestor tip de valori decât lățimea permisă de familia I [4].

3.4.3 INSTRUCȚIUNEA R

Instrucțiunile de acest format adresează 3 registre, 2 dintre ele fiind sursa datelor, cel de al 3-lea reprezentând destinația. Acestea sunt în general utilizate în operațile aritmetice, precum adunarea și scăderea numerelor [4]. Figura 11 prezintă formatul acestei instrucțiuni cât și câmpurile de date constituente.

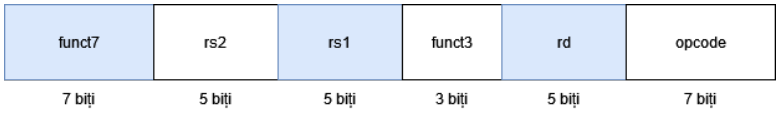

Figura 11: Instrucțiunea R și câmpurile de date

Campurile acestei instrucțiuni au următoarele semnificații [4]:

- Câmpul *opcode* indică tipul instrucțiunii care va fi executată de ALU.
- Câmpul *rd* conține adresa registrului unde va fi stocat rezultatul operației.
- Câmpul *rs1* conține adresa registrului primului operand.
- Câmpul *rs2* conține adresa registrului celui de al 2-lea operand.
- Câmpurile *funct7* și *funct3* conțin date adiționale pentru operația aritmetică de executat.

Un exemplu practic al acestei instrucțiuni se poate vedea în Figura 12. Datele din registrele 1 și 2 sunt însumate, rezultatul fiind plasat în registrul 3. De menționat faptul că trebuie acordată atenție sporită câmpurilor de specificare a operației de executat.

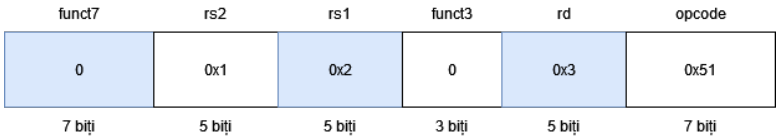

Figura 12: Exemplu de utilizare al instrucțiunii R

3.4.4 INSTRUCȚIUNEA S

Rolul instrucțiunilor de acest format este stocarea datelor. Pentru a facilita acest lucru, formatul instrucțiunii include 2 registre și câmpul imediat. Un registru este utilizat în calculul adresei de memorie, alături de un offset reprezentat de datele imediate, cel doilea conținând datele care se doresc stocate [4]. Figura 13 arată structura unei astfel de instrucțiuni.

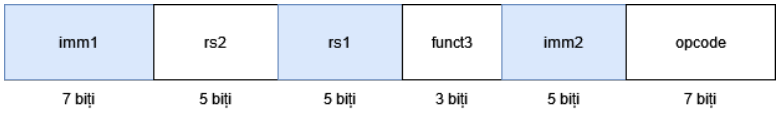

Figura 13: Instrucțiunea S și câmpurile de date

Semnificația câmpurilor este următoarea:

- Câmpul *opcode* indică tipul instrucțiunii de executat.
- Câmpul *rs1* conține adresa registrului în care se găsesc datele destinate stocării.
- Câmpul *rs2* conține adresa registrului față de a cărui valoare se va calcula offset-ul.
- Câmpul *funct3* conțin date adiționale despre operația de *store* executată.
- Câmpul *Immediate* format din *imm1* și *imm2*, în această ordine, reprezintă offset-ul adresei în care de dorește stocarea.

Un exemplu practic al acestei instrucțiuni se poate vedea în Figura 14. Valoarea din registrul 1 va fi stocată la adresa indicată de registrul 2, la care se va adauga un offset de *-1*.

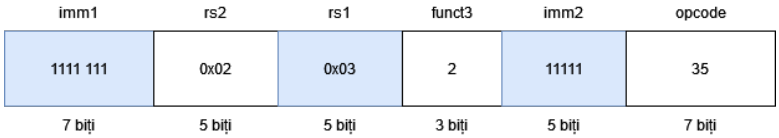

Figura 14: Exemplu de utilizare al instrucțiunii S

3.4.5 INSTRUCȚIUNEA I

Instrucțiunea I are rol în execuția operațiilor aritmetice cu valori imediate. Aceste valori sunt extinse la 32 de biti, ele ocupând doar 12 biți din lățimea instrucțiunii. În urma extensiei de semn, se efectuează operația aritmetică specificată prin câmpul *funct3* cu registrul *rs1*, rezultatul fiind stocat în registrul *rd* [4]. Figura 15 prezintă structura instrucțiunii și câmpurile relevante.

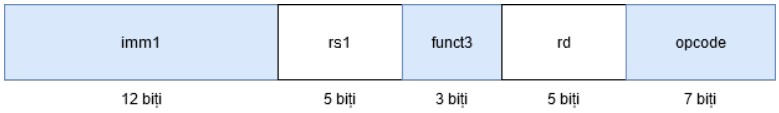

Figura 15: Instrucțiunea I și câmpurile de date

Fie o valoare imediată egală cu *5*, se dorește însumarea sa cu valoare din registrul *2*, rezultatul operației fiind trimis registrului *3*. Un astfel de exemplu se poate vedea în Figura 16.

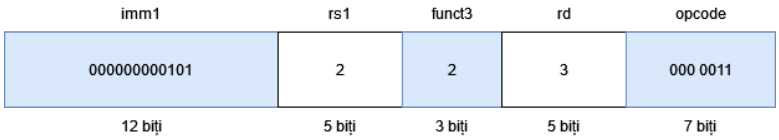

Figura 16: Exemplu de utilizare al instrucțiunii I

3.4.6 INSTRUCȚIUNEA U

Această instrucțiune este folositoare atunci când se dorește de către programator încărcarea unei valori imediate într-un registru, vizată ulterior unor calcule. Formatul instrucțiunii are în compoziția sa câmpul datelor imediate, întins pe 20 de biți, registrul destinație și codul identificator al operației de executat [4].

Cei 20 de biți sunt cei mai semnificativi ai numărului imediat, urmând ca acesta să fie extins la 32 de biți. Aranjamentul anterior descris poate fi observat în Figura 17.

Figura 17: Instrucțiunea U și câmpurile de date

Figura 18 prezintă un exemplu de utilizare, în registrul 5 fiind încărcată valoarea imediată 0X51AA3.

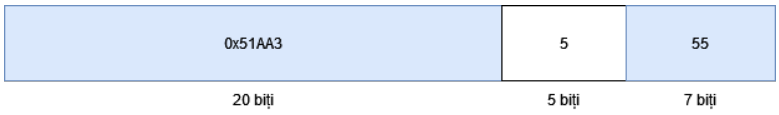

Figura 18: Exemplu de utilizare al instrucțiunii U

3.4.7 MEMORIA CACHE

Procesoarele prezintă o mare ineficiență când vine vorba de execuția instrucțiunilor care lucrează cu datele din memoria principală. Această ineficiență provine din diferența dintre timpul de execuție operațional al procesorului și timpul de acces din memoria principală la datele necesare calculului. Diferența dintre aceste două metrici temporale poartă numele de *latență*. Într-adevăr, avansurile tehnologice aduse componentelor digitale de-a lungul deceniilor au ajutat minimizarea aceste latențe, însă nu destul încât să o elimine din procesul de optimizare a pipeline-ului de execuție.

Astfel, prezența latenței ne conduce la analizarea a două mari problematici în modul de lucru cu date, și anume, problemele localizării temporale și a localizării spațiale. Localitatea spațială a datelor indică faptul ca există o mare posibilitate ca procesorul să dorească accesul unor adrese de memorie apropiate, de exemplu în cazul unei structuri de date vectoriale sau matriciale. Localizarea temporală pe de altă parte exemplifică posibilitatea accesului aceleiași adrese de memorie într-un interval de timp *t* [3].

Fie secvența de instrucțiuni RISC-V de acces a datelor din memoria principală prezentată în Figura 19. Se poate observa faptul că procesorul dorește accesul a mai multe spații de adresare consecutive. De fiecare dată când se extrag datele din aceste adrese, unitatea de execuție este obligată să aștepte conform unui timp Δt egal cu latența de acces a memoriei principale.

Secvență cod mașină	Latență acces	Adresă	Memoria principală
			x
lw rd, 1(rs1)	Δt	rs1 + 1	0x00ab0c01
lw t1, 2(rs1)	Δt	rs1 + 2	0xa4fb0402
lw t2, 3(rs1)	Δt	rs1 + 3	0x52b1211f
lw t2, 3(rs1)	Δt	rs1 + 3	
			x
			x
			x
			x
			x

Figura 19: Accesul la memorie și latența în cazul operației lw

Timpul total pe care procesorul îl are de așteptat este egal cu suma timpilor de acces al memoriei principale dintr-un interval de timp discret *t* în care sunt executate astfel de operații. Expresia matematică care se regăsește în Formula 2 prezintă un astfel de calcul al latenței totale, *Lt*, dintr-un interval de timp $[1, t]$.

$$Lt = \sum_{n=1}^{t} \Delta n \tag{2}$$

Latența de acces, în cazul unui procesor simplificat, implică ca urmare un timp de așteptare în care execuția procesorului este practic blocată, micșorând drastic eficiența. Însă, din acest exemplu, se pot de asemenea observa principiile localității anterior menționate. Procesorul accesează iterativ 3 zone de memorie alăturate, ultima fiind accesată de 2 ori, conformându-se principiului temporal.

Soluția micșorării acestei latențe vine sub forma unei memorii secundare bazate pe registre, aceasta purtând numele de *memorie cache*. Din cauza costurilor ridicate de implementare, dar și a amprentei ridicate asupra spațiului ocupat pe suprafața litografiată a procesorului, memoria cache are ca regulă generală o mărime limitată [5].

Există mai multe arhitecturi de implementare ale unei astfel de memorii, complexitatea digitală variind drastic. Diferența arhitecturală reiese din modul de asociere a adreselor din memoria principală cu spațiul de adresare a memoriei cache. Astfel, se disting memorii cache asociative direct, asociative pe seturi și nu în ultimul rând, asociative totale [5].

Metoda de asociere prezentată și implementată în această lucrare este cea asociativă directă, fiind de complexitatea cea mai redusă, prin comparație cu alternativele anterior menționate. Nu trebuie însă trecută cu vederea similaritatea cu celălalte tipuri de asocieri, hardware-ul fiind similar, tot similare fiind și metodele de calcul care vor fundamenta implementarea.

Cea mai importantă caracteristică a unei astfel de memorii este mărimea sa cât și lungimea în octeți a blocurilor care o compun. Figura 20 exemplifică aspectul unei memorii cache de 32 de octeți, aceștia fiind împărțiți în blocuri de câte 4 octeți. Prin urmare, fiecare bloc reține 4 cuvinte de câte un octet. Blocurile 0, 3 și 5 sunt încărcate cu date.

Este important de menționat faptul ca un bloc trebuie să acopere la cel mai rudimentar nivel cel puțin 2 adrese ale memoriei principale. Astfel, mărimea blocurilor este direct relaționată la mărimea memoriei de date, un bloc de cache acoperind *n* adrese din aceasta.

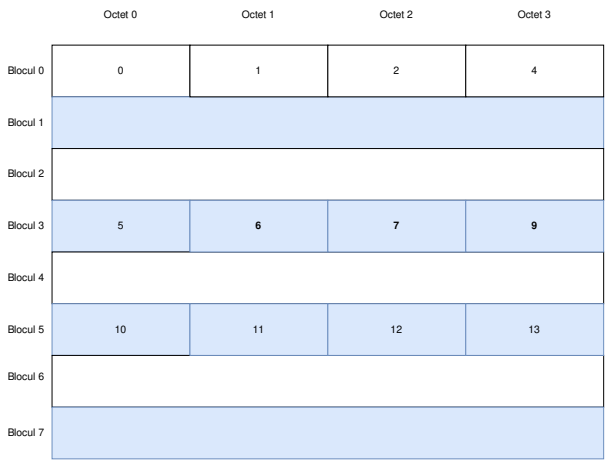

Figura 20: Arhitectura memoriei de date a cache-ului

Memoria principală este relaționată ca și mărime a cuvântului utilizat direct cache-ului. Prin urmare, plecând de la o memoria cache cu un cuvânt de 8 biți, prezentată în Figura 20, memoria de date va avea mărimea unei adrese egală cu un octet și o mărime totală complet arbitrară, fie aceasta 1KB. Cache-ul luat ca exemplu, acoperă prin urmare un bloc de 4 adrese. Figura 21 prezintă organizarea memoriei principale descrise anterior. Adresa memoriei va avea $log_2 1024 = 10$ biți.

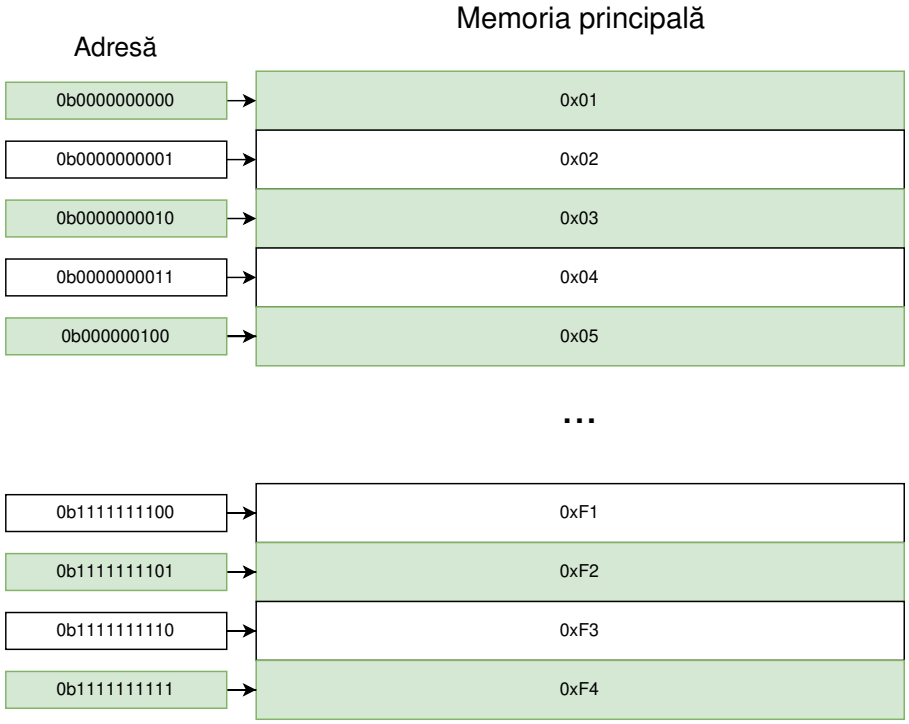

Figura 21: Organizarea memoriei principale

Pentru a reține datele eficient în cache, avem nevoie de un câmp de date din adresa memoriei principale care să ne semnifice blocul în care ne aflăm. Numărul de blocuri ale memoriei cache fiind egal cu 8, vor fi necesari $log_2 8 = 3$ biți numiți index. Ca și o regulă generală, pentru o memorie cache cu n blocuri, vor fi necesari $log_2 n$ biți.

De asemenea, pentru a relaționa fiecare adresă cu octetul din bloc în care aceasta trebuie depusă, avem nevoie de $log_2 4 = 2$ biți numiți offset, 4 fiind mărimea blocului din cache. Astfel, din adresa de 10 biți, 5 vor fi folosiți pentru a ne orienta după blocul în care ne aflăm. Restul de 5 sunt cunoscuți drept tag și vor fi utilizați pentru a identifica unic un set de 8 blocuri (32 de octeți). Figura 22 prezintă modul în care memoria a fost împărțită.

Figura 22: Câmpurile adresei de acces a memoriei principale

Combinația de tag și index reprezintă practic identificatorul unic al unui bloc din memoria principală, precum este acesta relaționat la nivelul cache-ului. Datorită importanței

pe care o posedă câmpul tag, acesta va fi stocat într-o memorie SRAM a cărei număr de coloane este egal cu numărul de blocuri din cache [5]. Prezența unui anumit tag la index-ul *i* din această memorie SRAM, denumită *TAG SRAM*, indică existența în cache a blocului identificat prin indexul și tag-ul respectiv și a tuturor celor 4 octeți din acesta. Figura 23 arată arhitectura memoriei TAG SRAM.

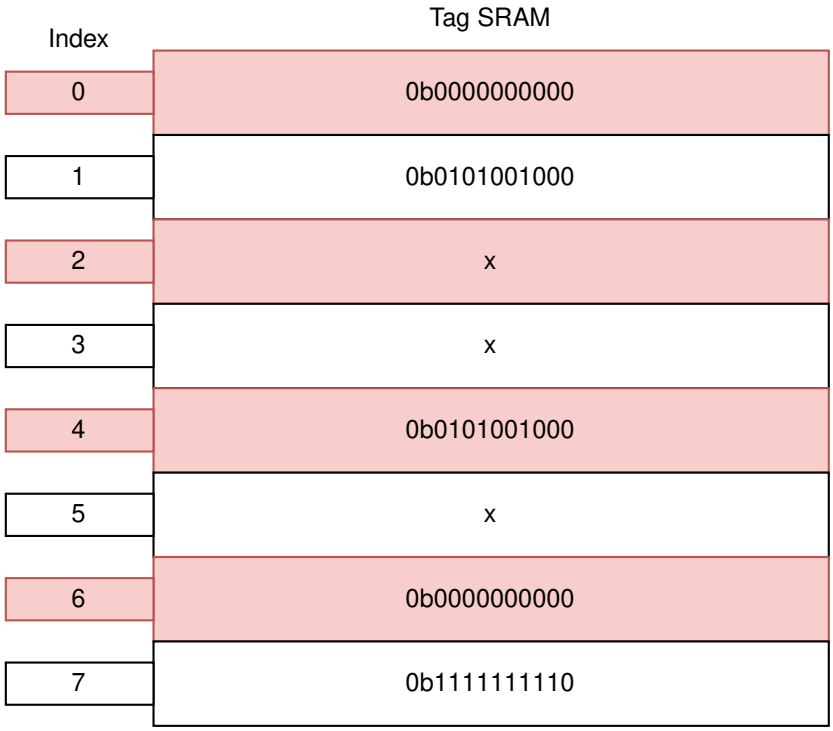

Figura 23: Organizarea TAG SRAM-ului

Se poate observa faptul că liniile corespondente indecșilor 0 și 6 au același tag stocat, același lucru este sesizabil și în cazul indecșilor 1 și 4. În cazul identității a doi sau mai mulți indecși, în memoria cache vor fi stocate blocuri care aparțin aceluiași tag, însă care cu un câmp *index* diferit.

Modul în care aceste două componente, cache SRAM și TAG SRAM comunică și stochează date, este arbitrat de o de-a treia entitate care poartă numele de *cache controller* [5]. Pe lângă arbitrare, unitatea de comandă are rolul de a interfata cu memoria, preluând date de la aceasta, conform adresei de acces cerute de procesor. De asemenea, răspunsul oferit înapoi procesorului este de o relevanță semnificativă.

Unitatea de comandă are 2 stări de răspuns posibile, *cache hit* și *cache miss*. Cache hit semnifică prezența datelor cerute pentru scriere sau citire de către procesor. Cache miss, pe de altă parte, indică lipsa acestor date din memoria cache.

Stările cache hit și cache miss determină acțiuni diferite la nivelul controlerului în funcție de tipul cererii procesorului, de citire sau de scriere.

La nivelul cache-ului predomină cererile de citire [5]. Acestea sunt în general îndeplinite prin extragerea unui bloc din memorie, dacă acesta nu există, semnalându-se un cache miss. Daca blocul a fost deja pre-încărcat, se omite cererea de extragere din memoria

principală, semnalându-se un cache hit, datele din adresa cerută fiind trimis de unitatea de comandă către procesor.

În cazul unei cereri de scriere în cache, lucrurile sunt puțin mai complexe. Există 2 mari strategi de îndeplinire a cereri de scriere, în cazul în care informațiile se găsesc deja în cache (cache hit), și anume *write through* și *write back* [5].

Write through implică actualizarea datelor atât la nivelul cache-ului cât și la nivelul memoriei principale, simultan.

Write back modifică datele doar la nivelul memoriei cache, urmând ca acestea să fie scrise în memoria principală odată cu înlocuirea blocului modificat. Semnalarea unui bloc modificat se va face printr-un bit suplimentar nimic *dirty bit*.

Dacă blocul în care se dorește scrierea nu se găsește în cache, controlerul va semnala un cache miss. Există din nou, 2 strategi relevante de a îndeplini cererea de scriere, acestea fiind *write allocate* și *no write allocate*.

Write allocate extrage blocul din memoria principală, după care modifică octeții relevanți ceruți de procesor.

No write allocate nu extrage blocul din memoria principală, acesta fiind tras în cache doar în momentul cererii de citire. Informațiile sunt prin urmare modificate doar la nivelul memoriei de date.

4 IMPLEMENTARE

4.1 ABSTRACȚIA MICROARHITECTURALĂ

Microarhitectura reprezintă organizarea ierarhică și modulară a componentelor digitale prin care se realizează implementarea practică a unui microprocesor. Precum parcurgem acest nivel de abstracție, vor fi prezentate toate componentele de bază pe care un microprocesor le necesită pentru a funcționa la cel mai de bază nivel. Odată ce toate astfel de componente sunt definite, este posibilă organizarea lor conform arhitecturii RISC-V, rezultând astfel într-un procesor funcțional.

Componentele vor fi prezentate atât în format grafic, prin diagrame digitale, cât și prin codul aferent entității VHDL. Testarea entităților se va face printr-un *testbench* VHDL [6], fiind prezentate formele de undă caracteristice.

4.2 SUMATORUL ȘI ARITMETICA NUMERELOR

Cea mai necesară operație pe care un sistem de calcul digital o poate executa este adunarea. Pe baza circuitului sumator se vor constitui alte componente digitale, precum *program counter-ul*. Din cauza utilizării extensive a circuitelor sumatoare, acestea sunt adesea ținta unei multitudini de optimizări, rezultând astfel circuite de o complexitate digitală mai ridicată, dar cu o amprentă temporală redusă [7].

Sumatorul implementat în această lucrare este de tipul ripple adder, carry-ul propagându-se de la un full adder la altul. Avantajul acestei implementări este ușurința modelării hardware, dezavantajul fiind lipsa de optimizare temporală și spațială a operației.

Schema digitală a unui sumator full adder poate fi observată în Figura 24. Acesta are rolul de a executa suma a 2 biți, precum s-a prezentat la nivelul abstracției numerice.

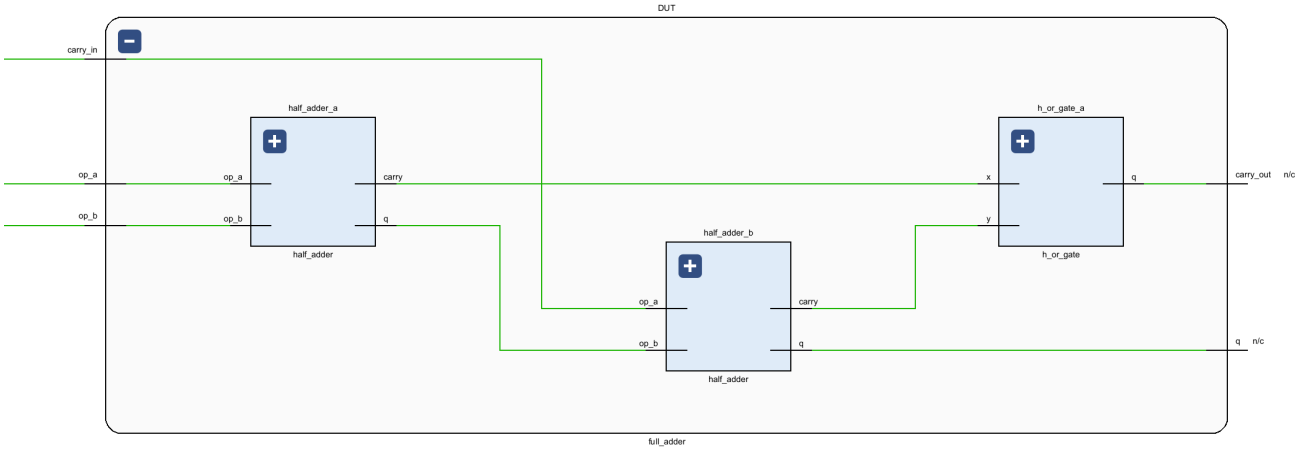

Figura 24: Sumator pe 2 biți

Implementarea VHDL a entității cât și a arhitecturii sumatorului integral este redată prin secvența de cod aferentă Figurii 25. Se observă faptul că această entitate este construită prin relaționarea a două sumatoare pe jumătăți și a unei porți logice *sau*.

```
entity h_or_gate is port(
    x: in std_logic;
    y: in std_logic;
    q: out std_logic
);
end h_or_gate;

architecture arch of h_or_gate is

begin
    q <= x or y;
end arch;

library arithmetic;
library IEEE;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity full_adder is port(
    op_a: in std_logic;
    op_b: in std_logic;
    carry_in: in std_logic;
    q: out std_logic;
    carry_out: out std_logic
);
end full_adder;

architecture arch of full_adder is
    component half_adder port(
        op_a: in std_logic;
        op_b: in std_logic;
        q: out std_logic;
        carry: out std_logic
    );
    end component;

    component h_or_gate port(
        x: in std_logic;
        y: in std_logic;
        q: out std_logic
    );
    end component;

    signal q_half_adder_a: std_logic := '0';
    signal carry_half_adder_a: std_logic := '0';
    signal carry_half_adder_b: std_logic := '0';

begin
    half_adder_a: half_adder port map(op_a => op_a, op_b => op_b, q => q_half_adder_a, carry => carry_half_adder_a);
    half_adder_b: half_adder port map(op_a => carry_in, op_b => q_half_adder_a, q => q, carry => carry_half_adder_b);
    h_or_gate_a: h_or_gate port map(x => carry_half_adder_a, y => carry_half_adder_b, q => carry_out);
end architecture;
```

Figura 25: Implementare VHDL a sumatorului pe 4 biți

Prin legarea serială a 8 sumatoare, rezultă entitatea fundamentală de adunare a procesorului nostru, și anume sumatorul de octeți sau *byte adder*. Structura acestuia este prezentată în Figura 26. Se poate observa faptul ca cele 8 sumatoare sunt împărțite în două grupuri de 4, fiecare numit *nibble adder*.

Implementarea VHDL a structurii digitale prezentată mai sus se poate vedea în Figura 27.

Sumatorul de 32 de biți, cunoscut de asemenea ca sumatorul de cuvinte sau *word adder* este prin urmare alcătuit din 4 sumatoare de octeți și poate fi văzut în Figura 28.

Pentru a efectua operația de scădere, sunt necesare mai multe lucruri. În primul rând, avem nevoie de un inversor pentru a reprezenta forma complementară a numărului de scăzut. Pe lângă acest lucru, este necesar un multiplexor 2:1, a cărui rol este alegerea dintre numărul inversat și valoarea sa inițială, în funcție de operația dorită. Tabela 13 prezintă modul în care va funcționa o astfel de selecție.

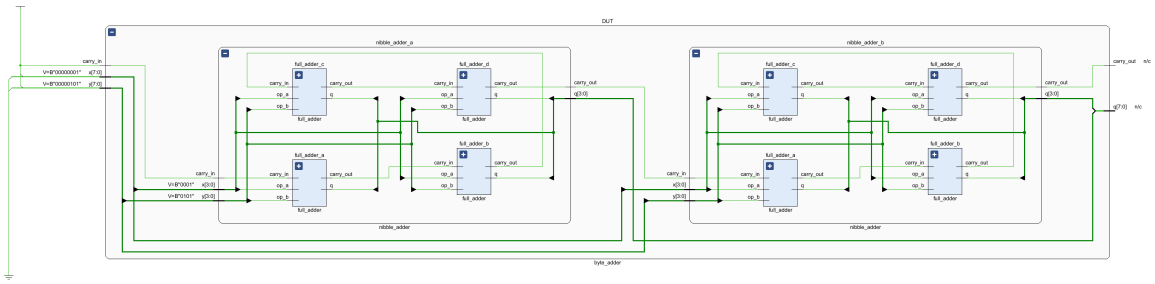

Figura 26: Sumator de octeți si elementele constituente

```vhdl
library IEEE;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity byte_adder is port(
    x: in std_logic_vector (7 downto 0);
    y: in std_logic_vector (7 downto 0);
    carry_in: in std_logic;
    q: out std_logic_vector (7 downto 0);
    carry_out: out std_logic
);
end byte_adder;

architecture arch of byte_adder is
    component nibble_adder port(
        x: in std_logic_vector (3 downto 0);
        y: in std_logic_vector (3 downto 0);
        q: out std_logic_vector (3 downto 0);
        carry_in: in std_logic;
        carry_out: out std_logic
    );
    end component;

    signal sig_carry_out: std_logic := '0';
begin
    nibble_adder_a: nibble_adder port map(x => x(3 downto 0), y=> y(3 downto 0),
                    carry_in => carry_in, q => q(3 downto 0), carry_out => sig_carry_out);
    nibble_adder_b: nibble_adder port map(x => x(7 downto 4), y=> y(7 downto 4),
                    carry_in => sig_carry_out, q => q(7 downto 4), carry_out => carry_out);
end architecture;
```

Figura 27: Implementare VHDL a sumatorului a sumatorului de octeți

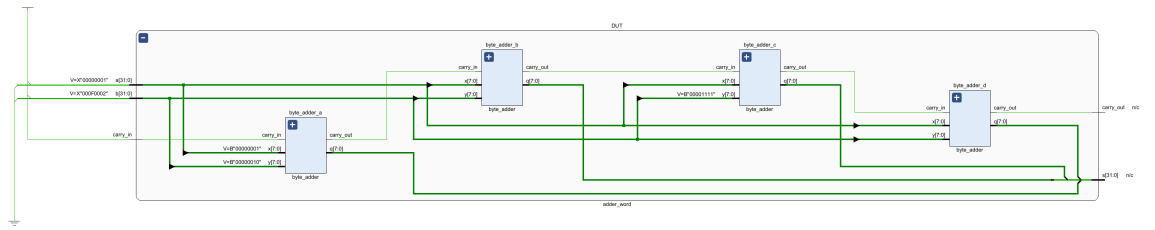

Figura 28: Sumator ripple carry adder de 32 de biți

Se poate observa faptul că semnalul de *carry* este setat pe 1 în cazul scăderii. Acest lucru formează practic complementul de 2 necesar scăderii, numărul fiind doar inversat în prealabil, rezultând un complement de 1.

A	B	Carry	Sel	Operație
0x0406001F	0x031400A5	0	0	Adunare
0x0013121F	0x01144EB5	1	1	Scădere

Tabela 13: Efectuarea operațiilor în funcție de selecția multiplexorului

Prin comasarea inversorului, a multiplexorului și a sumatorului rezultă astfel ceea ce vom denumi unitatea aritmetică. Schema digitală a acestuia se poate regăsi în Figura 29.

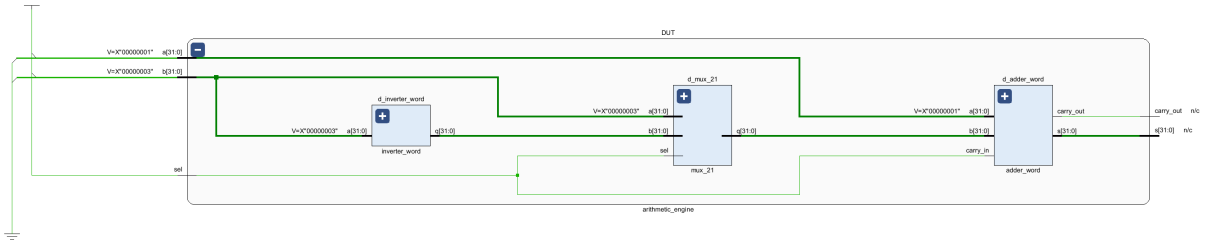

Figura 29: Unitatea aritmetică

Entitatea VHDL e unității aritmetice este redată prin codul prezentat în Figura 30.

```vhdl
library IEEE;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity arithmetic_engine is generic(
    word: integer := 31
    );

    port(
        a: in std_logic_vector (word downto 0);
        b: in std_logic_vector (word downto 0);
        sel: in std_logic;
        carry_out: out std_logic;
        s: out std_logic_vector (word downto 0)
    );
end arithmetic_engine;

architecture arch of arithmetic_engine is
    component adder_word generic(...
    );
    end component;

    component inverter_word generic(...
    );

    port(...
    );
    end component;

    component mux_21 generic(...
    );

    port(
        a: in std_logic_vector (word downto 0);
        b: in std_logic_vector (word downto 0);
        sel: in std_logic;
        q: out std_logic_vector (word downto 0)
    );
    end component;

    signal inverter_output: std_logic_vector (word downto 0) := (others => '0');
    signal multiplexed_output: std_logic_vector (word downto 0) := (others => '0');

begin
    d_inverter_word: inverter_word port map(a=> b, q=> inverter_output);
    d_mux_21: mux_21 port map(a=> b, b=> inverter_output, sel=> sel, q=> multiplexed_output);
    d_adder_word: adder_word port map(a=> a, b=> multiplexed_output, carry_in=> sel, s=>s, carry_out=> carry_out);
end architecture;
```

Figura 30: Implementare VHDL a unității aritmetice

4.3 OPERAȚII LOGICE ȘI UNITATEA LOGICĂ

Pe lângă operațiile aritmetice, un procesor de asemenea are posibilitatea de executare a operaților logice pe biți. Microprocesorul nostru va avea implementarea hardware a funcțiilor SAU, ȘI, XOR. Toate entitățile necesare efectuării acestor operații vor fi comasate într-un element hardware denumit unitatea logică [1].

Figura 31 conține schema digitală a motorului logic cât și a elementelor care-l definesc, cele 3 porți logice pe 32 de biți ȘI, SAU, XOR.

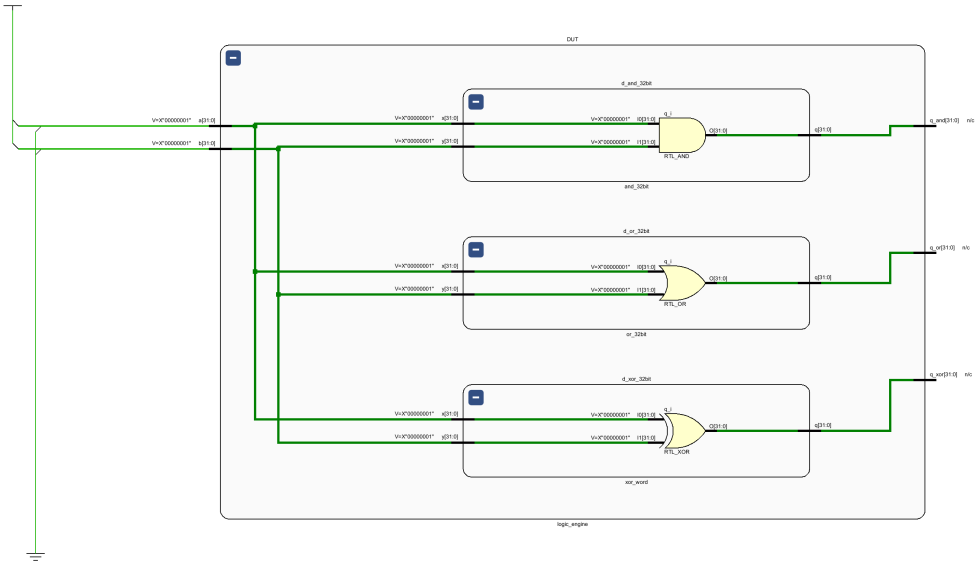

Figura 31: Motorul logic și entitățile interne

Spre deosebire de calculul aritmetic, operațiile logice sunt o trivialitate, lucru redat nu doar prin diagrama digitală, dar și prin codul asociat unității logice, prezent în Figura 32.

```
1   library IEEE;
2   use ieee.std_logic_1164.all;
3   use ieee.numeric_std.all;
4
5   entity logic_engine is generic(
6           word: integer := 31
7       );
8
9       port(
10          a: in std_logic_vector (word downto 0);
11          b: in std_logic_vector (word downto 0);
12          q_and: out std_logic_vector (word downto 0);
13          q_or: out std_logic_vector (word downto 0);
14          q_xor: out std_logic_vector (word downto 0)
15      );
16  end logic_engine;
17
18  architecture arch of logic_engine is···
52  begin
53      d_and_32bit: and_32bit port map(x=> a, y=> b, q=> q_and);
54      d_or_32bit: or_32bit port map(x=> a, y=> b, q=> q_or);
55      d_xor_32bit: xor_word port map(x=> a, y=> b, q=> q_xor);
56  end architecture;
```

Figura 32: Implementare VHDL a unității logice

4.4 UNITATEA ARITMETICĂ ȘI LOGICĂ

Unitatea aritmetică și logică este prima entitate ierarhică superioară pe care o vom reprezenta. Aceasta este formată din entitățile logice definite în prealabil, precum unitatea logică și cea aritmetică, dar conține de asemenea elementul pentru detecție de *overflow* [1].

Utilitatea detectării unui posibil *overflow* reiese în momentul în care dorim să executăm instrucțiunea *less than* [1]. În cazul a două numere, a și b, se va executa operația de scădere, semnul rezultant acestei operații determinând care dintre cele 2 numere este mai mare. O magnitudine negativă indică faptul ca primul număr, a, este mai mic, în schimb, semnul pozitiv indică spre b fiind numărul mai mic.

Precum se poate vedea în Tabela 14, intervalul alocat numerelor pe 32 de biți este depășit doar în cazul scăderii a două valori de semn opus.

A	B	A-B	Overflow
$A \leq 0$	$B \leq 0$	$-2^{32}+1 \leq A-B \leq 2^{31}-1$	Fără overflow
$A \geq 0$	$B \geq 0$	$-2^{32}+1 \leq A-B \leq 2^{31}-1$	Fără overflow
$A \geq 0$	$B \leq 0$	$0 \leq A-B \leq 2^{32}-2$	Overflow posibil
$A \leq 0$	$B \geq 0$	$-2^{33} \leq A-B \leq 0$	Overflow posibil

Tabela 14: Rezultatele posibile în cazul scâderii a 2 numere binare pe 32 de biți

De asemenea, semnul diferenței este mereu diferit de semnul descăzutului, în cazul unui overflow. Astfel, acesta poate fi detectat prin verificarea bit-ului de semn al diferenței în cazul în care operanzii sunt de magnitudini opuse. Tabela 15 prezintă cazurile în care se poate detecta overflow-ul, raportat la bitul de semn al operanzilor.

Bit de Semn A	Bit de Semn B	Bit de Semn A-B	Overflow
1	0	1	Fără overflow
1	0	0	Overflow
0	1	0	Fără overflow
0	1	1	Overflow

Tabela 15: Valorile logice în cazul scăderii a 2 numere binare

În Figura 33 se poate vedea diagrama logică a circuitului de detecție al overflow-ului, acesta fiind construit conform ecuației logice deduse anterior.

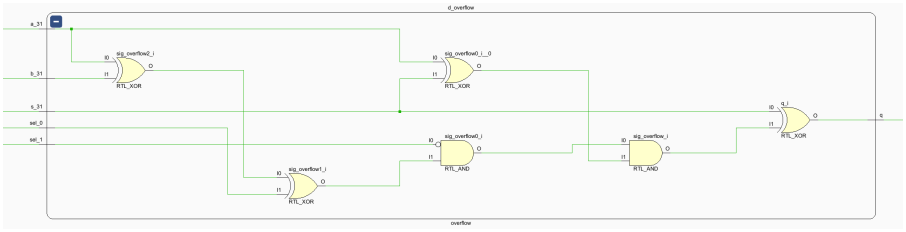

Figura 33: Diagrama logică a circuitului de detecție overflow

Prin urmare, îmbinând toate elementele modelate anterior, rezultă schema digitală a unității aritmetice și logice, vizibilă în Figura 34. Figura 35 prezintă codul entității și al arhitecturii VHDL aferent acestei componente.

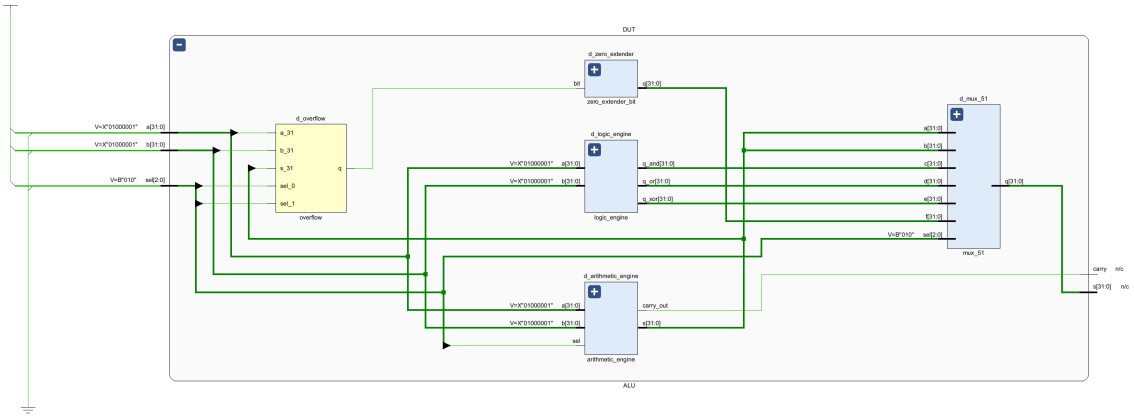

Figura 34: Unitatea aritmetică și logică

```vhdl
library IEEE;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity ALU is generic (
        word: integer := 31;
        opt: integer := 2
    );

    port(
        a: in std_logic_vector (word downto 0);
        b: in std_logic_vector (word downto 0);
        sel: in std_logic_vector (opt downto 0);
        s: out std_logic_vector (word downto 0);
        carry: out std_logic
    );

end ALU;

architecture arch of ALU is
    component arithmetic_engine is generic(…
    end component;

    component logic_engine generic(…
    port(…
    );
    end component;

    component overflow port(…
    );
    end component;

    component zero_extender_bit generic(…
    );

    port(…
    );
    end component;

    component mux_51 generic(…
    end component;


    signal sig_arithmetic_engine_output: std_logic_vector (word downto 0) := (others => '0');

    signal sig_logic_engine_and: std_logic_vector (word downto 0) := (others => '0');
    signal sig_logic_engine_or: std_logic_vector (word downto 0) := (others => '0');
    signal sig_logic_engine_xor: std_logic_vector (word downto 0) := (others => '0');

    signal sig_overflow: std_logic := '0';

    signal sig_zero_extender: std_logic_vector (word downto 0) := (others => '0');


begin
    d_arithmetic_engine: arithmetic_engine port map(a=> a, b=> b, sel=> sel(0), s=> sig_arithmetic_engine_output, carry_out=> carry);
    d_logic_engine: logic_engine port map(a=> a, b=> b, q_and => sig_logic_engine_and, q_or=> sig_logic_engine_or, q_xor=> sig_logic_engine_xor);
    d_overflow: overflow port map(a_31=> a(31), b_31=> b(31), s_31=> sig_arithmetic_engine_output(31), sel_0=> sel(0), sel_1=> sel(1), q=> sig_overflow);
    d_zero_extender: zero_extender_bit port map(bit=> sig_overflow, q=> sig_zero_extender);
    d_mux_51: mux_51 port map(a=> sig_arithmetic_engine_output, b=> sig_arithmetic_engine_output,
                              c=> sig_logic_engine_and, d=> sig_logic_engine_or, e=> sig_logic_engine_xor, f=> sig_zero_extender, sel=> sel, q=> s);
end architecture;
```

Figura 35: Implementare VHDL a unității aritmetice și logice

4.5 DISPOZITIVE DE MEMORARE

Pentru a-și putea executa instrucțiunile într-un mod autonom, dar și pentru a putea stoca informații, procesorul va avea nevoie de dispozitive de memorare. Acestea dispozitive vin în mai multe forme, cel mai bine distingându-se memoriile volatile și nevolatile.

Diferența principală dintre aceste două memorii este faptul că memoria volatilă are nevoie de o constantă reactualizare a datelor, aceasta fiind construită pe baza unei matrici de condensatoare și tranzistoare. Prin urmare, pentru a simplifica design-ul hardware al procesorului, dar și pentru a reduce nivelul de abstracție, se vor utiliza memorii nevolatile.

Memoria nevolatilă implementată în această lucrare se bazează pe celule de bistabile active pe front crescător. Datorită utilizării bistabilelor, latența de acces a datelor stocate în memorie este mică, principalul dezavantaj fiind implementarea practică, pricină costurilor ridicate ale unei astfel de memorii, utilizându-se mult mai multe tranzistoare decât în alternativa volatilă.

Memoria din care se vor încărca instrucțiunile executate de procesor va purta numele de *Memoria de instrucțiuni* iar, memoria generală destinată stocării datelor va purta numele de *Memoria de date* [1].

Accesul la nivelul ambelor memorii se va face la nivel de octet, fiecare cuvânt regăsindu-se astfel la o adresă multiplu de 4. Pentru a acomoda încărcarea instrucțiunilor din memorie, *program counter-ul* va incrementa mereu la adresa instrucțiunii curente 4, obținându-se astfel adresa următoarei instrucțiuni de executat.

Figura 36 prezintă diagrama memoriei de instrucțiuni, așa cum aceasta este generată prin intermediul sintezei hardware oferite de Vivado.

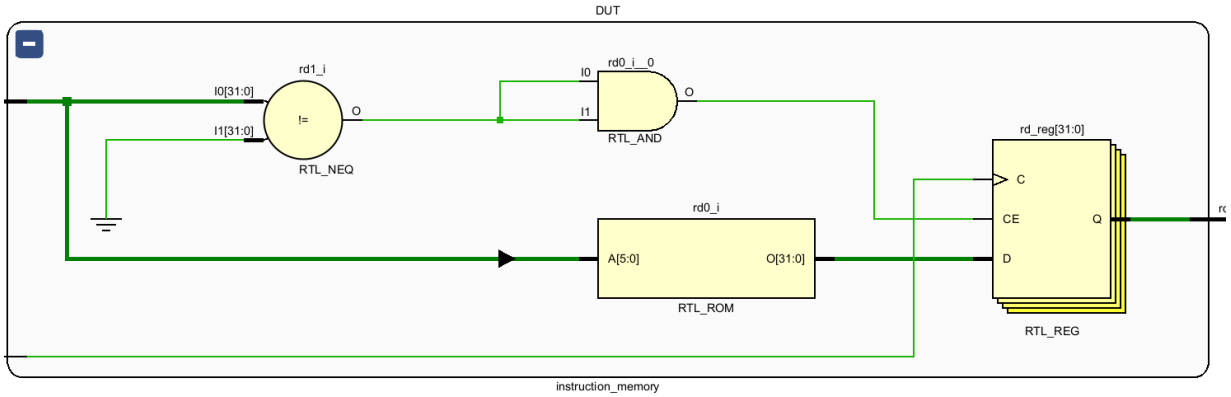

Figura 36: Memoria de instrucțiuni

Codul entității VHDL al acestei memorii poate fi consultat în Figura 37. Este notabil modul în care ultimii 2 biți ai adresei instrucțiunii de executat sunt ignorați, cu ajutorul secvenței de cod *address(7 downto 2)*.

```vhdl
library IEEE;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity instruction_memory is generic(
        word: integer := 31;
        xlen: integer := 8
    );
    port(
        clk: in std_logic;
        address: in std_logic_vector (word downto 0);
        rd: out std_logic_vector (word downto 0)
    );
end instruction_memory;

architecture arch of instruction_memory is
    type matrix is array(2**xlen - 1 downto 0) of std_logic_vector(word downto 0);
    signal instruction: matrix := (
        others => x"00000000");

begin
    process(clk, address) begin
     if rising_edge(clk) then
         if address /= x"UUUUUUUU" and address /= x"XXXXXXXX" then
            rd<= instruction(to_integer(unsigned(address(7 downto 2))));
        end if;
    end if;
    end process;
end architecture;
```

Figura 37: Implementare VHDL a memoriei de instrucțiuni

În Figura 38 putem observa design-ul hardware generat de sinteza automată pentru memoria de date. Se remarcă multiplexorul *RTL MUX*, rolul acestuia fiind de a lega la masă datele care ies din memorie, în cazul în care se dorește scrierea în aceasta, semnalul selector fiind reprezentat de *write enable(we)*.

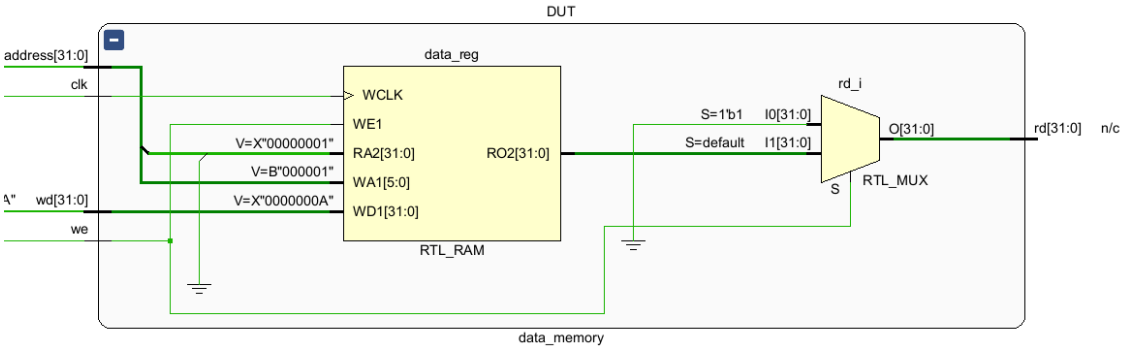

Figura 38: Memoria de date ca entitate VHDL

Codul entității VHDL al acestei memorii poate fi consultat în Figura 39. La fel ca în cazul memoriei de instrucțiuni, ultimii 2 biți ai adresei de intrare sunt ignorați, asigurând un comportament adresabil pe octeți.

```vhdl
library IEEE;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity data_memory is generic(
        word: integer := 31;
        memory_size: integer := 8
    );

    port(
        clk: in std_logic;
        we: in std_logic;
        address: in std_logic_vector (word downto 0);
        wd: in std_logic_vector (word downto 0);
        rd: out std_logic_vector (word downto 0)
    );
end data_memory;

architecture arch of data_memory is
    type matrix is array(2**memory_size downto 0) of std_logic_vector(word downto 0);

    signal data: matrix := (
        6 => x"00000001",
        2 => x"00000002",
        others => x"00000000");
    signal output: integer := 0;

begin
    output <= to_integer(unsigned(address(memory_size -1 downto 2)));
    process(clk, address, we, wd) begin
        if rising_edge(clk) then
            if we = '1' then
                data(to_integer(unsigned(address(memory_size - 1 downto 2)))) <= wd;
            end if;
        end if;
    end process;

    rd<= x"00000000" when we = '1' else data(output);
end architecture;
```

Figura 39: Implementare VHDL a memoriei de date

4.6 FIȘIERUL DE REGISTRE

Precum s-a prezentata anterior, fișierul de registre al procesorului nostru acomodează 32 de registre cu funcții diverse, fiecare având o capacitate de stocare egală cu un cuvânt. Memoriile implementate anterior sunt practic identice cu fișierul de registre, din punct de vedere al sintezei automate, datorită arhitecturii de tipologie nevolatilă.

Pe lângă ciclul de tact și write enable, fișierul de registre are 4 intrări relevante, *rs1, rs2, rs3, wrs3* și 2 ieșiri *rd1, rd2*. Rs1, rs2 și rs3 conțin adresele registrelor al căror acces îl dorim, wrs3 conținând cuvântul de scris în registrul indicat prin rs3. Implicit, rd1 și rd2 vor trimite în exterior cuvintele regăsite în registrele indicate de rs1 și rs2 [4].

Cea mai importantă caracteristică a acestui element de design este faptul că se poate realiza citirea datelor în paralel cu scrierea lor, astfel fiind posibilă executarea instrucțiunilor consecutive de efectuare a operațiilor cu datele din registre. Dintre toate cele 32 de adrese posibile, este interzisă scrierea datelor în registrul 0, acesta fiind legat la masă, facilitând efectuarea operațiilor cu 0, reducând astfel numărul de instrucțiuni necesare execuției unui program.

Figura 40 conține diagrama generată de sinteza automată a fișierului de registre. Se observă modul de multiplexare a ieșirilor, acestea fiind mereu 0 în cazul accesului registrului 0.

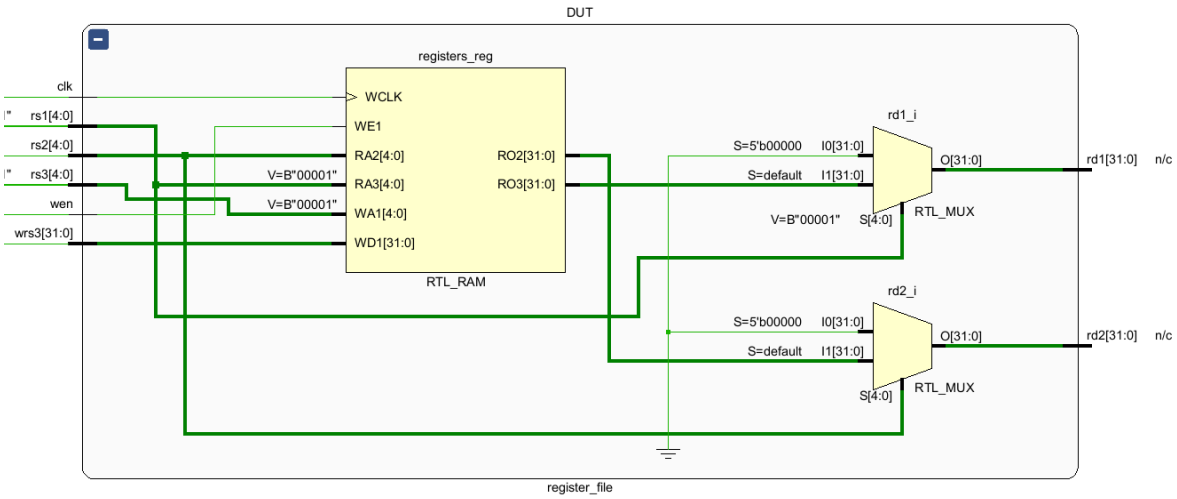

Figura 41: Fișierul de registre

Codul entității VHDL al acestui component se poate vedea în Figura 41.

```
6  > entity register_file is generic(···
21   end register_file;
22
23   architecture arch of register_file is
24       type matrix is array(word downto 0) of std_logic_vector(word downto 0);
25
26 >     signal registers: matrix := (···
36   begin
37       process(clk) begin
38           if rising_edge(clk) then
39               if wen = '1' then
40                   registers(to_integer(unsigned(rs3))) <= wrs3;
41               end if;
42           end if;
43       end process;
44
45       process(rs1, rs2, clk) begin
46           if to_integer(unsigned(rs1)) = 0 then
47               rd1 <= x"00000000";
48           else rd1 <= registers(to_integer(unsigned(rs1)));
49           end if;
50
51           if to_integer(unsigned(rs2)) = 0 then
52               rd2 <= x"00000000";
53           else rd2 <= registers(to_integer(unsigned(rs2)));
54           end if;
55       end process;
56
57   end architecture;
```

Figura 41: Implementare VHDL a fișierului de registre

4.7 EXECUȚIA SINCRONIZATĂ ȘI PROGRAM COUNTER-UL

Pentru a executa sincron și secvențial instrucțiunile din memorie, avem nevoie de un modul care să realizeze incrementarea periodică a adresei instrucțiunii curente.

Entitatea destinată acestui scop poartă numele de *program counter* și este formată dintr-un sumator, un modul generator de ciclu de tact și un bistabil pentru a transmite periodic indicele adresei de executat. Această implementare este suplimentată de un multiplexor 2:1, a cărui rol va fi selecția sursei instrucțiunii stocate în bistabil, în cazul în care se execută operația de *branch* [1].

Diagrama logică a acestui component poate fi observată în Figura 42.

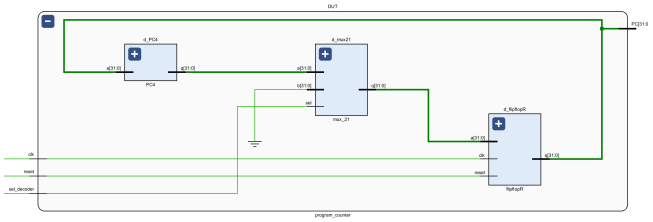

Figura 42: Program counter-ul și elementele constituente

Datorită abstractizării și modularizării implementării componentelor digitale definite până acum, program counter-ul nu necesită altceva decât definirea entității sale și a bistabilului constituent. Sumatorul și multiplexorul sunt definite în prealabil, fiind folosite de unitatea aritmetică și logică.

Figura 43 conține codul ce definește un bistabil sincron pe front crescător. Semnalul de reset are o importanță deosebită, inițializând contorul și începând astfel execuția instrucțiunilor.

```
5   entity flipflopR is
6       generic(
7           word: integer := 31
8       );
9       port(
10          clk: in std_logic;
11          reset: in std_logic;
12          a: in std_logic_vector(word downto 0):= x"00000000";
13          q: out std_logic_vector(word downto 0)
14  );
15  end flipflopR;
16
17
18  architecture arch of flipflopR is
19
20  begin
21      process(clk, reset, a) begin
22          if reset = '1' then
23              q<= x"00000000";
24          elsif rising_edge(clk) then
25              q <= a;
26          end if;
27      end process;
28  end architecture;
```

Figura 43: Implementare VHDL a bistabilului sincron

Codul aferent întregii entități VHDL a program counter-ului se regăsește în Figura 44.

```
6   entity program_counter is generic(
7       word: integer := 31
8       );
9       port(
10          sel_decoder: in std_logic;
11          clk: in std_logic;
12          reset:in std_logic;
13          PC: buffer std_logic_vector(word downto 0)
14      );
15  end program_counter;
16
17  architecture arch of program_counter is
18      component flipflopR ···
27      );
28      end component;
29
30      component PC4 generic(···
32      );
33      port(
34          a: in std_logic_vector(word downto 0);
35          q: out std_logic_vector(word downto 0)
36      );
37      end component;
38
39      component mux_21 generic(···
48      );
49      end component;
50
51      signal PCNext: std_logic_vector(word downto 0):= x"00000000";
52      signal Plus4: std_logic_vector(word downto 0):= x"00000000";
53
54  begin
55      d_flipflopR: flipflopR port map(clk=> clk, reset=> reset, a=> PCNext, q=> PC);
56      d_PC4: PC4 port map(a=> PC, q=> Plus4);
57      d_mux21: mux_21 port map(a=> Plus4, b=> x"00000000", sel=> sel_decoder, q=> PCNext);
58
59  end architecture;
```

Figura 44: Implementare VHDL a program counter-ului

Program counter-ul nu are o utilitate de sine stătătoare, fiind nimic mai mult decât un sumator ciclic cu semnalul de tact. Acesta dobândește semnificație funcțională în momentul în care este legat la memoria de instrucțiuni, căreia ii va furniza indexul următoarei secvențe de executat.

Figura 45 prezintă o astfel de construcție logică, aceasta fiind practic entitatea de încărcare a instrucțiunilor în unitatea de comandă a procesorului.

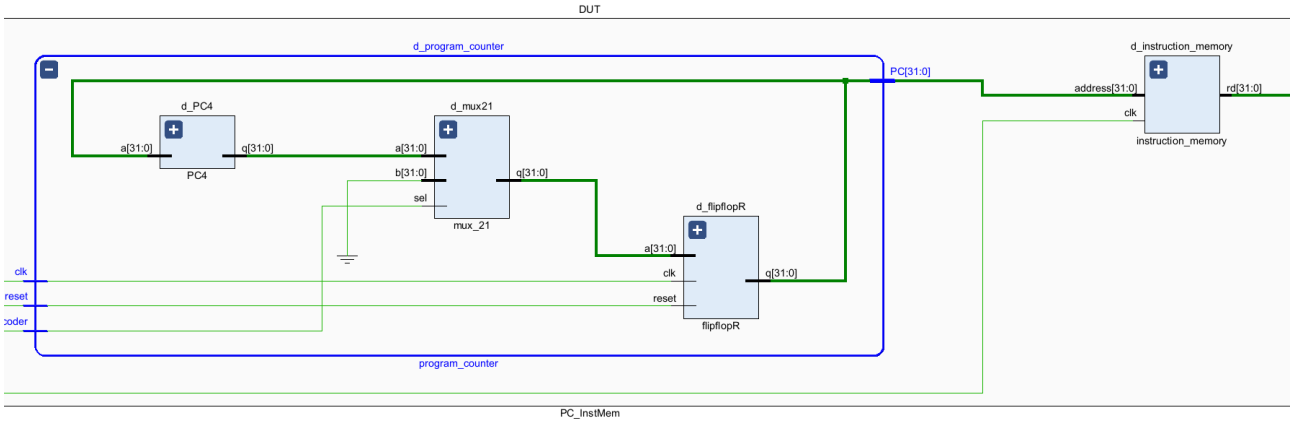

Figura 45: Construcția program counter - memoria instrucțiunilor

Cum se poate observa, program counter-ul este practic un feedback loop a cărei execuție începe la acționarea semnalului de reset al procesorului, acesta oprindu-se, din punct de vedere funcțional, la întâlnirea instrucțiunii nule, instrucțiune care semnifică finalitatea programului de executat.

4.8 DECODIFICATORUL DE INSTRUCȚIUNI

În lipsa unei entități care să interpreteze cuvântul de comandă transmis de memoria de instrucțiuni, procesorul nu are posibilitatea de a executa cod. Decodificatorul de instrucțiuni este elementul care va îndeplini un astfel de rol, trimițând biții de comanda mai departe elementelor digitale de execuție [2].

Decodificatorul va fi modelat după instrucțiunile pe care procesorul le va implementa. Pentru a asigura o capabilitate computațională cât mai funcțională, păstrând gradul de complexitate în limite adecvate, se vor implementa decodificările unui subset de instrucțiuni din setul de bază aritmetic RISC-V. Subsetul implementat va asigura decodarea unei instrucțiuni din fiecare familie aparținând setului RISC-V, permițând astfel o eventuală extensie a hardware-ului pentru a suporta mai multe operații. Printe instrucțiunile implementate și decodate se numără următoarele:

- Instrucțiunile aritmetice din familia R, asigurând calculul cu valorile din fisierul de registre. În acest set se vor regăsi comenzile *add*, *sub*, *or*, *and*.
- Instrucțiunile din familia S, acestea asigurând încărcarea valorilor din registrul *rs2* în memoria de date la adresa calculată ca valoarea din *rs1* adunată la valoarea imediată furnizată. Comanda regăsită în acest set este *sw (store word)*.
- Instrucțiunile din familia I, acestea asigurând încărcarea valorilor în registrul *rs1* dintr-o adresă din memoria de date calculată ca suma unui offset furnizat ca valoare imediată și valoarea din registrul *rs2*. Comanda *lw(load word)* este instrucțiunea din această familie pe care decodificatorul o va implementa.
- Instrucțiunile din familia B, acestea asigură posibilitatea execuției condiționate. Se testează egalitatea valorilor din registrele *rs1* și *rs2*. În caz de egalitate, program counterul sare la instrucțiunea egală cu adresa curentă adunată la valoarea imediate furnizată. Instrucțiunea de acest tip pe care o vom implementa este *beq*.

Din moment ce dorim o modularitate cât mai mare a hardware-ului,decodificatorul urmează să fie împărțit în două componente cu rol de decodificare. Primul component este practic decodificatorul principal, semnalând componentelor inferioare operațiile care sunt pe cale de a fi executate. Decodificatorul secundar poartă numele de decodificatorul aritmetic și are rolul de a orchestra modul de funcționare al unității aritmetice și logice [1]. În mod practic, decodificatorul aritmetic va primi un cuvânt de comandă de la entitatea superioară, urmând ca acel cuvânt să fie interpretat, comenzile aritmetice fiind trimise unității de calcul.

Modul de funcționare al decodificatorul principal este prezentat în Tabela 16. Câmpul *Instr* reprezintă instrucțiunea pe care dorim să o executăm iar *Op* reprezintă codul operațional al acesteia.

Instr	Op	ALUOp	Reg Write	ImmSrc	ALUSrc	MemWrite	ResultSrc	Branch
lw	0000011	00	1	00	1	0	1	0
sw	0100011	00	0	01	1	1	-	0
R	0110011	10	1	–	0	0	0	0
beq	1100011	01	0	10	0	0	-	1

Tabela 16: Decorodul principal

Câmpurile trimite de către decodificatorul spre componentele inferioare au următoarele semnificații:

- *ALUOp* este cuvântul de comandă trimis decodificatorul aritmetic, acesta va indica în concordanță cu câmpul funct3 și funct7 operație pe care unitatea aritmetică și logică o va executa.
- *Reg Write* este bit-ul care indică fișierului de registre dacă se va realiza o scriere la nivelul său.
- *ImmSrc* va indica unității de extindere a semnului modul în care această operație trebuie realizată. Acest câmp reprezintă o necesitate din moment ce valorile imediate vin în mărimi diferite în funcție de tipul instrucțiunii executate.
- *ALUSrc* comută între semnalul primit de al doilea operand al unității aritmetice și logice. Dacă ALUSrc are valoarea 1, sursa celui de al doilea operand va fi furnizată de extensorul de semn, daca acest bit este resetat, sursa va proveni din fișierul de registre.
- *Memwrite* înștiințează memoria principală ca urmează să se execute o scriere la nivelul său în momentul în care bitul este setat.
- *ResultSrc* are rolul de a comuta semnalele care vor fi stocate în fisierul de registre. Daca acest bit are valoarea 1, se va stoca în fișierul de registre cuvântul provenit din memoria principală. Valoarea 0 indică stocarea în fișierul de registre al rezultatului furnizat de unitatea aritmetică și logică.
- *Branch* semnifică că urmează să se execute operație de branch, facându-se astfel un salt de la adresa actualei instrucțiuni la alta, rezultată din adunarea valori curente din program counter la o valoare imediată oferită de instrucțiune.

Codul aferent acestui decodificatorul se poate consulta în Figura 49.

```vhdl
entity decoder is port(
        opcode: in std_logic_vector(6 downto 0);
        Branch: out std_logic;
        ResultSrc: out std_logic;
        MemWrite: out std_logic;
        ALUSrc: out std_logic;
        ImmSrc: out std_logic_vector(1 downto 0);
        RegWrite: out std_logic;
        ALUOpcode: out std_logic_vector(1 downto 0)
    );
end decoder;

architecture arch of decoder is
    signal output: std_logic_vector(8 downto 0);
begin
    process(opcode) begin
        case opcode is
            when "0000011" => --lw
            output <= "100101000";
            when "0100011" =>  -- sw
                output<= "00111-000";
            when "0110011" => -- R-type
                output<= "1--000010";
            when "1100011" => -- beq
                output<= "01000-101";
            when others =>
                output<= "---------";
            end case;
    end process;
    (RegWrite, ImmSrc(1), ImmSrc(0), ALUSrc, MemWrite, ResultSrc, Branch, ALUOpcode(1), ALUOpcode(0)) <= output;
end architecture;
```

Figura 46: Implementare VHDL a decodificatorul principal

Sinteza automată înlocuiește logica combinațională cu memoria ROM, datele necesare fiind stocate la adresele acestei memorii. În funcție de cuvântul instrucțiunii, multiplexorul intern al acestei memorii comută semnalul de ieșire, trimițând astfel componentelor subordonate semnalele de control corecte. Figura 47 prezintă acest fapt.

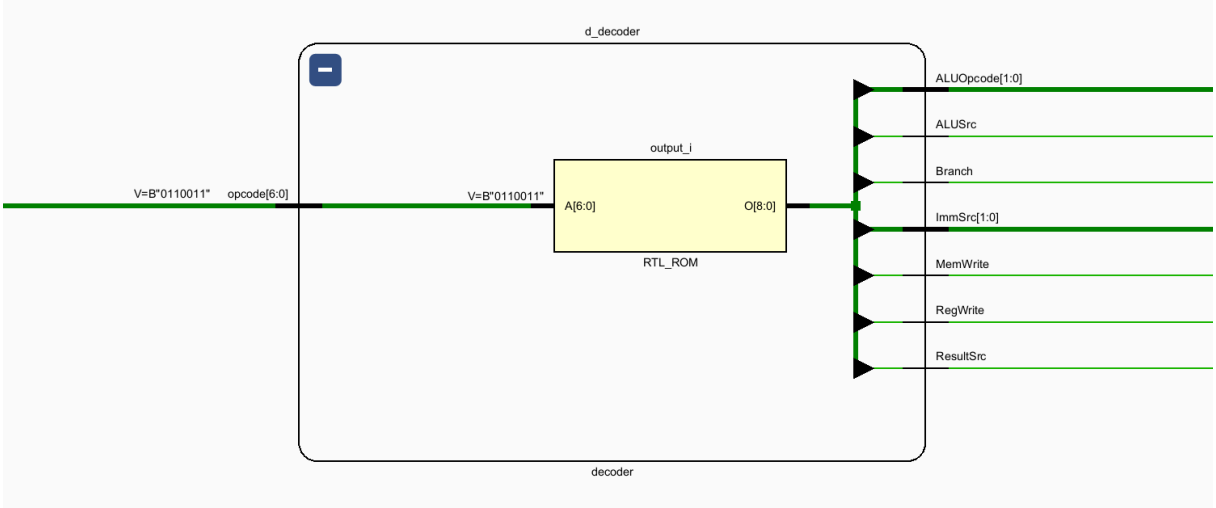

Figura 47: Entitatea VHDL a decodificatorul principal

Decodificatorul aritmetic pe de altă parte, are rolul de a semnala unității de execuție ce tip de operație este cerută de instrucțiune. Tabela 17 prezintă modul său de funcționare. ALUControl este semnalul de ieșire al acestei componente spre unitatea aritmetică și logică, specificând multiplexorului acesteia operația cerută.

ALUOp	funct3	op(5), funct7	ALUControl	Instrucțiune
00	—	–	000(scădere)	lw, sw
01	—	–	001(adunare)	beq
10	000	00, 01, 10	000(adunare)	add
10	000	11	001(scădere)	sub
10	010	–	101(mai mic decât)	slt
10	110	–	011(sau)	or
10	111	–	010(și)	and

Tabela 17: Decodificatorul aritmetic

Pe lângă datele provenite de la unitatea superioară, decodificatorul aritmetic extrage din cuvântul instrucțiunii datele *funct3, op din bitul al 5-lea, funct7*, acestea având rol în specificarea detaliată a operației. Combinația op(5), funct7 selectează între adunare în cazul instrucțiunilor aritmetice R, funct3 fiind, tot pentru această familie de instrucțiuni, selectorul operațiilor logice.

Codul VHDL al decodificatorul aritmetic este prezentat în Figura 48.

```
entity ALUDecoder is generic(
        word: integer := 31
    );
    port(
        opcode: in std_logic_vector(6 downto 0);
        ALUOp: in std_logic_vector(1 downto 0);
        funct3: in std_logic_vector(2 downto 0);
        funct7: in std_logic;
        ALUControl: out std_logic_vector(2 downto 0)
    );
end ALUDecoder;

architecture arch of ALUDecoder  is
    signal output: std_logic_vector(2 downto 0);
begin
    process(opcode, ALUOp, funct3, funct7)
        variable concat: std_logic_vector(1 downto 0);
    begin
        case ALUOp is
            when "00" =>
                output <= "000";
            when "01" =>
                output <= "001";
            when "10" =>
                case funct3 is
                    when "000" =>
                        concat:= opcode(5) & funct7;
                        if concat = "11" then
                            output <= "001";
                        else
                            output <= "000";
                        end if;
                    when "010" =>
                        output <= "101";
                    when "110" =>
                        output <= "011";
                    when "111" =>
                        output <= "010";
                    when others =>
                        output <= "---";
                end case;
             when others =>
                output <= "---";
            end case;
    end process;
    ALUControl <= output;
end architecture;
```

Figura 48: Implementare VHDL a decodificatorului aritmetic

Entitatea logică generată de sinteza VHDL pentru decodificatorul aritmetic se poate vedea în Figura 49. Acesta este practic format prin legarea în cascadă a 3 multiplexoare, fiecare comutând după un semnal de intrare diferit.

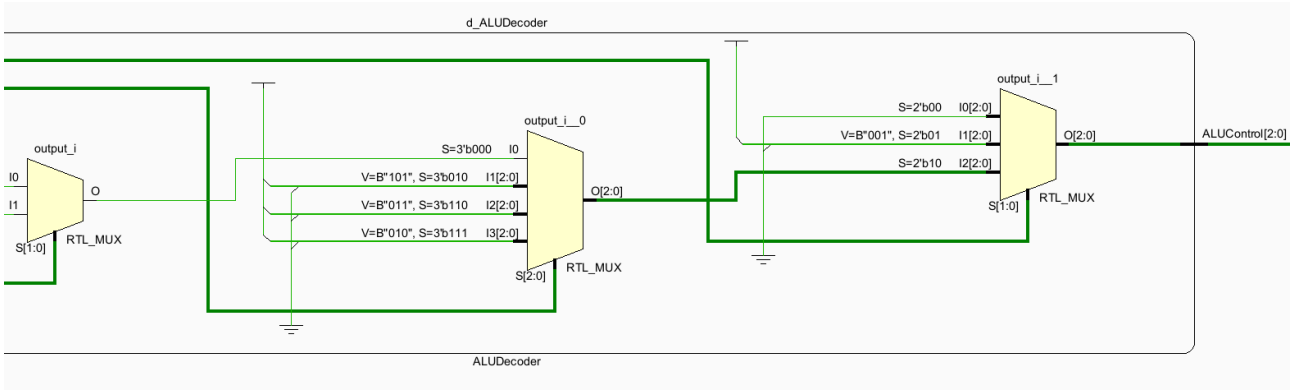

Figura 49: Entitatea VHDL a decodificatorului aritmetic

Prin conectarea celor două decodoare, cel aritmetic și cel logic, rezultă unitatea de decodificare centrală a procesorului. Această entitatea digitală poartă de asemenea numele de unitate de comandă în cadrul literaturii de specialitate. Figura 50 prezintă entitatea VHDL aferentă acestei componente.

```
entity Controller is generic(
        word: integer := 31
    );

    port(
        Zero: in std_logic;
        Instruction: in std_logic_vector(word downto 0);
        PCSrc: out std_logic;
        ResultSrc: out std_logic;
        MemWrite: out std_logic;
        ALUSrc: out std_logic;
        ImmSrc: out std_logic_vector(1 downto 0);
        RegWrite: out std_logic;
        ALUControl: out std_logic_vector(2 downto 0)
);
end Controller;

architecture arch of Controller is
    component decoder port(
        opcode: in std_logic_vector(6 downto 0);
        Branch: out std_logic;
        ResultSrc: out std_logic;
        MemWrite: out std_logic;
        ALUSrc: out std_logic;
        ImmSrc: out std_logic_vector(1 downto 0);
        RegWrite: out std_logic;
        ALUOpcode: out std_logic_vector(1 downto 0)
    );
    end component;

    component ALUDecoder port(
        opcode: in std_logic_vector(6 downto 0);
        ALUOp: in std_logic_vector(1 downto 0);
        funct3: in std_logic_vector(2 downto 0);
        funct7: in std_logic;
        ALUControl: out std_logic_vector(2 downto 0)
    );
    end component;

    signal sig_ALUOp: std_logic_vector(1 downto 0);

begin
    d_decoder: decoder port map(opcode=> Instruction(6 downto 0), Branch=>PCSrc,
                                ResultSrc=> ResultSrc, MemWrite=> MemWrite,
                                ALUSrc=> ALUSrc, ImmSrc=> ImmSrc,
                                RegWrite=> RegWrite, ALUOpcode=> sig_ALUOp);
    d_ALUDecoder: ALUDecoder port map(opcode=> Instruction(6 downto 0), ALUOp=> sig_ALUOp,
                                      funct3 => Instruction(14 downto 12), funct7=> Instruction(30), ALUControl=> ALUControl);
end architecture;
```

Figura 50: Implementare VHDL a unității de comandă

Circuitul digital generat de sinteză este nimic mai mult decât o interconectare a componentelor anterior menționate, acesta poate fi observat la nivelul Figurii 51.

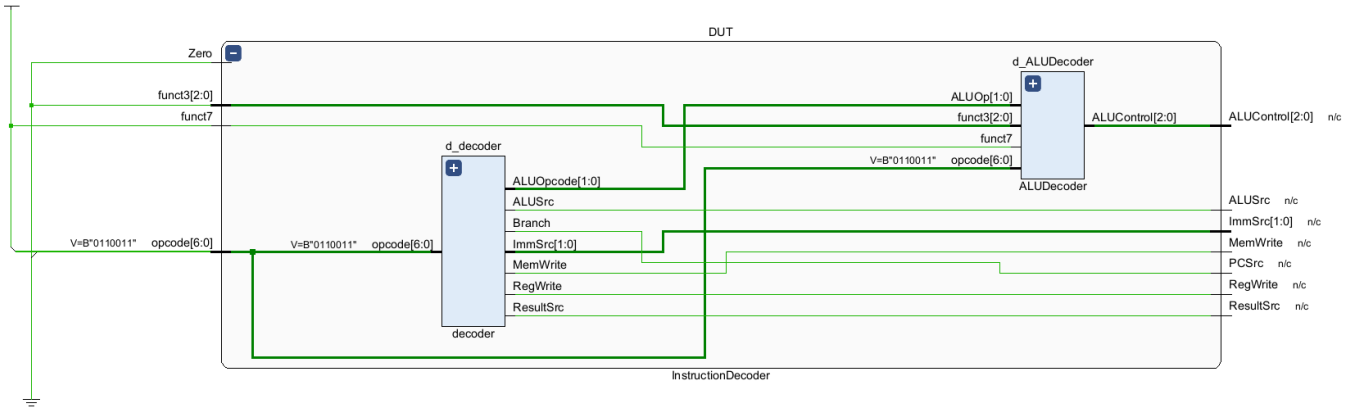

Figura 51: Schema digitală a unității de comandă

4.9 UNITATEA DE EXECUȚIE

Unitatea de execuție reprezintă entitatea nucleului procesorului care gestionează calculul aritmetic și logic, dar și calcularea adreselor următoarelor instrucțiuni de executat, în funcție de anumite restricții impuse. Restricțiile impuse alegerii subiectului următorului ciclu de execuție vin în general sub forma instrucțiunilor de tip branch [1].

Modelarea acestei componente implică conectarea tuturor entităților anterior definite, cu excepția unității de comandă. Un considerent important este faptul că dispozitivele de memorare, mai specific memoriile de instrucțiuni și de date, nu vor fi luate considerate în design-ul unității de comandă, acestea fiind tratate sub umbrela dispozitivelor externe. Această decizie este luată pentru a crește gradul de modularitate al procesorului, astfel înlăturând limitarea impusă de micile dimensiuni ale unei memorii specifice, existând mereu posibilitatea sporirii mărimi acesteia prin înlocuirea componentei. Mărimea memoriei este unul dintre principalii factori limitatori ai procesoarelor moderne, procesoare care se îndreaptă ușor spre un platou al creșterii vitezei ciclurilor de tact, soluția compensatoare fiind reprezentată de mărimea cache-ului și modificarea arhitecturii interconexiunii cu dispozitivele de memorare către una care eficientizează viteza de acces.

Înainte de a interconecta subcomponentele acestei unități, trebuie definit modul de organizare al execuției, și anume drumul parcurs de datele prevenite din memoria de instrucțiuni, de la operația cerută prin cuvântul de control la calcularea și stocarea rezultatului.

Cea mai vitală conexiune pentru buna funcționare a procesorului este reprezentată de legătura dintre memoria externă de instrucțiuni și program counter-ul [1]. La fiecare ciclu de tact, mai specific pe secvența crescătoare a acestuia, program counter-ul livrează

memorii de instrucțiuni index-ul adresei de memorie a căror date sunt destinate procesării.

Concomitent cu transmiterea unilaterală a datelor, counter-ul incrementează cu 4 octeți index-ul următoarei instrucțiuni de executat. Motivul unei incrementări cu 4 octeți este implementarea arhitecturală byte-addresable a memoriei. Există bineînțeles posibilitatea unei incrementări unare, în cazul în care se majorează mărimea datelor din cuprinsul memoriei la mărimea cuvântului cu care lucrează procesorul, în cazul nostru, la 4 octeți. Evident, o astfel de modificare ar însemna o micșorare a spațiului de adresare, adică al numărului de adrese disponibile, la o valoarea egală cu mărimea spațiului de adresare byte-addressable împărțit la numărul de octeți din mărimea adresei refactorizate.

În starea inițială, program counter-ul nu se prezintă ca fiind inițializat cu o valoarea. Inițializarea acestuia implică acționarea pin-ului de reset, încărcându-se astfel în counter valoarea 0. În ciclul de tact următor inițializării, se livrează memoriei de instrucțiuni valoarea prezentă în counter, în cazul nostru 0. Tot în acest ciclu, noua valoare a counter-ului este aleasă ca rezultatul multiplexării dintre valoarea incrementată anterior și o nouă valoare calculată, în cazul instrucțiunii beq. Valorile nu sunt încărcate instantaneu datorită naturii secvențiale a acestor componente, ele fiind construite pe baza bistabilelor. Această procedură se repetă până la finalizarea execuției sau până cand adresele din memoria de instrucțiuni sunt epuizate. Figura 52 prezintă modul în care decurge procedeul descris, acesta reprezentând practic momentul de începere al execuției.

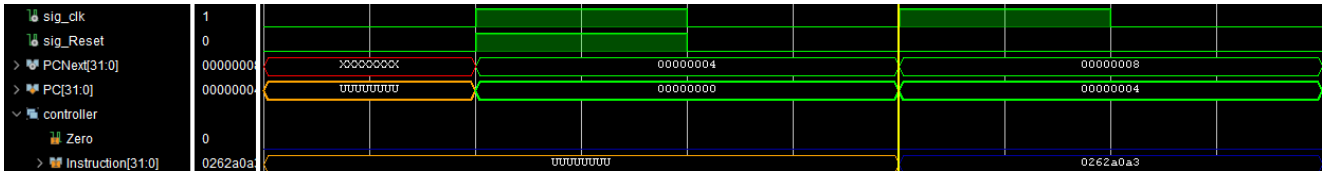

Figura 52: Încărcare și incrementarea instrucțiunilor

Se observă faptul că înaintea acționării semnalului de reset, *sig rst*, program counter-ul, reprezentat prin semnalul *PC*, este neinițializat, valoarea lui fiiind egală cu *U(Uninitialized)*. Odată ce semnalul de reset este acționat sincron cu semnalul de tact, *sig clk*, program counter-ul este inițializat cu 0. Neexecutându-se o instrucțiune de tip *beq*, valoarea nou calculată pentru counter este egală cu 4, vizibilă la nivelul semnalului *PCNext*.

La următoarea perioadă a ciclului de tact este transmisă valoarea 0 memoriei de instrucțiuni, aceasta transmițând mai departe prin semnalul *Instruction* cuvântul de comandă destinat decodării și execuției. Tot în acest ciclu de tact se modifică valoarea din counter cu cea calculată anterior. Acest procedeu se repetă pentru fiecare instrucțiune de executat. Din punct de vedere digital, circuitul implementat până în acest punct este cel prezentat în Figura 45.

Odată ce adresa instrucțiunii a fost calculată, urmează ca aceasta să fie transmisă unității de comandă pentru decodarea cuvântului. Concomitent cu decodarea are loc, în funcție de instrucțiune, accesul la nivelul fișierelor de registre, al memoriei și ulterior transmiterea datelor din registrele sau adresele memoriei cerute spre unitatea aritmetică și logică.

De asemenea, dacă instrucțiunea are un câmp imediat, se produce cuvântul de calcul prin extensia de semn.

Prin urmare, semnalele de intrare ale fișierului de registre destinate accesului vor fi legate la câmpurile de acces al registrelor din cuvântul produs de memoria de instrucțiuni. Semnalul de scriere într-un registru va fi multiplexat între valoarea provenita de la ALU și cea extrasă din memorie, în cazul în care se execută instrucțiunea *lw(load word)*.

Semnalele de ieșire ale fișierului de registre sunt direct corespondente cu semnalele de intrare ale unității aritmetice și logice. În cazul celui de al doilea operand care se transmite spre ALU, este însă nevoie de un multiplexor pentru a selecta între valoarea imediată produsă de unitatea extensiei de semn și cuvântul provenit dintr-un registru cerut. Această multiplexare este necesară din cauza nevoii de calcul cu valori imediate, lucru fară de care nu se poate realiza accesul la memoria date.

Astfel, diagrama digitală generată de sinteza automată a modului anterior descris de relaționare a componentelor este prezentată în Figura 53. Acestă entitate digitală va purta numele de unitatea de execuție a procesorului.

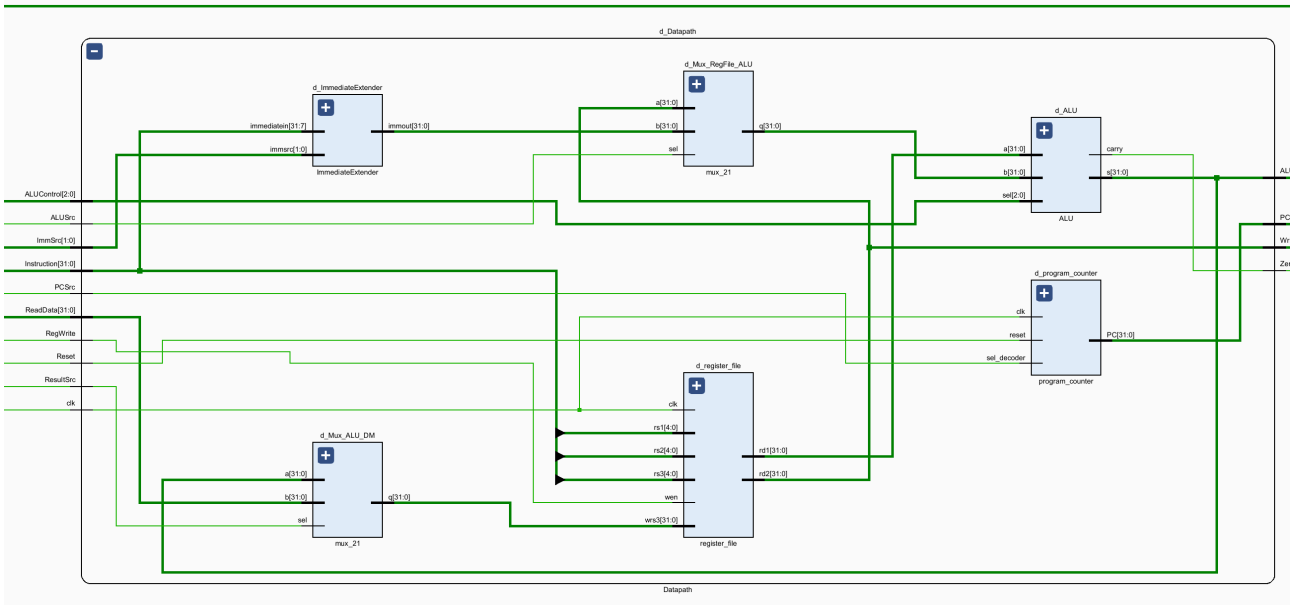

Figura 53: Unitatea de execuție

Se pot observa clar legăturile dintre componente și rolul operațiilor de multiplexare. De asemenea, sunt vizibile intrările provenite de la unitatea de comandă, semnalele fiind direcționate fiecărui element cu rol în execuție, sigura excepție fiind program counter-ul. Memoria principală furnizează date prin semnalul *ReadData*, acesta fiind ulterior multiplexat pentru scriere în registre, în cadrul multiplexorului *ALU-DM*.

Semnalele de ieșire sunt datele calculate de ALU, datele provenite din registrul specificat cât și următoarea adresă a instrucțiunii ce urmează a fi executată. Datele provenite din unitatea aritmetică și logică au pentru majoritatea comenzilor un rol intern destinat stocării în registre, dar în cazul instrucțiunilor de tipul *sw(store word)* și *lw(load word)*, acestea au o semnificație externă, indicând adresa de acces la nivelul memoriei de date.

4.10 NUCLEUL RISC-V

Odata cu modelarea unității de execuție toate componentele microprocesorului sunt finalizate. Prin combinarea unității de comandă cu cea de execuție rezultă un nucleu RISC-V single-cycle. Termenul de single-cycle provine din faptul că procesorul nostru execută o operație numerică sau de transport a datelor într-o perioadă aproximativ egală cu cea a unui ciclu de tact. Teoretic vorbind, perioada acestui ciclu are ca specificație acoperirea duratei celui mai lung timp de execuție acoperit de una dintre instrucțiunile procesorului.

Este important de menționat că dispozitivele de memorare nu fac parte din nucleul propriu-zis, memoria fiind considerată o entitate modulară și mutabilă. Figura 54 prezintă nucleul RISC-V generat de sinteza automată. Alături de cele 2 unități structurale se poate observa prezența memoriei de date și a celei de instrucțiuni.

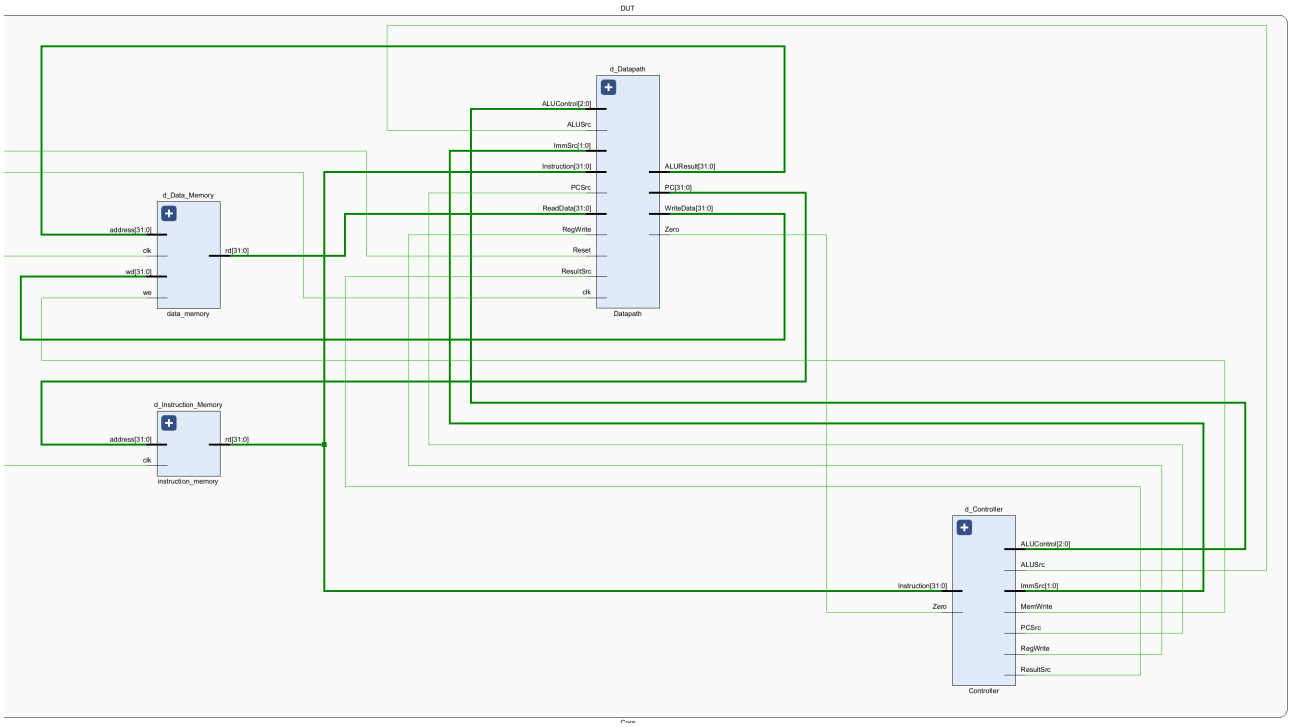

Figura 54: Nucleul RISC-V RV32I

Entitatea VHDL a nucleului este formată în jurul tuturor entităților superioare implementate. Semnalele de comunicare între componentele microprocesorului sunt de o importanță deosebită, acestea fiind practic impulsurile care asigură funcționarea corectă a tuturor modulelor odată cu execuția unei instrucțiuni [1]. O legare inadecvată implică activarea unei suite de module hardware greșite, rezultatul calculelor fiind astfel irelevant și incorect.

De asemenea, semnalele trebuie inițializate prevenind astfel intrarea pe ramuri de execuție încă nedefinite. Spre exemplu, o valoare inițială alta decât 0 poate activa dispozitivele de memorare, scriind astfel date aleatorii la adrese nedestinate scrierii.

Codul sursă VHDL aferent nucleului se regăsește la nivelul Figurii 55.

```vhdl
entity Core is port(
    clk: in std_logic;
    Reset: in std_logic
);
end Core;

architecture arch of Core is
    component Datapath generic(…
    end component;

    component Controller generic(…
    );
    port(…
    );
    end component;

    component instruction_memory generic(…
    );
    port(…
    );
    end component;

    component data_memory generic(…
    );

    port(…
    );
    end component;

    signal sig_PCSrc: std_logic := '0';
    signal sig_ResultSrc: std_logic := '0';
    signal sig_MemWrite: std_logic := '0';
    signal sig_ALUSrc: std_logic := '0';
    signal sig_ImmSrc: std_logic_vector(1 downto 0) := "00";
    signal sig_RegWrite: std_logic := '0';
    signal sig_ALUControl: std_logic_vector(2 downto 0) := "000";

    signal sig_Zero: std_logic := '0';

    signal sig_Instruction: std_logic_vector(31 downto 0):= x"00000000";

    signal sig_ReadData: std_logic_vector(31 downto 0):= x"00000000";
    signal sig_PC: std_logic_vector(31 downto 0):= x"00000000";
    signal sig_ALUResult: std_logic_vector(31 downto 0):= x"00000000";
    signal sig_WriteData: std_logic_vector(31 downto 0):= x"00000000";

begin

    d_Controller: Controller port map(Zero=> sig_Zero, Instruction=> sig_Instruction, PCSrc=> sig_PCSrc,
                                      ResultSrc=> sig_ResultSrc, MemWrite => sig_MemWrite, ALUSrc=> sig_ALUSrc,
                                      ImmSrc=> sig_ImmSrc, RegWrite=> sig_RegWrite, ALUControl=> sig_ALUControl);

    d_Datapath: Datapath port map(clk=>clk, Reset=> Reset, Instruction=> sig_Instruction, ReadData=> sig_ReadData,
                                  PCSrc=> sig_PCSrc, ResultSrc=> sig_ResultSrc, ALUSrc=> sig_ALUSrc, ImmSrc=> sig_ImmSrc,
                                  RegWrite=> sig_RegWrite, ALUControl=> sig_ALUControl, Zero=> sig_Zero, PC=> sig_PC,
                                  ALUResult=> sig_ALUResult, WriteData=> sig_WriteData);

    d_Instruction_Memory: instruction_memory port map(clk=> clk, address=> sig_PC, rd=> sig_Instruction);

    d_Data_Memory: data_memory port map(clk=> clk, we=> sig_MemWrite, address=> sig_ALUResult, wd=> sig_WriteData, rd=> sig_ReadData);
end architecture;
```

Figura 55: Implementare VHDL a nucleului RISC-V

Procesorul single-cycle are ca mare avantaj relativa simplitate a implementării digitale. Dezavantajul unui astfel de procesor este faptul că suntem limitați din punct de vedere al operațiilor pe care le putem executa [1]. Comenzi complexe asupra numerelor precum înmulțirea și împărțirea nu sunt astfel fezabile, alternativa reprezentării acestora fiind adunări, scăderi și aproximări repetate.

În ciuda acestor probleme, procesorul single-cycle este o implementare cu un caracter didactic valoros, existând posibilitatea extinderii sale într-o unitate de execuție multi-cycle cu vaste capabilități și aplicații.

4.11 MEMORIA CACHE

În cazul în care dorim utilizarea unei memorii de date cu latență ridicată a accesului la date, utilizarea unei memorii cache intermediare devine obligatorie în creșterea metricilor de execuție [8]. Astfel, ca posibilă soluție de implementare, se va prezenta procesul de design al unei memorii cache pe decursul acestui capitol. Ca mențiune, cache-ul este de-a dreptul opțional pentru buna funcționare a nucleului singe-cycle, acesta fiind dezvoltat în jurul unor memorii de date și instrucțiuni construite pe baza matricilor de registre.

Precum dictează intuiția, sunt 2 utilizări spre care ne putem îndrepta când vine vorba de folosirea cache-ului în cardul unui sistem de calcul. Primul drum este marcat de utilizarea clasică care implică extragerea blocurilor de date din memoria principală, eliminând latența de acces. A doua cale este marcată de utilizarea cache-ului cu scop în minimizarea latenței de transmitere a instrucțiunilor ce urmează rulate.

Aceste 2 metodologii nu trebuie însă tratate ca fiind mutual exclusive, ambele fiind implementate la nivelul unui procesor, reprezentând un mare salt făcut spre maximizarea randamentului computațional.

Design-ul unui cache începe cu memoriile *CACHE SRAM* și *TAG SRAM*, urmând ca schimbul acestora cu memoria inferioară să fie orchestrat de un *cache controler* [5].

Pentru simplificarea ilustrării conceptelor de design, se va lua în considerare că procesorul, pentru care implementăm acest nivel de cache, are o arhitectura pe 8 biți și un spațiu de adresare al memoriei egal cu 2^{16} adrese byte-addresable. Practic, cuvântul de lucru al microprocesorului de 1 octet este congruent cu octetul dintr-o adresa.

Pentru a determina modul în care biții din adresă sunt împărțiți este nevoie prima dată de definirea mărimii memoriei cache cât și a blocurilor care o compun. Pentru simplificarea calculelor, cache-ul va fi compus din 8 blocuri, fiecare bloc având un număr de 4 octeți.

Prin urmare, câmpului de offset îi vor fi alocați $log_2 4 = 2$ biți, urmând ca câmpul index sa fie compus din $log_2 8 = 3$ biți. Restul de 11 biți vor fi reprezentați de identificatorul unic asociat unei serii de 8 seturi, și anume tag-ul. în Figura 56 se poate vedea clar acest aranjament.

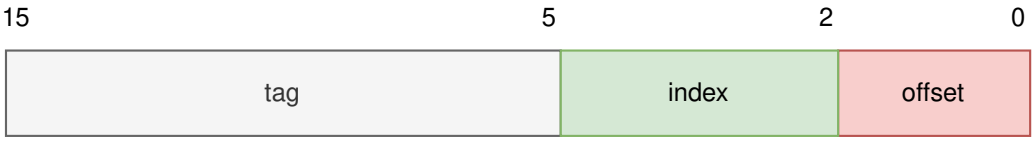

Figura 56: Câmpurile din adresa memoriei

Memoria cache SRAM are ca structura diagrama digitală din Figura 57. Se pot distinge cele 8 registre cu o mărime de 32 de biți care sunt ulterior trecute printr-o serie de 8 buffere urmând ca semnalul de ieșire sa fie ales de multiplexor conform semnalului de selecție controlat de index-ul blocului relevant.

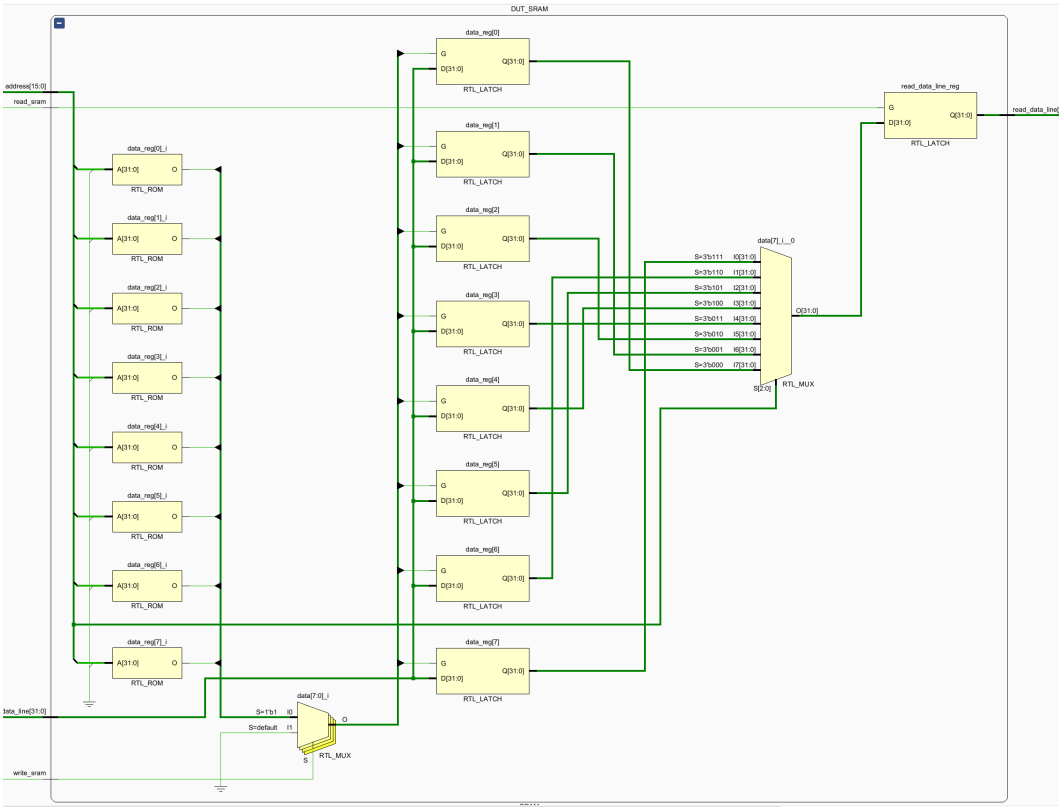

Figura 57: Structura memorie cache SRAM

Codul din spatele acestei generări este prezentat în Figura 58.

```vhdl
entity SRAM is port (
    address: in std_logic_vector(15 downto 0);
    read_sram: in std_logic;
    write_sram: in std_logic;
    write_data_line: in std_logic_vector(31 downto 0);
    read_data_line: out std_logic_vector(31 downto 0)
);
end SRAM;

architecture a_SRAM of SRAM is
    type matrix is array(7 downto 0) of std_logic_vector(31 downto 0);

    signal line_index: integer := 0;
    signal data: matrix := (
        0 => x"00000000",
        1 => x"00000000",
        2 => x"00000000",
        3 => x"00000000",
        4 => x"00000000",
        5 => x"00000000",
        6 => x"00000000",
        7 => x"00000000"
    );

    begin
        line_index <= to_integer(unsigned(address(4 downto 2)));

        EXECUTE:
        process(address, read_sram, write_sram, write_data_line) begin

            if(read_sram = '1') then
                read_data_line <= data(line_index);
            end if;

            if(write_sram = '1') then
                data(line_index) <= write_data_line;
            end if;

        end process;
end a_SRAM;
```

Figura 58: Implementare VHDL a memoriei cache SRAM

În ceea ce privește tag-ul, acesta va fi stocat în memoria tag SRAM a cărei structură este similara cu cea a cache SRAM-ului. Pe lângă câmpul de 10 biți ce reprezintă tag-ul, se mai folosește un bit de validitate, acesta fiind pus pe 1 odată cu aducerea unui bloc in cache. Semnalele de intrare ale acestei componente sunt reprezenta de comanda de citire, prin semnalul *read tag* și de comanda de scriere, prin *write tag*.

Structura memoriei tag SRAM este relatată la nivel digital în Figura 59.

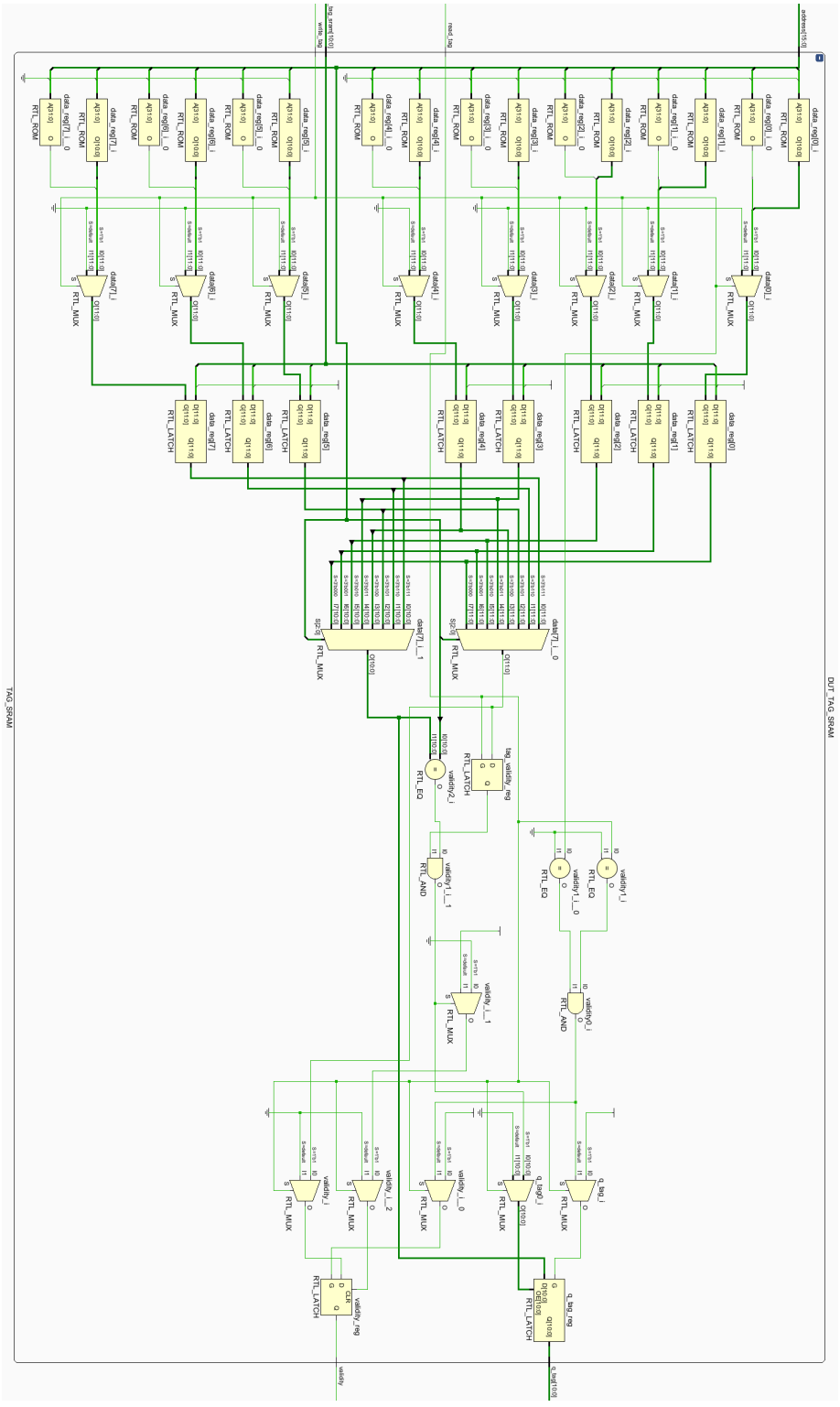

Figura 59: Diagrama digitală a memoriei tag SRAM

Prima jumătate a structurii este similară cu cea a cache SRAM-ului, fiind compusă din registre si elemente de buffer pentru procedura de multiplexare după index. Diferența vine sub forma circuitului de selecție al semnalului de ieșire, acesta fiind condiționat de cele 2 intrări. Prin urmare, semnalul de ieșire este reprezentat de tag-ul de interes. Prezența unui tag valid semnifică existența unui bloc de cache din memoria principală, lipsa tag-ului arată viceversa. Codul acestei implementări se regăsește la nivelul Figurii 60.

```
5  entity TAG_SRAM is port(
6      write_tag: in std_logic;
7      read_tag: in std_logic;
8      address: in std_logic_vector(15 downto 0);
9      write_data_tag_sram: in std_logic_vector(10 downto 0);
10     validity: out std_logic;
11     q_tag: out std_logic_vector(10 downto 0)
12 );
13 end TAG_SRAM;
14
15 architecture a_TAG_SRAM of TAG_SRAM is
16     type matrix is array(7 downto 0) of std_logic_vector(11 downto 0);
17
18     signal data: matrix := (
19         0 => "000000000000",
20         1 => "000000000000",
21         2 => "000000000000",
22         3 => "000000000000",
23         4 => "000000000000",
24         5 => "000000000000",
25         6 => "000000000000",
26         7 => "000000000000"
27     );
28     signal tag_from_address: std_logic_vector(10 downto 0);
29     signal index: integer;
30     signal tag_validity: std_logic;
31
32     begin
33     tag_from_address <= address(15 downto 5);
34     index <= to_integer(unsigned(address(4 downto 2)));
35     LOGIC:
36     process(write_tag, read_tag, write_data_tag_sram) begin
37
38         if(read_tag = '0' and write_tag = '0') then
39             validity <= '0';
40             q_tag <= "ZZZZZZZZZZZ";
41         end if;
42
43         if(read_tag = '1') then
44             tag_validity <= data(index)(11);
45             if(tag_from_address = data(index)(10 downto 0) and tag_validity = '1') then
46                 validity <= data(index)(11);
47                 q_tag <= data(index)(10 downto 0);
48             else
49                 validity <= '0';
50                 q_tag <= "ZZZZZZZZZZZ";
51             end if;
52         end if;
53
54         if(write_tag = '1') then
55             data(index)(10 downto 0) <= write_data_tag_sram;
56             data(index)(11) <= '1';
57         end if;
58
59     end process;
60 end architecture;
```

Figura 60: Implementare VHDL a memoriei tag SRAM

Memoria RAM principală va avea rolul de a gestiona trimiterea blocurilor de 4 octeți spre unitatea de comandă a cache-ului. Prim urmare, în ciuda caracterului byte-addresable, cuvântul transmis în exterior va fi de 4 octeți [5]. Fiecare combinație unică de tag și index va determina accesarea unui astfel de cuvânt. Scrierea pe de altă parte se va face la nivel de byte. Arhitectura generată de sinteza automată este similară cu cea prezentată în 38. Codul VHDL al acestei entități este prezentat în Figura 61.

```
entity RAM_64KB is port(
    address: in std_logic_vector(15 downto 0);
    data_dram_out: out std_logic_vector(31 downto 0);
    data_dram_in: in std_logic_vector(31 downto 0);
    read_ram: in std_logic;
    write_ram: in std_Logic;
    ready_ram: out std_logic;
    clk: in std_logic;
    reset: in std_logic
);
end RAM_64KB;

architecture a_RAM_64KB of RAM_64KB is
    type matrix is array(127 downto 0) of std_logic_vector(7 downto 0);

    signal data: matrix := (
        0 => x"01",
        1 => x"02",
        2 => x"03",
        3 => x"04",
        4=> x"f0",
        5=> x"f1",
        6=> x"f2",
        7=> x"f3",
--        1996 => x"a0",
--        1997 => x"b0",
--        1998 => x"c0",
--        1999 => x"d0",
        others => x"00"
    );
    signal tagline: std_logic_vector(13 downto 0);
    signal address_int: integer;

    begin
        tagline <= address(15 downto 2);

    process(address, data_dram_in, clk, reset, read_ram, write_ram)
        variable concatenated_address: std_logic_vector(15 downto 0) := x"0000";
        variable current_block: std_logic_vector(1 downto 0) := "00";
        variable memory_block: std_logic_vector(31 downto 0);
        variable index: integer := 0;
        begin
            if(rising_edge(clk)) then
                if(reset /= '1') then
                    if(read_ram = '1') then
                        for i in 0 to 3 loop
                            concatenated_address := tagline & current_block;
                            index := to_integer(unsigned(concatenated_address));

                            memory_block((8 * (i+1) -1) downto 8 * i) := data(index);
                            current_block := std_logic_vector((unsigned(current_block)) + 1);
                        end loop;
                            data_dram_out <= memory_block;
                            ready_ram <= '1';
                            current_block := "00";
                            memory_block := x"00000000";
                        else
                            ready_ram <= '0';
                            data_dram_out <= x"00000000";
                        end if;
                    if(write_ram = '1') then
                            ready_ram <= '1';
                            index := to_integer(unsigned(address));

                            data(index) <= data_dram_in(7 downto 0);

                            current_block := "00";
                            memory_block := x"00000000";
                    end if;
                    if(read_ram = '0' and write_ram = '0') then
                        data_dram_out <= x"00000000";
                        ready_ram <= '0';
                    end if;
                else
                    data <= (others => (others=>'0'));
                end if;
            end if;
    end process;
end architecture;
```

Figura 61: Implementare VHDL a memoriei RAM

Unitatea de comandă a cache-ului va fi modelată ca un automat cu stări finite, astfel ilustrând în cel mai mare detaliu modul acesteia de funcționare [5]. Odată cu implementarea controlerului, trebuie să decidem asupra modului în care cache-ul nostru va gestiona cererile de scriere și citire.

Pentru a ilustra conceptul și a reduce complexitatea procedeului de stocare, memoria cache va implementa, prin intermediul controlerului, politicile write through și no write allocate.

Starea inițială este reprezentată de *idle*. De asemenea, în urma îndeplinirii cererii procesorului, automatul memoriei cache se va întoarce tot în această stare.

În funcție de operați cerută, fie acesta de citire sau de scriere, se poate intra în starea *read prepare* respectiv *write prepare*. Cele 2 stări pregătitoare au rolul de a seta semnalele de acces la nivelul tag SRAM-ului. Accesul la nivelul componentelor necesită pregătire în prealabil, astfel mitigând întârzierea semnalului din pricina propagării la nivelul bistabilelor.

Odată ce tag-ul a fost citit, se analizează structura acestuia. Dacă tag-ul este valid și inițializat, se continuă pe ramurile de *read hit*, respectiv *write hit*. Dacă tag-ul rezultat este invalid, controlerul intră pe ramurile *read miss* sau *write miss*, în funcție de cererea microprocesorului.

Tranziția în starea *write miss* conduce la resetarea semnalelor de acces la nivelul modulelor inferioare și trecerea controlerului în starea inițială *idle*.

În cazul unui *read miss*, cache-ul nostru extrage blocul relevant din memoria principală, trecând pe rând în stările *upate data read, finish update* și într-un final întorcându-se în *idle*. Starea *update data read* are rolul de a comunica cu memoria, primind de la aceasta bocul de 4 octeți. Tot la nivelul acestei stări sunt scrise în tag SRAM și cache SRAM datele cerute. Starea intermediară *finish update* are rol de reinițializare a semnalelor de interfață cu memoriile cache-ului.

După tratarea cererii de *write miss*, dacă microprocesorul dorește accesul la o adresă de memorie identificată cu același tag și index găsit la nivelul unei adrese extrase anterior, se va continua în starea *get cache data* a ramurii *read hit*. La nivelul acestei stări se va trimite spre procesor octetul cerut, prin extragerea sa din blocul de cache accesat. În urma transmiterii datelor, se va trece din nou în starea de *idle*.

Cererea de scriere într-un bloc care este prezent în cache implică prima dată scrierea datelor în memoria principală, la nivelul stării *write hit*. Odată ce memoria a semnalat încheierea ciclului de scriere, se va trece în starea *get memory data*, blocul cu datele acum modificate fiind transmis memoriei cache SRAM. La fel ca în cazul ramurii de *read miss*, ciclul de scriere va continua cu starea *finish update*, după resetarea semnalelor de acces, urmând tranziția în starea inițială.

Stările enumerate și anterior explicate, precum și semnalele care determină tranziția lor, se pot găsi la nivelul diagramei automatului cu stări finite din Figura 62.

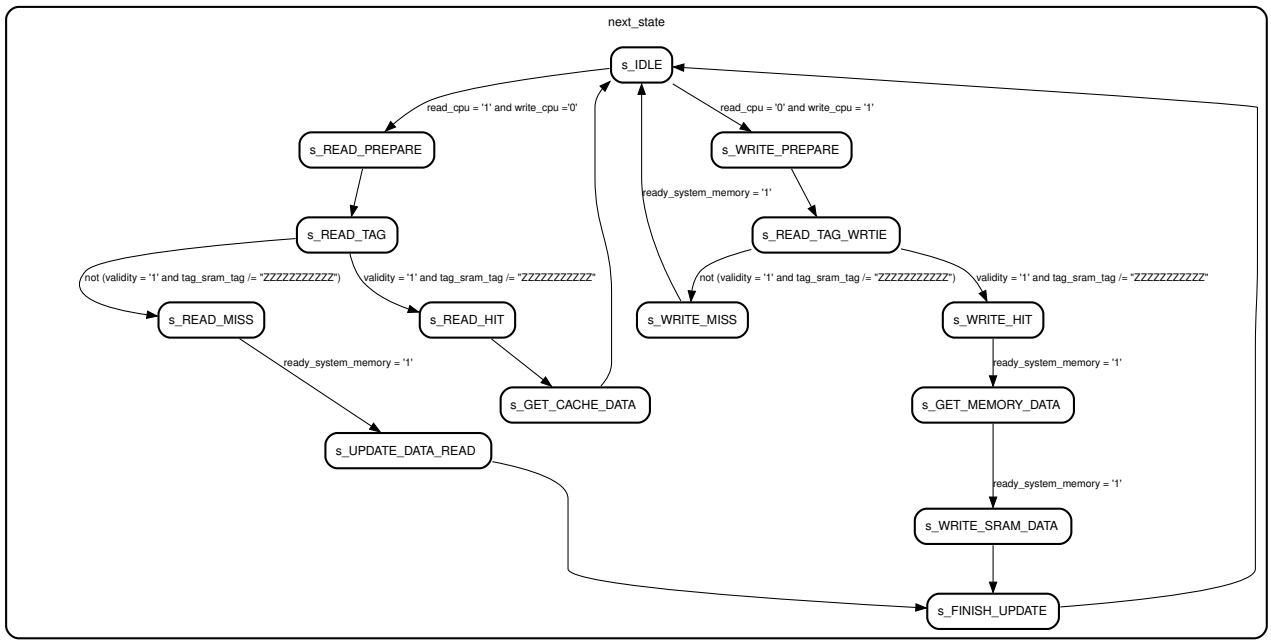

Figura 62: Diagrama de stări a cache controlerului

Prin comasarea celor 3 entități descrise, tag SRAM, cache SRAM și nu în ultimul rând cache controler, rezultă memoria cache L1, a cărei diagramă digitală poate fi consultată în Figura 63. Se poate observa și prezența memoriei inferioare la nivelul acestei entități, însă, aceasta nu trebuie considerată ca făcând parte din cache, fiind doar un dispozitiv de memorare terț.

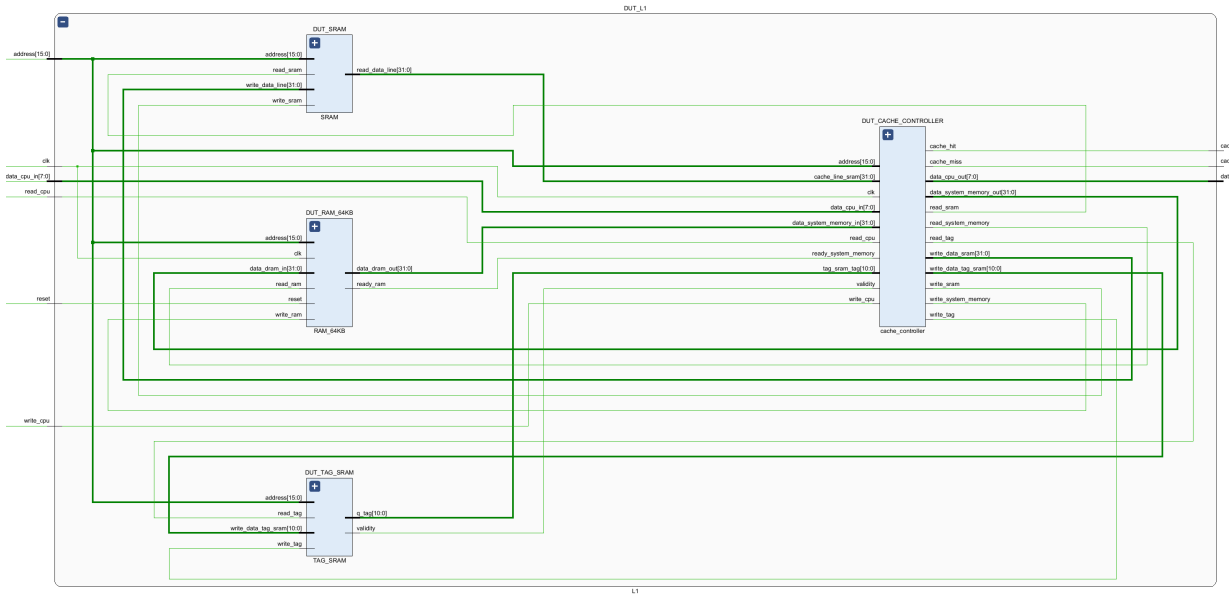

Figura 63: Diagrama digitală a memoriei cache L1

5 TESTAREA ȘI VALIDAREA ENTITĂȚILOR

Pentru a exemplifica buna funcționare a entităților modelate, se vor implementa entități de testare VHDL, numite *testbench* [8]. În cadrul acestor entități de test se vor analiza doar dispozitivele finale, și anume nucleul single-cycle RISC-V și memoria cache L1. La nivelul testbench-urilor ce urmează prezentate se vor evidenția cazuri de utilizare cheie care ilustrează scenarii de execuție comun întâlnite în practică.

5.1 TESTAREA NUCLEULUI RISC-V

Se vor ilustra instrucțiunile sw(store word), lw(load word) cât și una dintre instrucțiunile de calcul aritmetico-logic, precum add, sub și and. Exemplificarea tuturor instrucțiunilor aritmetice este exagerată, acestea funcționând într-un mod similar, cu excepția cuvântului de comandă transmis unității aritmetice și logice. Prima instrucțiune pe care o vom testa este store word. Cuvântul de comanda utilizat este prezentat în Figura 64.

```
Limbaj de asamblare =   sw x6, 5(x5)
          Cod Masină    0000 0000 0110 0010 1010 0010 1010 0011
       Hexadecimal =    0x0062a2a3
            Format =    S-type
 Set de instrucțiuni    RV32I
```

Figura 64: Exemplu de utilizare al instrucțiunii store word

Conform acestui cuvânt, se va stoca informația din cadrul registrului x6 într-o adresă de memorie egală cu valoarea imediată 5 însumată la valoarea din registrul x5. Execuția instrucțiunii va începe în momentul în care se acționează semnalul de reset asupra microprocesorului. Odată cu această tranziție, progam counter-ul începe numărătoarea ciclică care determină execuția secvențială a instrucțiunilor.

Prima comandă fiind cea prezentată anterior, decodificatorul de instrucțiuni semnalează nevoie de scriere a datelor la nivelul memoriei inferioare, astfel fiind setat semnalul *MemWrite*. În paralel sunt extrase datele din registrele x5 și x6. Cuvântul registrului x5 va intra împreuna cu valoarea imediată rezultată extinderii de semn în unitatea aritmetică și logică pentru efectuarea calculului adresei. Valorea din registrul x6 va fi preluată de pinul de scriere a datelor la nivelul memoriei principale. Odată cu începerea următoarei perioade a ciclului de tact, datele din registru vor fi scrise în adresa cerută de rezultatul însumării efectuate de ALU.

În Figura 65 se poate observa forma de undă generată de acest exemplu. Se pot, de asemenea, distinge semnalele intermediare de decodare atât ale decodificatorului principal cât și al decodificatorului secundar.

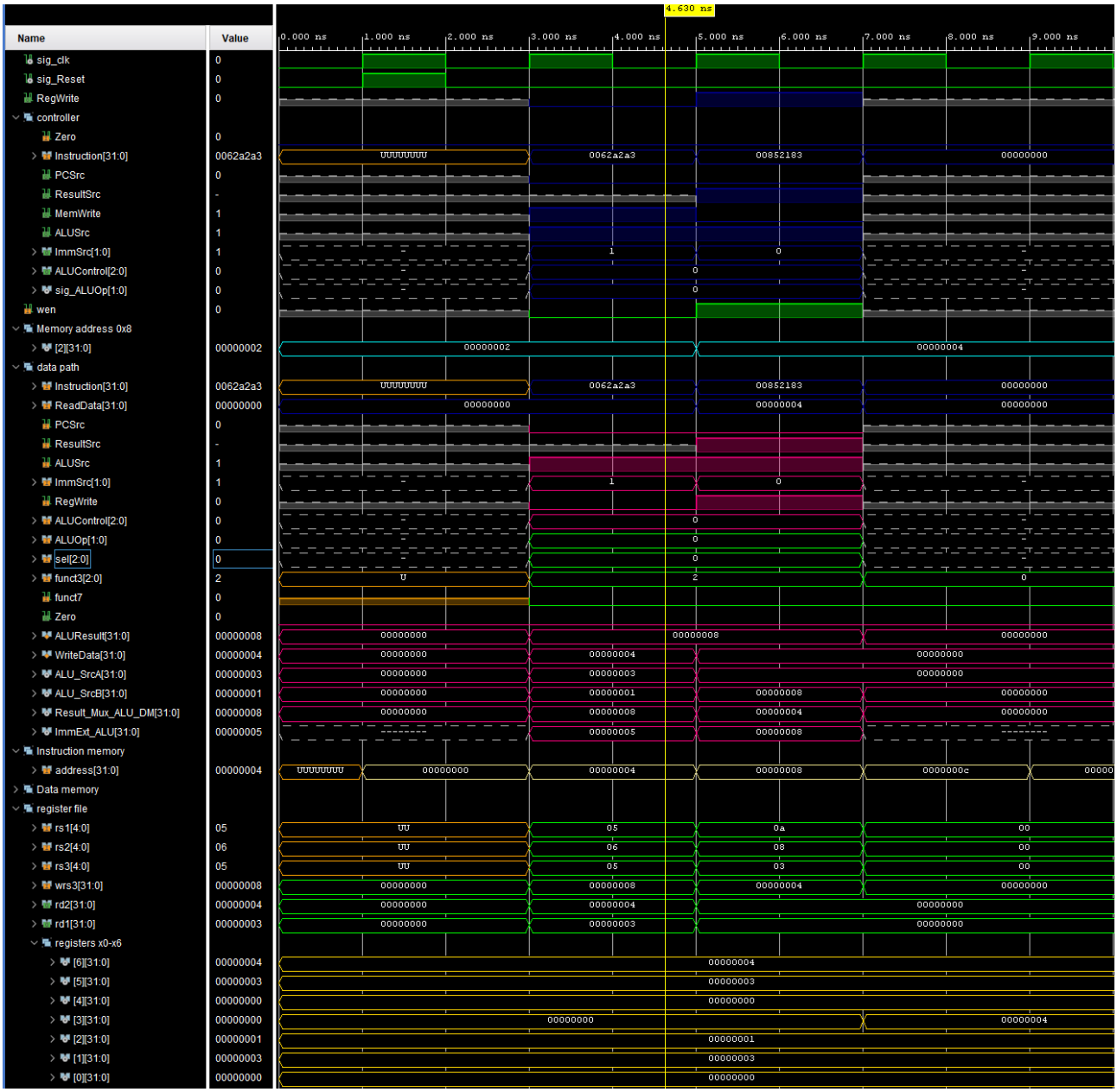

Figura 65: Forma de undă a testbench-ului instrucțiunii sw

Cuvântul 0x4 din registrul x6 este scris la nivelul adresei de memorie 0x08. De menționat faptul că această adresă se referă la al treilea spațiu de adresare din memorie de date, aceasta fiind byte-adressable. Foarte importantă este faptul că scrierea propriu zisă se efectuează doar în momentul în care se ajunge pe frontul crescător al următorului ciclu de tact.

A doua instrucțiune pe care o vom analiza este cea de accesare a cuvintelor din memoria de date, și anume, lw(load word). Aceasta este similară cu instrucțiunea prezentată anterior, datele provenite dintr-o adresă a memoriei, adresă calculată ca suma unui cuvânt din registrul comandat cu o valoare imediată furnizată, sunt stocate în cel de al doilea registru extras din câmpurile instrucțiunii. Formatul unei astfel de comenzi poate fi analizat în Figura 66.

```
Limbaj de asamblare =   lw x3, 8(x10)

          Cod Masină    0000 0000 1000 0101 0010 0001 1000 0011

        Hexadecimal =   0x00852183

             Format =   I-type

 Set de instrucțiuni    RV32I
```

Figura 66: Exemplu de utilizare al instrucțiunii sw(store word)

Acest cuvânt de comandă dorește stocarea informației din adresa de memorie calculată ca valoarea imediată 0x8 adunată la valoarea registrului x10 în registrul x3. Forma de undă a testbench-ului care prezintă efectuarea corectă a acestei instrucțiuni se poate consulta în Figura 67.

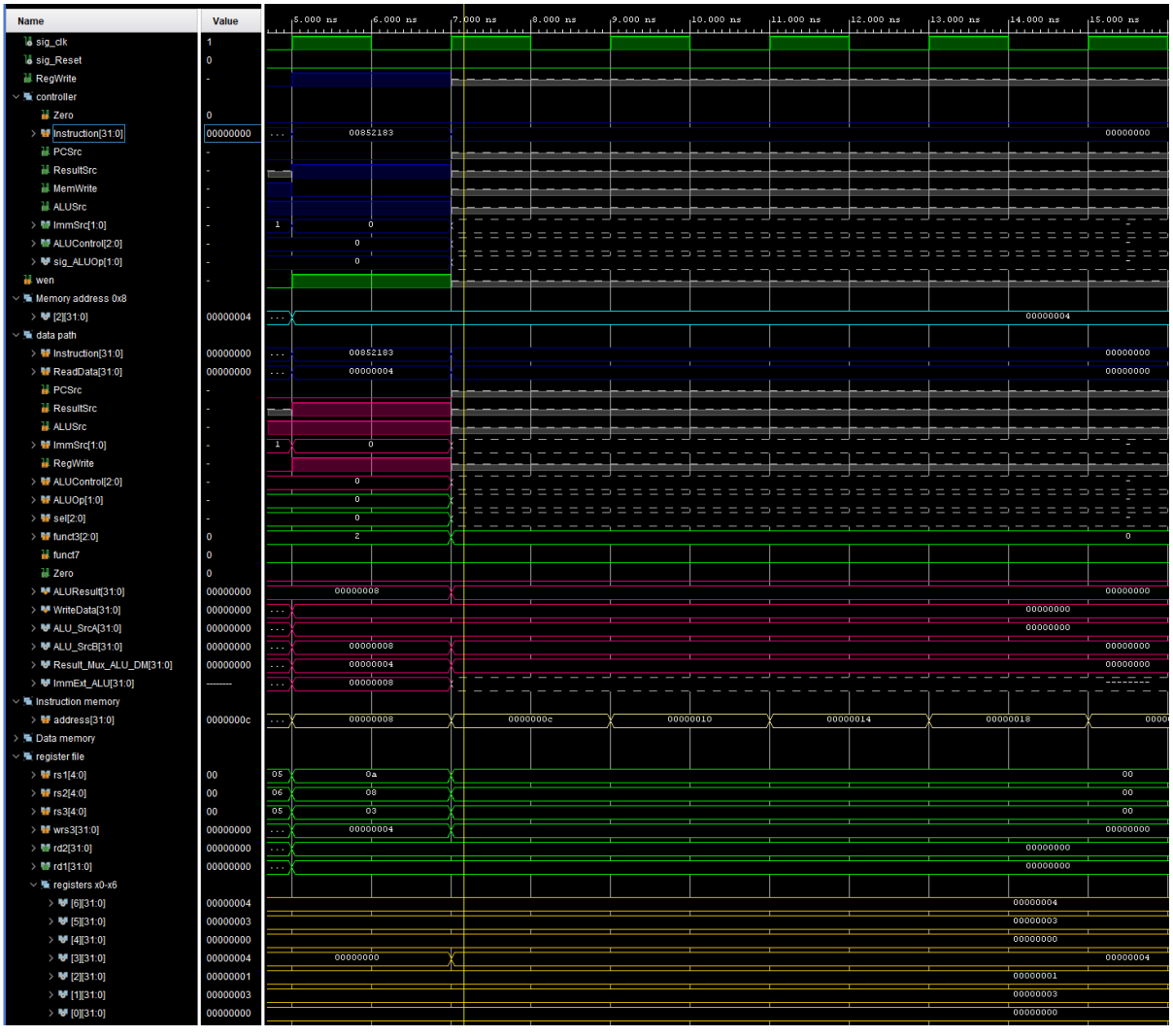

Figura 67: Forma de undă a testbench-ului instrucțiunii sw

Se poate observa marea notă de similaritate cu instrucțiunea lw(load word) anterior prezentată. La nivel funcționat, acestea două sunt similare, singura diferență fiind polaritatea de acces a memoriei, o instrucțiune realizând scrierea, cealaltă efectuând citirea și încărcarea în registre.

Ultimul exemplu de utilizare al instrucțiunilor va fi comanda de însumare a conținutului din 2 registre și plasarea rezultatului într-unul de al treilea. Toate instrucțiunile care efectuează calcule asupra numerelor cu datele din interiorul registrelor se comportă identic, singura diferență fiind cuvântul pe care decodificatorul aritmetic îl trimite unității aritmetice și logice, conform operației de executat. Instrucțiunea care urmează a fi executată este prezentă în Figura 68. Forma de undă și modul în care execuția decurge este prezentat în Figura 69.

```
Limbaj de asamblare =   add x7, x6, x5

          Cod Masină    0000 0000 0101 0011 0000 0011 1011 0011

       Hexadecimal =    0x005303b3

            Format =    R-type

 Set de instrucțiuni    RV32I
```

Figura 68: Exemplu de utilizare al instrucțiunii add

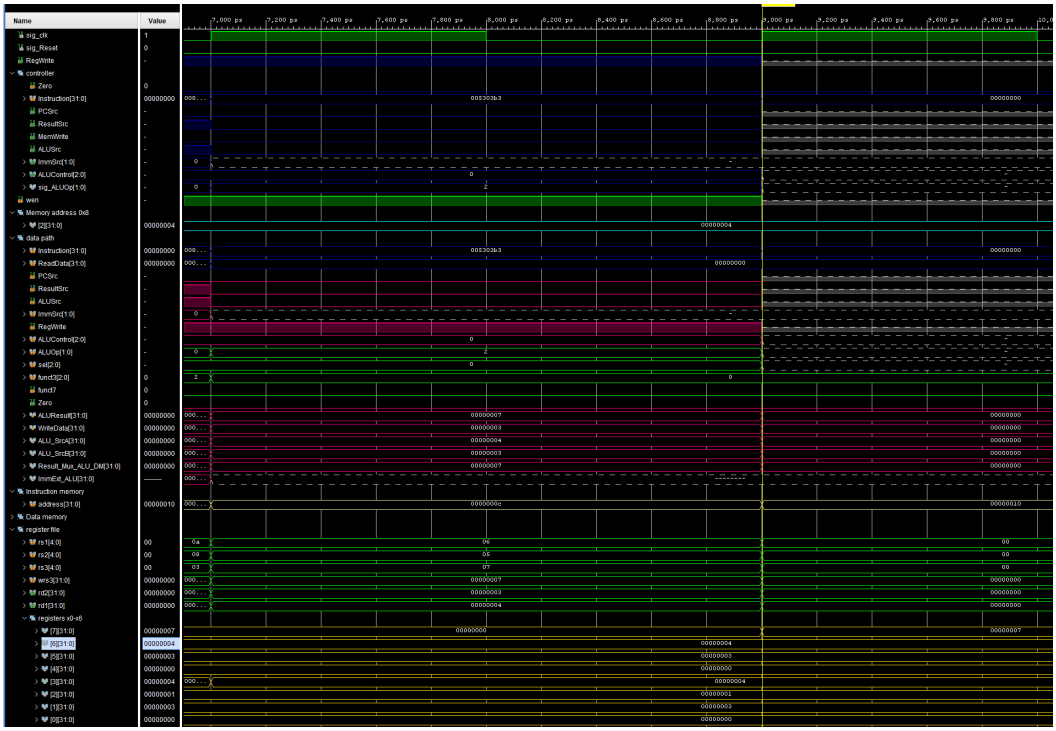

Figura 69: Forma de undă a testbench-ului instrucțiunii add

Cum se poate vedea, valorile din registrele x6 și x5 sunt extrase și trimite unității aritmetice și logice pentru a efectua operația de adunare. În urma însumării, rezultatul este încărcat în registrul x7. Sunt notabile semnalele produse de decodificatorul aritmetic, acesta fiind cea mai relevantă componentă de diferențiere a operațiilor, în cazul de execuție al comenzilor aritmetice.

5.2 TESTAREA MEMORIEI CACHE

Pentru ilustrarea bunei funcționări a memoriei cache, se vor analiza mai multe scenarii de utilizare. Va fi prezentat succesiv modul în care cache-ul raspunde cererilor de scriere și citire, atât în cazul unui răspuns de tipul cache miss cât și de cache hit.

Figura 70 prezintă un exemplu ce cache hit, la prima accesare a cache-ului. Procesorul cere datele de la adresa 0x0000, însă, acestea nefiind valide și prezente în cache, se returnează un cache miss de către unitatea de comandă.

Astfel, utilitatea cache-ului este mai mult decât evidentă, în perioada pierdută cu tratarea a unui miss fiind posibile cel puțin 3 accesări la nivelul cache SRAM-ului. Eficiența acestuia poate fi ridicată prin alterarea design-ului digital și a modului de tratare a diferitelor cereri ale procesorului.

Figura 70: Forma de undă a testbench-ului unui cache miss

Se poate observa latența mare de acces în cazul unui miss, semnalul cache miss rămânând setat pentru numeroase cicluri de tact, suprapunându-se cu extragerea unui bloc în cache.

Figura 71 ne prezintă ce se va întâmpla în cazul unei cereri de acces la nivelul adreselor din același bloc, acestea aflându-se deja în cache, grație miss-ului anterior. De această dată, datele vor fi trimise procesorului mult mai rapid, în doar 4 cicluri de tact. Prin comparație, în cazul unui cache miss, au fost nevoie de mult mai multe perioade ale tactului, și anume 14, pentru a trata accesul.

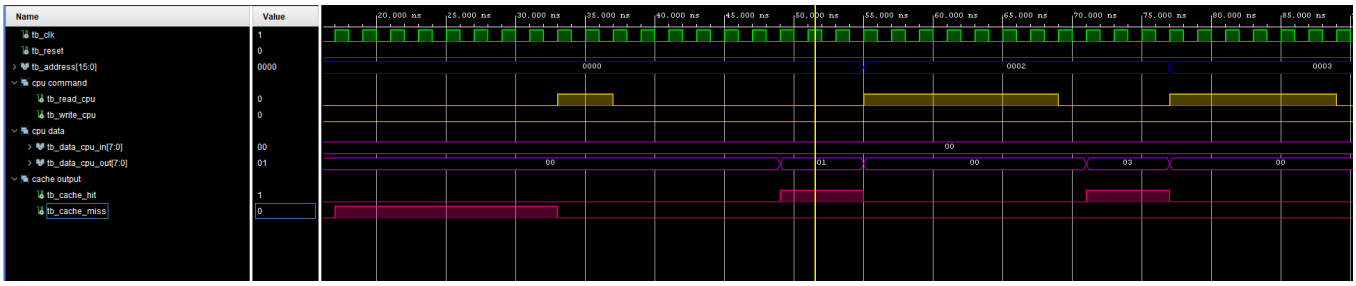

Figura 71: Forma de undă a testbench-ului unui cache read hit

Următoarea operație pe care o vom trata este cache write-ul. În cazul unui hit, datele vor si scrise atât la nivelul memoriei inferioare cât și la nivelul cache SRAM-ului. În exemplul prezentat prin Figura 72 se va scrie un nou octet la adresa 0x000, acesta fiind pus pe valoarea 0x2. Pentru a testa că scrierea într-adevăr a funcționat corect, se va efectua o cerere de citire de la aceeași adresă. După cum se poate vedea, rezultatul este cel corect, cache SRAM-ul transmițând procesorului valoarea scrisă anterior, 0x2.

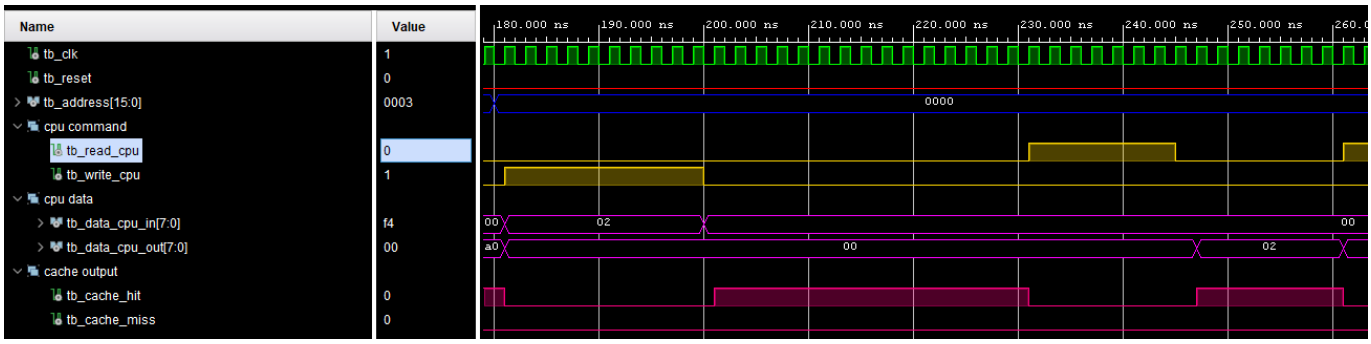

Figura 72: Forma de undă a testbench-ului unui cache write hit

Dacă se dorește scrierea la nivelul unei adrese care nu se găsește în cache, rezultatul va fi unul de write miss, datele urmând sa fie scrise doar în memoria inferioară. Foarte important este faptul că acestea adrese nu vor fi extrase din memoria inferioară în cache.

Figura 73 prezintă o cere de scriere la adresa 0x1FA. Datele sunt scrise, însă, nu sunt puse în cache. Următoarea cerere de citire trage datele în cache, rezultând tot într-un cache miss. Ultima operație de citire de la această adresă trimite procesorului printr-un read hit, datele scrise anterior prin operație de write miss. Este notabilă corectitudinea datelor extrase și sincronizarea cu memoria inferioară.

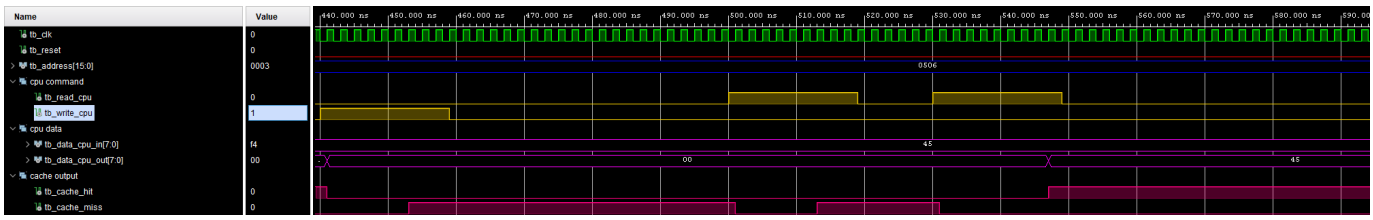

Figura 73: Forma de undă a testbench-ului unui cache write miss, urmat de 2 cereri de read

6 CONCLUZII

6.1 CONCLUZII

Prin cadrul acestei lucrări se realizează, în primul rând, studiul elementelor de logică digitală necesare implementării unui sistem computațional. O notă importantă este faptul că limbajul de programare VHDL se consideră familiar, nefiind studiat în cadrul lucrării. În urma enumerării elementelor teoretice sunt implementate componentele de bază digitale ale unui procesor, urmând ca acestea să fie interconectate conform respectării specificației RISC-V.

Implementarea microprocesorului vizează doar un subset de instrucțiuni ale acestei specificații, implementarea celor rămase fiind posibilă prin extensia decodificatorului principal.

De asemenea, se ilustrează problema care face din memoria cache o entitate digitală de nelipsit din cadrul design-ului unui procesor. Această lucrare prezintă prin urmare o implementare simplificată a unui modul cache superior L1.

6.2 PERSPECTIVE DE DEZVOLTARE

Deși obiectivul final al lucrării a fost atins, există numeroase direcții spre care o viitoare îmbunătățire a sistemului se poate îndrepta, printre acestea numărându-se următoarele:

- Transformarea procesorului dintr-unul single-cycle în unul multi-cycle, fiind astfel posibilă implementarea unor instrucțiuni aritmetice mai complexe, precum înmulțirea numerelor.
- Îmbunătățirea modulului de decodificare, crescând capabilitățile de execuție ale nucleului prezentat în lucrare.
- Utilizarea unui hardware mai eficient în cadrul unității aritmetice și logice, percum un lookahead carry adder.
- Modificarea arhitecturii memoriei cache în una mai apropiată de implementările moderne, cum ar fi un cache total asociativ ce utilizează un algoritm LRU.
- Implementarea unui procesor cu mai multe nuclee RISC-V, acest lucru implicând utilizarea protocoalelor de coerență la nivelul memoriei și a celor de sincronizare la nivelul nucleelor.

BIBLIOGRAFIE

[1] S. Harris, D. Harris, Digital Design and Computer Architecture: RISC-V Edition, Morgan Kaufmann, 2021.

[2] C. Petzold, Code: The Hidden Language of Computer Hardware and Software, Microsoft Press, 2000

[3] D. A. Patterson, J. L. Hennessy, Computer Organization and Design RISC-V Edition: The Hardware Software Interface, Morgan Kaufmann, 2017.

[4] *** https://drive.google.com/file/d/1uviu1nH-tScFfgrovvFCrj7Omv8tFtkp/view

[5] H. Gurban, Laboratorul 7 sisteme multiprocesor, Facultatea de Automatică și Calculatoare, Universitatea Politehnică Timișoara, 2023.

[6] M. Ciletti, Advanced Digital Design with the Verilog HDL, ediția a 2-a, Prentice Hall, 2010.

[7] M. Ercegovac, and T. Lang, Digital Arithmetic, Morgan Kaufmann, 2003.

[8] J. Handy, Cache Memory Book, ediția a 2-a, Morgan Kaufmann, 1998.