Microprocesorul RISC-V

Canditat: Dan-Alexandru Bulzan

Coordonator științific: Ș.l.dr.ing Eugen-Horațiu Gurban

Sesiune: Iunie 2024

1 INTRODUCERE

1.1 SCOPUL ȘI MOTIVAȚIA LUCRĂRII

Implementarea personală a setului de instrucțiuni RISC-V, s-a născut din dorința de a realiza ceea ce poate fi considerat nimic mai puțin decât un apogeu al metodelor științifice din ultimul secol, și anume, procesorul.

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 usor sa ne pierdem în complexitățiile 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.

Implementarea va fi realizată în limbajul de descriere hardware VHDL, entitățile urmând să fie simulate prin intermediul Vivado, soluție de design și sinteză hardware oferită de Advanced Micro Devices.

DE CONTINUAT

2 STUDIU BIBLIOGRAFIC

2.1 ARHITECTURA RISC

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

Acronimul RISC, face referintă la *reduced instruction set computer* sau calculator cu set de instrucțiuni reduse. Un microprocesor care implementează o astfel de filozofie, utilizează un set de instructiuni 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 numar 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 lucrari, RISC-V.

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 facându-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 supercomputerului 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.

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.

2.4 IMPLEMENTĂRI RISC-V

Datorită proliferării lipsite de licentă 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 cunoștiintelor despre organizarea calculatoarelor, obișnuiesc să prezinte ca suport didactic implementări succinte ale unui nucleu RISC-V. 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 intregimea 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.

DE CONTINUAT

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.

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, detalile de la un anumit nivel logic al unui sistem, sunt redate sumar și considerate ca atare de catre 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 interacționeaza 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, microarhitectura 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 este modul corect de reprezentare*; *cum asigurăm coerența datelor cu analizarea acestora de către sistemul de calcul*; *cum ne asigurăm ca datele indiferent formatului ligibil uman, nu sunt iligibile 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 logico-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 radix-ul unui sistem de numerație o poate lua este *1*, corespunzând unui sistem cu un singur simbol, fiecare număr continând *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 hexadecimală și cea binară. În Tabela 1 se pot observa bazele anterior menționate, însoțite de simbolurile aferente cât și de un exemplu reprezentativ.

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

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

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 defineste acest aspect este redată prin formula 2.

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

3.2.1 NUMERELE BINARE

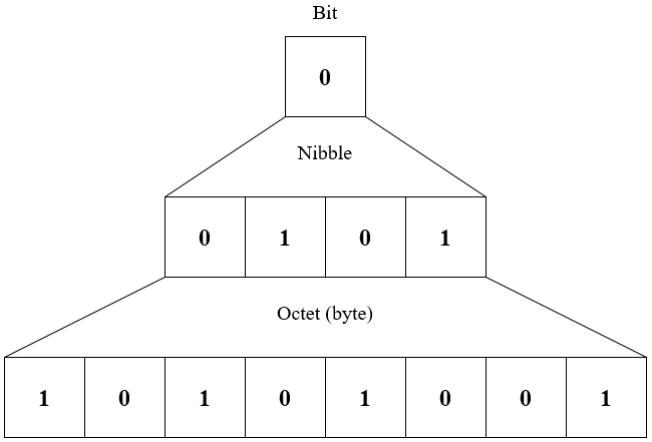
Modul de funcționare a dispozitivelor digitale este constituit pe oscilațiile rapide ale semnalelor electrice, semnale a căror valori se identifică cu unul dintre membri faimosului cuplu binar, *0 și 1*. Prin urmare, datele vor avea o reprezentare care utilizează radixul binar.

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*.

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

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



Octet-ul va reprezenta unitatea fundamentală și indivizibilă pentru microarhitectura 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 realația dintre magnitudinea binară (număul de biți folosiți în reprezentare) și cantitata de date reprezentate prin plaja de valori adiacentă, ignorând existenta numerelor negative.

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

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}	$0 \leq n \leq 2^{64} - 1$

3.2.2 BAZA HEXADECIMALĂ

Datorită clarificării reprezentării datelor în radix-ul binar, întelegerea bazei hexadecimale va fi cu atât mai simplă. Convertirea unui număr din binar in hexadecimal 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 hexadecimală.

Tabela 3: Echivalența nibble - simbol hexadecimal

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

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.

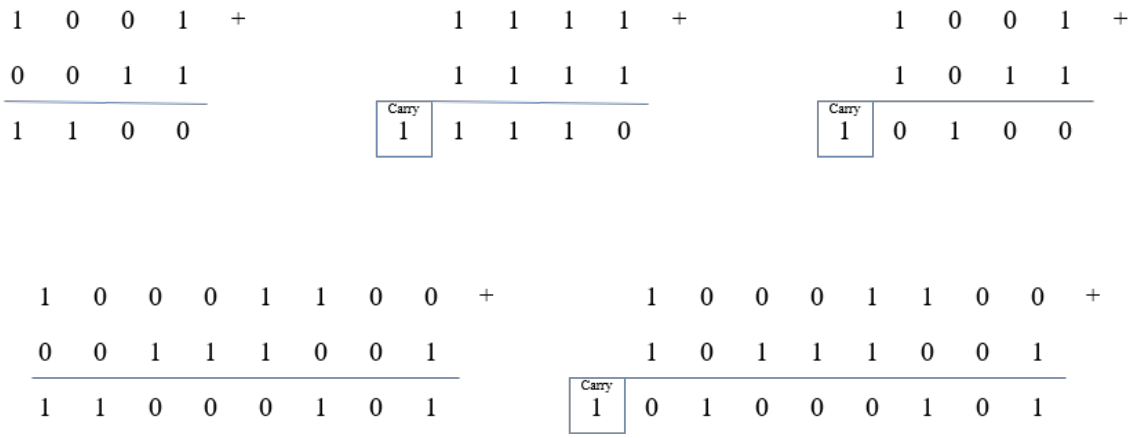
Cea mai elementară operație matematică care poate fi aplicată unui număr este adunarea. Însumarea numerelor este cu atât mai facilă cu cât numărul de simboluri folosite în reprezentarea acestora scade. Spre norocul lumii digitale, radix-ul utilizat permite efectuarea operațiilor matematice în cele mai simple metode. Există doar 4 operații fundamentale posibile, acestea fiind prezentate în Tabela 3.

Tabela 4: Operațiile fundamentale de însumare a numerelor binare

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

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, acestă 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 positive sunt adunate fiind acum clarificat, următoara 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.

Primul mod de reprezentare este cel prin semn-magnitudine. 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 4 prezintă aplicarea acestei reprezentări asupra numerelor pe 8 biți.

Tabela 5: Reprezentarea prin semn-magnitudine

Valoare binară	Semn magnitudine	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

Se pot astfel distinge următoarele lucruri:

- Există două reprezentări posibile pentru 0, și anume ± 0.
- Deși se acopera 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-magnitudine 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 5 conține reprezentările numerelor de 8 biți.

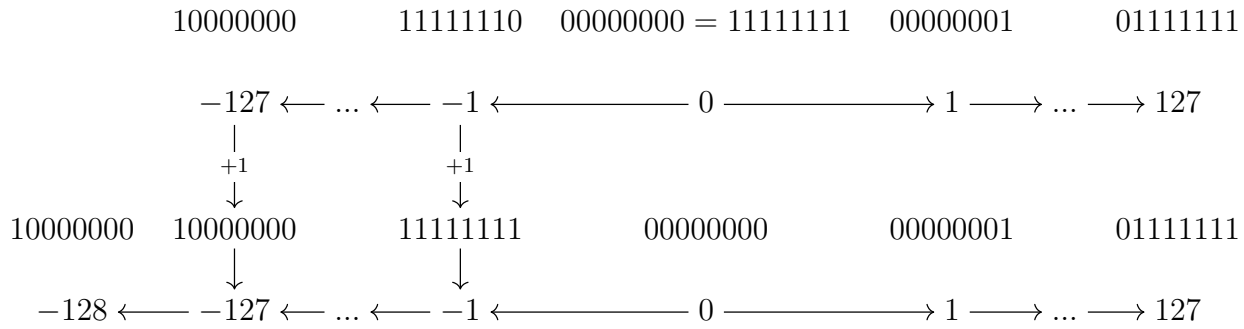
Tabela 6: Reprezentarea prin complement de 1

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

La fel ca în cazul reprezentării prin semn-magnitudine, 0 dorește să respecte principiul superpoziției, diferența pricipală 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. 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*). Subtilitățile de funcționare ale unui tranzistor sunt dincolo de scopul acestei lucrări, acesta fiind considerat cutia neagră de la baza implementării entităților digitale.

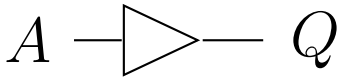
O poartă logică are rolul de a implementa o funcție matematică a algebrei Booleane. Comportamentul acestora este considerat ideal, întârzierile de propagare a impulsului astfel neglijabile. Pentru analiza funcției algebrice descrise se vor utiliza tabele logice,a căror rol este de a relaționa în format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

3.3.1 BUFFER

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

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 NOT

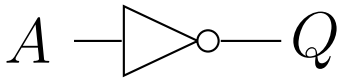
De asemenea cunoscut sub numele de *inversor*, rolul acestei porți logice este de a schimba polaritatea semnalului primit.

Tabela 7: Funcția logică NOT

A	Q
1	0
0	1

Tabela 7 prezintă functionalitatea 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 SAU

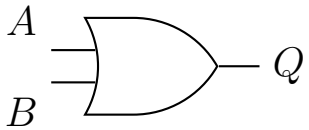
Poarta logică *sau* reprezintă implementarea operației de disjuncție matematică, a cărei rezultat este 0 doar atunci când ambele intrări logice sunt 0. Tabela 8 prezintă valorile operației raportate la intrările logice. Expresia matematică a operației *sau* este $Q = A + B$.

Tabela 8: Funcția logică SAU

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

Simbolul utilizat în diagramele circuitelor este prezentat prin Figura 6.

Figura 6: Simbolul porții logice SAU



3.3.4 ȘI

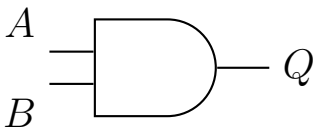
Poarta logică *și* reprezintă implementarea operației de conjuncție matematică, a cărei rezultat este 1 doar atunci când ambele intrări logice sunt 1. Tabela 9 prezintă valorile operației raportate la intrările logice. Expresia matematică a operației *și* este $Q = A \cdot B$.

Simbolul utilizat în diagramele circuitelor este prezentat prin Figura 7.

Tabela 9: Funcția logică ȘI

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

Figura 7: Simbolul porții logice ȘI



3.3.5 XOR

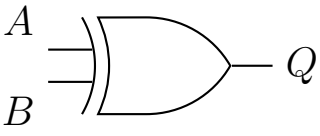
Poarta logică *xor*, de asemenea cunoscută ca *SAU exclusiv* setează semnalul de output pe 1 logic doar în cazul în care cel mult una dintre intrări este activă. Comportamentul acesteia este cel mai bine reprezentat de Tabela 10. Expresia matematică care descrie această poartă este $Q = A \oplus B$.

Tabela 10: Funcția logică XOR

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

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 NAND

Ca o regulă generala, totalitatea porților logice la ale căror denumire se atașează prefixul *N-* sunt varianta negată a portii logice originale. Însă, spre deosebire de restul porților, NAND este de asemenea cunoscută ca și poarta logică fundamentală. Această denumire reiese din operațiile algebrei Booleane.

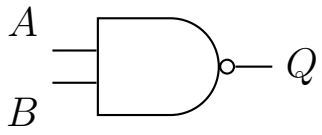
În mod firesc, semnalul de ieșire al acestei porți va fi complementarul operației *and*. La nivelul Tabelei 11 se poate observa acest lucru. Expresia matematică a acestei operații este $Q = \overline{A \cdot B}$.

Tabela 11: Funcția logică NAND

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

Simbolul utilizat în diagramele circuitelor este prezentat prin Figura 7.

Figura 9: Simbolul porții logice NAND



3.3.7 MULTIPLEXARE

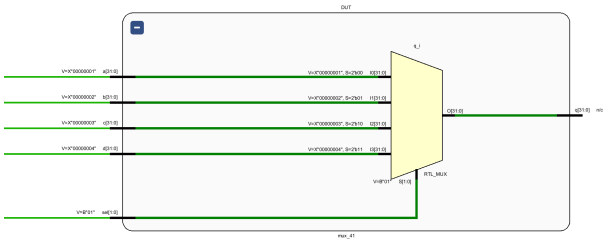
Pe lângă circuitele clasice reprezentate de porțile logice, un alt component digital, multiplexorul, merită studiat, acesta având o importanță deosebită. Prin multiplexare se întelege selecția unui anumite intrări dintr-o lista de n posibilități. Selecția este dictată de un alt semnal de intrare numit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

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.

Figura 10: Diagrama logică a unui multiplexor



3.4 ABSTRACTIA ARHITECTURALĂ

Arhitectura face legătura dintre microarhitectura 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 detalile 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 microarhitectural 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 fișiser 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, îndică 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.

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:

- Instrucțiuni R, *Register-Register*, pentru operațile 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 încarcare imediată a octeților cei mai semnificativi în registre.

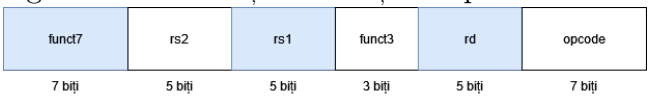
Tabela 12: Fișierul de registre RV32I

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

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. Figura 11 prezintă formatul acestei instrucțiuni cât și câmpurile de date constituente.

Figura 11: Instrucțiunea R și câmpurile de date

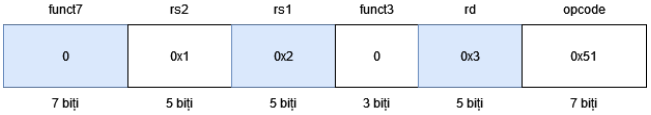


Campurile acestei instrucțiuni au următoarele semnificații:

- Câmpul *opcode* indică tipul instrucțiinii 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.

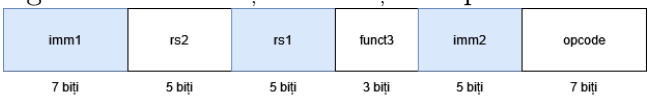
Figura 12: Exemplu de utilizare al instrucțiunii R



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. Figura 13 arată structura unei astfel de instrucțiuni.

Figura 13: Instrucțiunea R și câmpurile de date

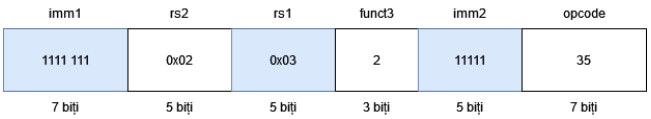


Semnificația câmpurilor este următoarea:

- Câmpul *opcode* indică tipul instrucțiinii 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 R



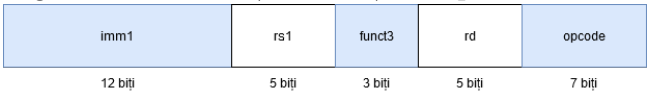
3.4.5 INSTRUCȚIUNEA I

Instrucțiunea I are rol în execuția operaților aritmetice cu valori imediate. Aceste valori sunt extinse la 32 de biti, ele ocupând doar 12 biți din lătimea instrucțiunii. În urma extensiei de semn, se efectueaza operația aritmetică specificată prin câmpul *funct3* cu registrul *rs1*, rezultatul fiind stocat în registrul *rd*. 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 *1*, rezultatul operației fiind trimis registrului *3*. Un astfel de exemplu se poate vedea în Figura 16

Figura 16: Instrucțiunea I și câmpurile de date



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. 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 aritmetic al procesorului și timpul de acces din memoria pricipală la datele necesare calculului. Diferența dintre aceste două metrici temporale poartă numele de *latență*. Într-adevăr, avansurile tehnologice aduse componentelor digitale de-a lungul decenilor, printre cele mai importante numărându-se îmbunătățirile tehnicii de litografie, 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 spațilă 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.

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.

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

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

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, miscorând drastic eficiența. Însă, din acest exemplu, se pot de asemenea observa principile 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.

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.

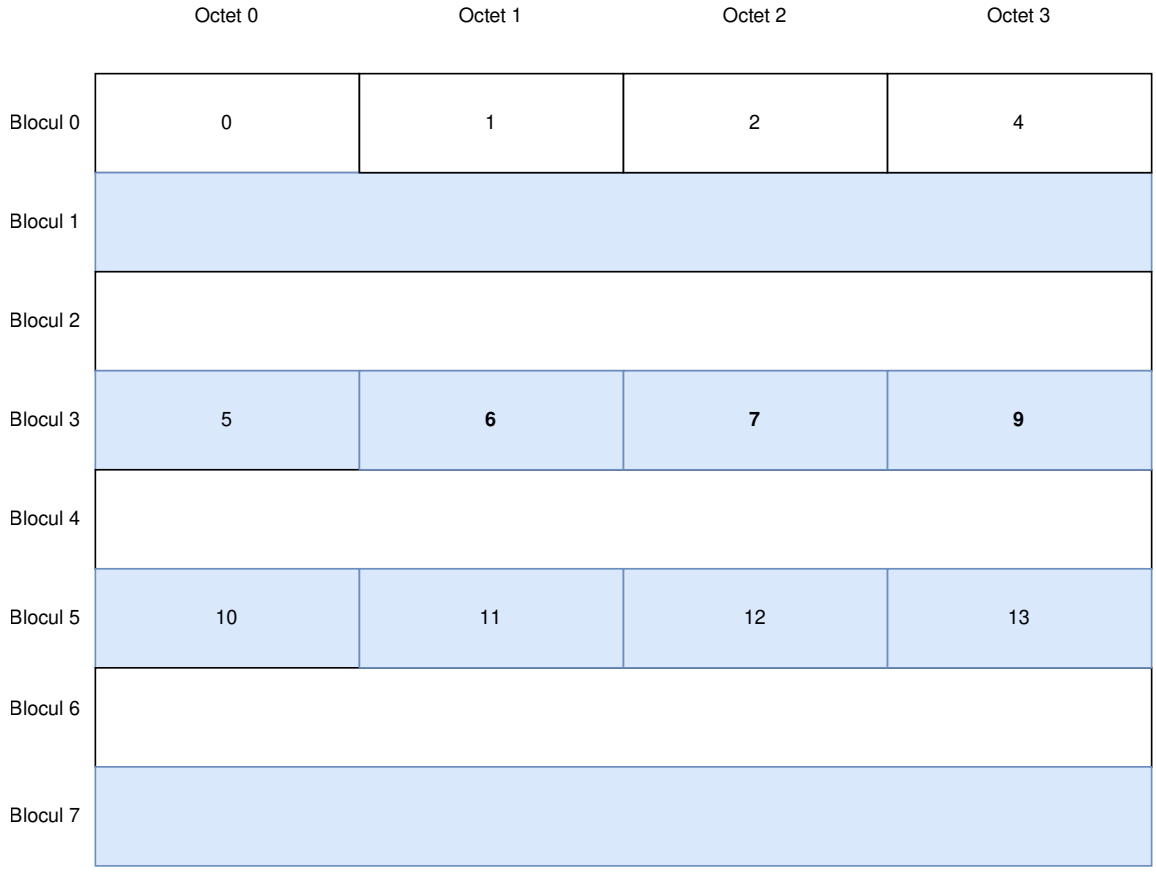
Metoda de asociere prezentată și implementată în această lucrare este cea asociativă directă, fiind de complexitatea cea mai redusă, prin comparație cu alternativele anterior menționate. Nu trebuie însă trecută cu vederea similaritatea cu celălalte tipuri de asocieri,

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.

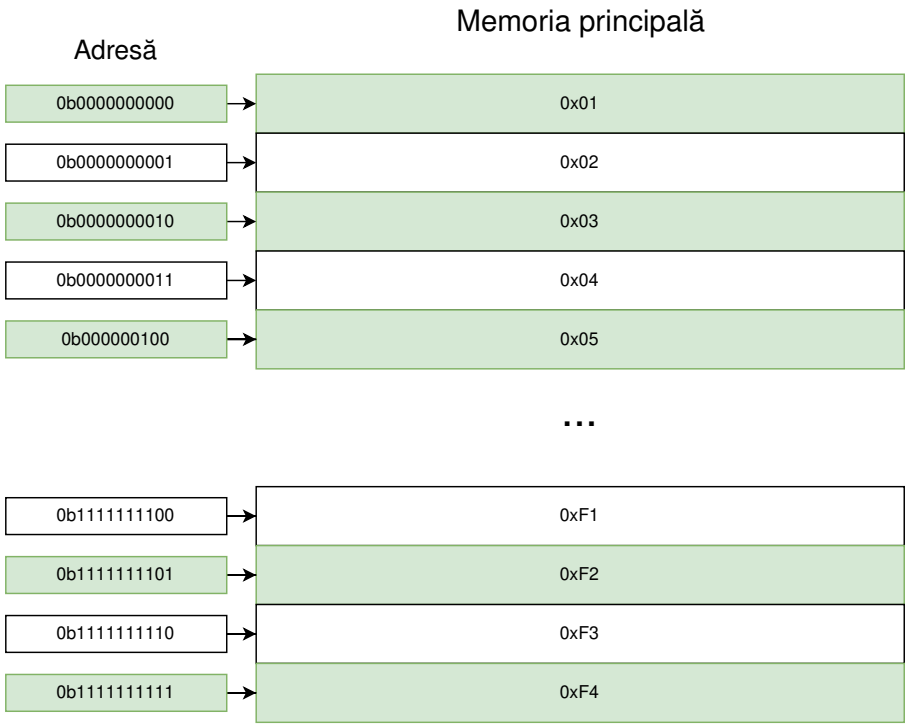
Este important de menționat faptul ca un bloc trebuie să acopere la cel mai rudimentar nivel cel putin 2 adrese ale memoriei principale. Astfel, mărimea blocurilor este direct relaționată la mărimea memoriei de date, un bloc de cache acoperind n adrese din aceasta.

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.

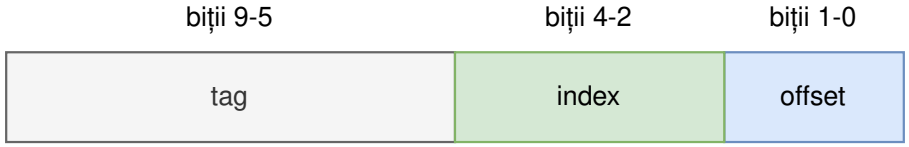
Figura 21: Organizarea memoriei principale



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 dupa 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.

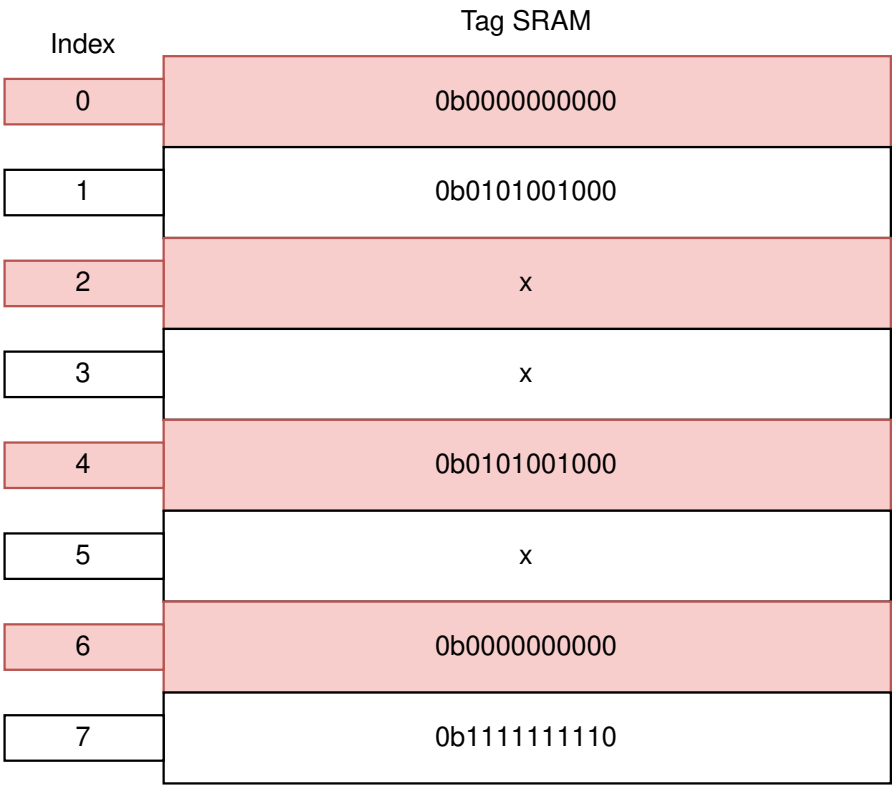
Figura 22: Organizarea memoriei principale



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. 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.

Se poate observa faptul că linile 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

Figura 23: Organizarea memoriei principale



indecsi, în memoria cache vor fi stocate blocuri care aparțin aceluiași tag, însă care cu un camp *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*. Pe lângă arbitrare, elementul de control 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.

Elementul de control 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 functie de tipul cererii procesorului, de citire sau de scriere.

La nivelul cache-ului predomină cererile de citire. 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 controler 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*.

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 odata cu înlocuirea blocului modificat. Semnalarea unui bloc modificat se va face printr-un bit suplimentar numic *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, dupa 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.

3.5 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, fiind prezentate formele de undă caracteristice.

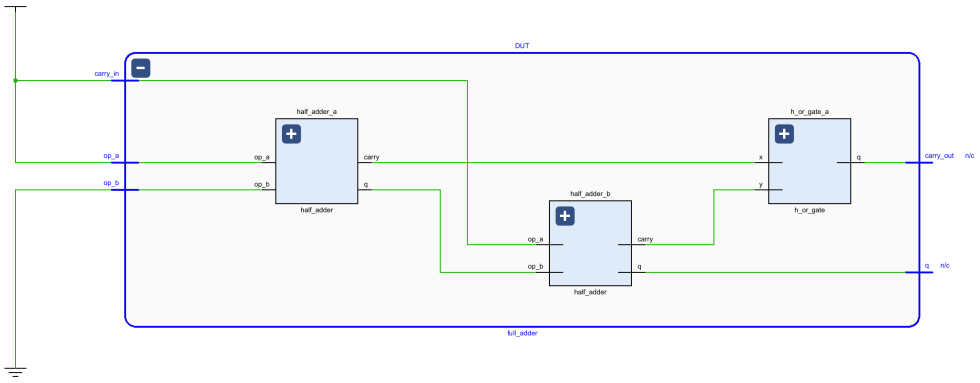
3.5.1 SUMATORUL ȘI ARITMETICA NUMERELOR

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.

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 integral și structura sa internă



Implementarea VHDL a entității cât și a arhitecturii sumatorului integral este redată prin secvența de cod aferentă Figurii 25. Se observă faptul că acestă entitate este construită

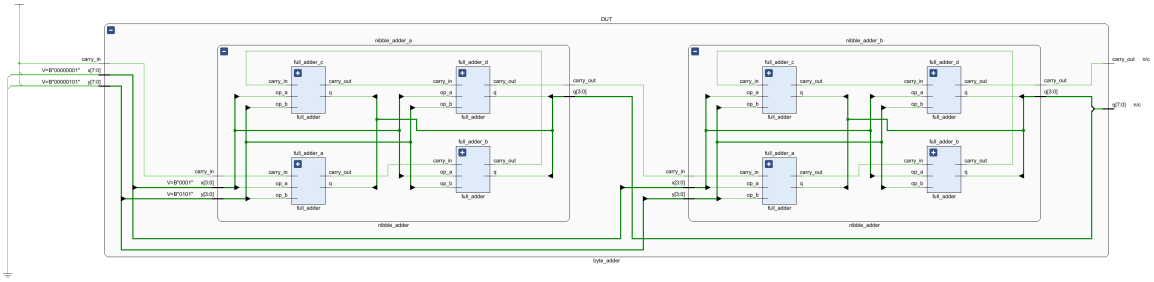
prin relaționarea a două sumatoare pe jumătăți și a unei porți logice *sau*.

Figura 25: Entitatea și arhitectura VHDL a sumatorului integral

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


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;
```

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 împărține în două grupuri de 4, fiecare numit *nibble adder*.

Figura 26: Sumator de octeți si elementele constituente



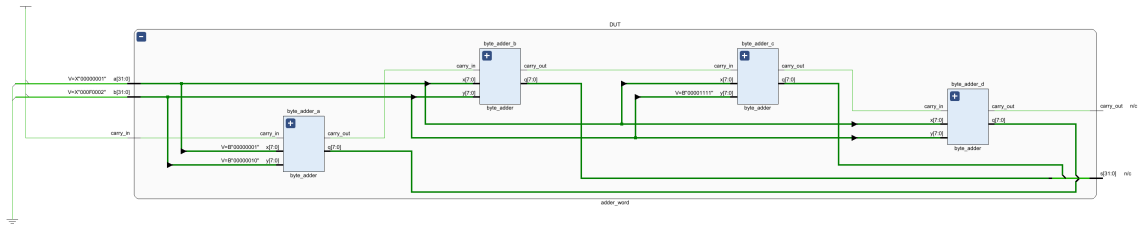
Implementarea VHDL a structurii digitale prezentată mai sus se poate vedea în Figura 30.

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.

Figura 27: Entitatea și arhitectura VHDL a sumatorului de octeți

```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 28: Sumator ripple carry adder de 32 de biți



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.

Tabela 13: Efectuarea operaților în funcție de selecția multiplexorului

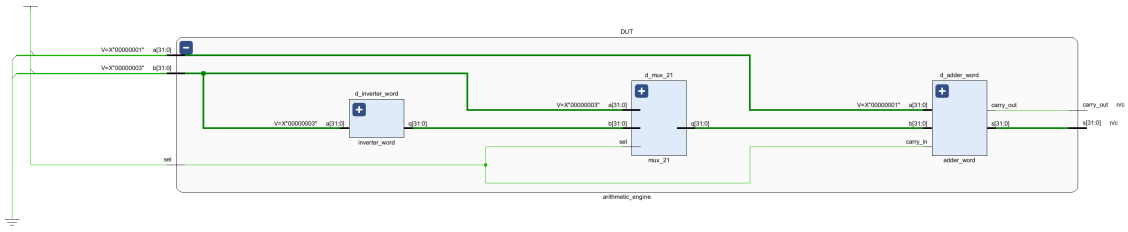
A	B	Carry	Sel	Operație
0x0406001F	0x031400A5	0	0	Adunare
0x0013121F	0x01144EB5	1	1	Scădere

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

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.

Figura 30: Entitatea și arhitectura VHDL a unității aritmetice

```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;
```

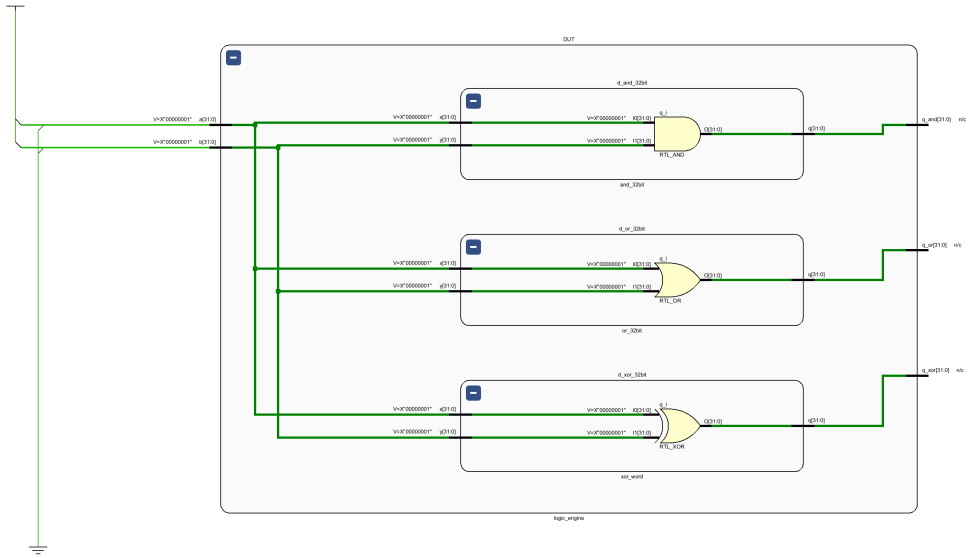
3.5.2 OPERAȚII LOGICE ȘI UNITATEA LOGIC

Pe lângă operațile 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.

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, operațile logice sunt o trivialitate, lucru redat nu doar prin diagrama digitală, dar și prin codul asociat unității logice, prezent în Figura 32.

Figura 32: Entitatea și arhitectura VHDL a unității logice

```
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;
```

3.5.3 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*.

Utilitatea detectării unui posibil *overflow* reiese în momentul în care dorim să executăm instrucțiunea *less than*. În cazul a două numere, a și b, se va execut 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.

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

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
$A \leq 0$	$B \leq 0$	$-2^{33} \leq A-B \leq 0$	Overflow posibil

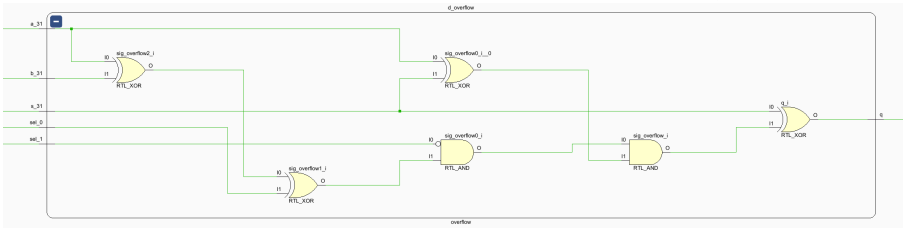
De asemenea, semnul diferenței este mereu diferit de semnul descăzutului, în cazul unui overflow. Astfel, acesta poate fi detectat prin verificarea bit-ului de semn al diferenței în cazul în care operanzii sunt de magnitudini opuse. Tabela 15 prezintă cazurile în care se poate detecta overflow-ul, raportat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

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

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

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

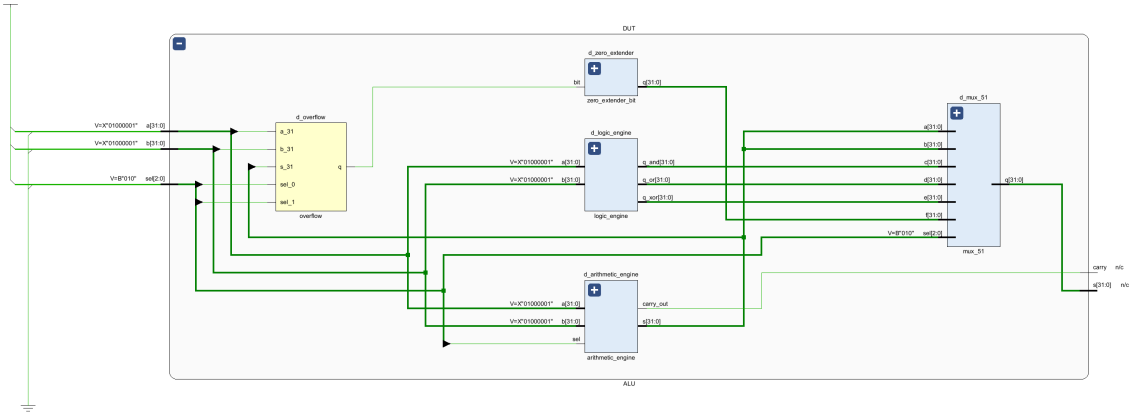


Figura 35: Unitatea aritmetică și logică

```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;
```

3.5.4 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 capacitoare 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.

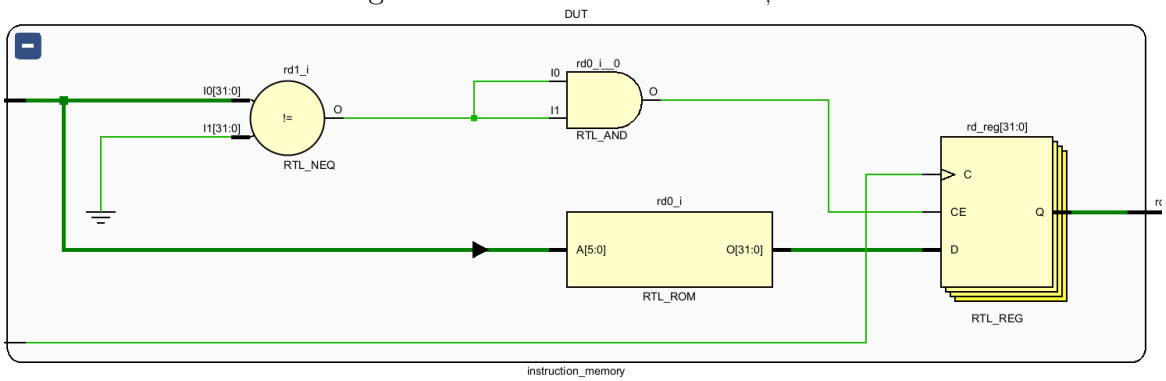
Memoria nevolatilă implementată în această lucrare se bazează pe celule de bistabile active pe front crescător. Datorită utilizării bistabilelor, latența de acces a datelor stocate în memorie este mică, principalul dezavantaj fiind implementarea practică, pricină costurilor ridicate ale unei astfel de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

Memoria din care se vor incărca instrucțiunile executate de procesor va purta numele de *Memoria de instrucțiuni* iar, memoria generală distinată stocării datelor va purta numele de *Memoria de date*.

Accesul la nivelul ambelor memorii se va face la nivel de octet, fiecare cuvânt regasindu-se astfel la o adresă multiplu de 4. Pentru a acomoda încărcarea instrucțiuniilor 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



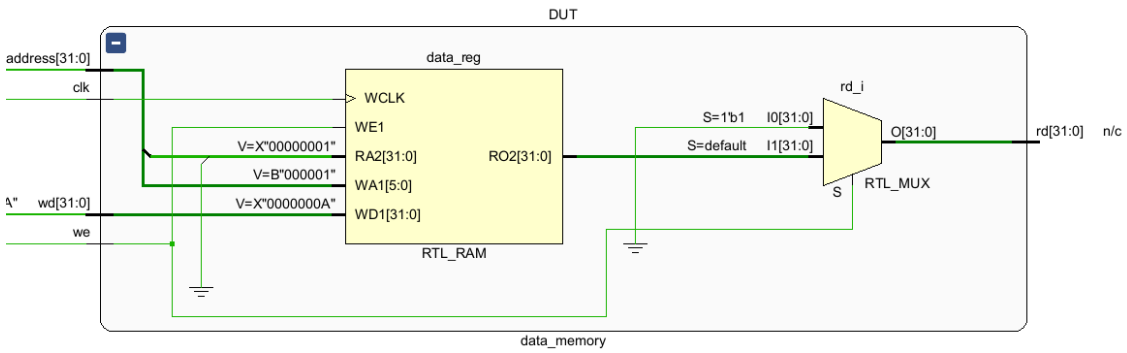
Codul entității VHDL al acestei memorii poate fi consultat în Figura 37. Este notabil modul în care ultimii 2 biți ai adresei instrucținii de executat sunt ignorați, cu ajutorul secvenței de cod *address(7 downto 2)*.

Figura 37: Memoria de Instrucțiuni ca entitate VHDL

```
1   library IEEE;
2   use ieee.std_logic_1164.all;
3   use ieee.numeric_std.all;
4
5   entity instruction_memory is generic(
6           word: integer := 31;
7           xlen: integer := 8
8       );
9       port(
10          clk: in std_logic;
11          address: in std_logic_vector (word downto 0);
12          rd: out std_logic_vector (word downto 0)
13      );
14  end instruction_memory;
15
16  architecture arch of instruction_memory is
17      type matrix is array(2**xlen - 1 downto 0) of std_logic_vector(word downto 0);
18      signal instruction: matrix := (
19          others => x"00000000");
20
21  begin
22      process(clk, address) begin
23       if rising_edge(clk) then
24           if address /= x"UUUUUUUU" and address /= x"XXXXXXXX" then
25              rd<= instruction(to_integer(unsigned(address(7 downto 2))));
26          end if;
27      end if;
28      end process;
29  end architecture;
```

În Figura 37 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 Instrucțiuni 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.

Figura 39: Memoria de Instrucțiuni ca entitate VHDL

```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;
```

3.5.5 FIȘIERUL DE REGISTRE

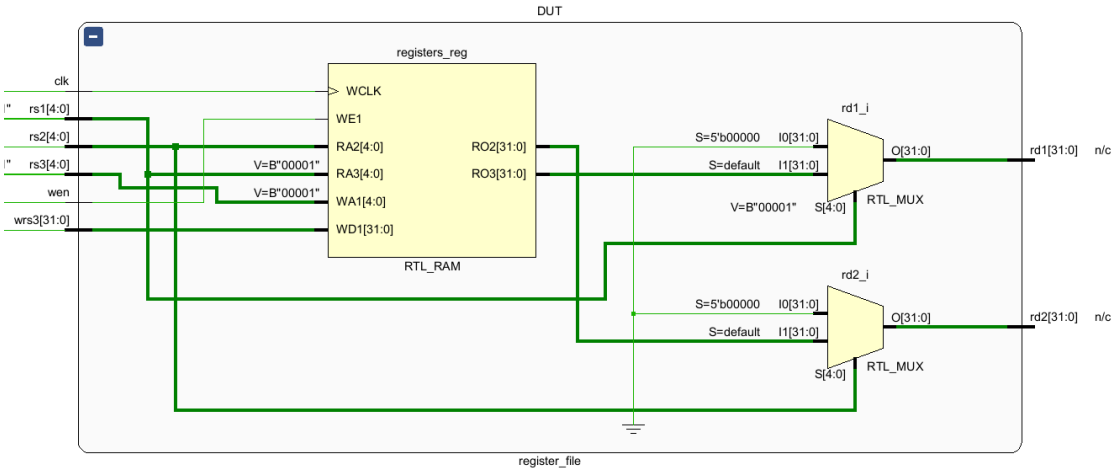
Precum s-a prezentata anterior, fișierul de registre al procesorului nostru acomodează 32 de registre cu funcții diverse, fiecare având o capacitate de stocare egală cu un cuvânt. Memorile implementate anterior sunt practic identice cu fișierul de registre, din punct de vedere al sintezeix automate,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

Cea mai importantă caracteristică a acestui element de design este faptul că se poate realiza citirea datelor în parelel cu scrierea lor, astfel fiind posibilă executarea instrucțiunilor consecutive de efectuare a operaților cu datele din registre. Dintre toate cele 32 de adrese posibile, este intersiză scrierea datelor în registrul 0, acesta fiind legat la masă, facilitând efectuarea operaților 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 40: Fișierul de registre



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

Figura 41: Fișierul de registre ca entitate VHDL

```
entity register_file is generic(···
end register_file;

architecture arch of register_file is
    type matrix is array(word downto 0) of std_logic_vector(word downto 0);

    signal registers: matrix := (···
begin
    process(clk) begin
        if rising_edge(clk) then
            if wen = '1' then
                registers(to_integer(unsigned(rs3))) <= wrs3;
            end if;
        end if;
    end process;

    process(rs1, rs2, clk) begin
        if to_integer(unsigned(rs1)) = 0 then
            rd1 <= x"00000000";
        else rd1 <= registers(to_integer(unsigned(rs1)));
        end if;

        if to_integer(unsigned(rs2)) = 0 then
            rd2 <= x"00000000";
        else rd2 <= registers(to_integer(unsigned(rs2)));
        end if;
    end process;

end architecture;
```

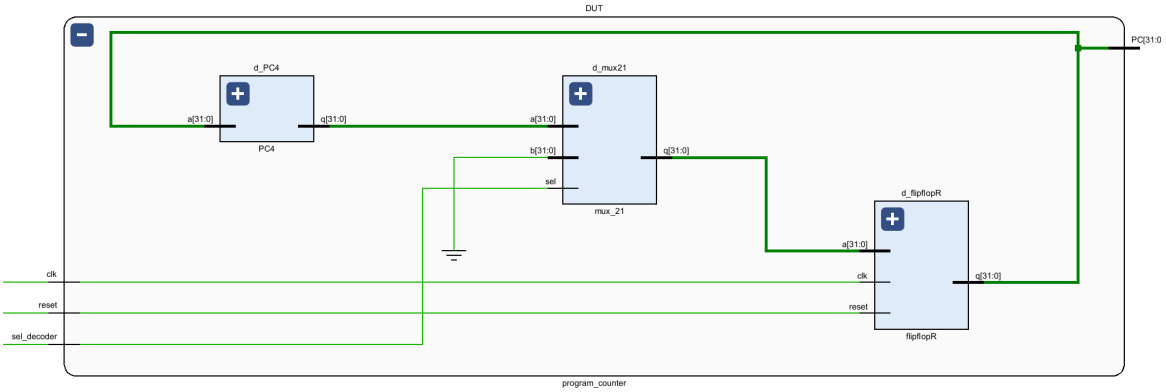
3.5.6 EXECUȚIA SINCRONIZATĂ ȘI PROGRAM COUNTER-UL

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. Acestă 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*.

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.

Figura 43: Bistabilul sincron ca entitate VHDL

```
entity flipflopR is
    generic(
        word: integer := 31
    );
    port(
        clk: in std_logic;
        reset: in std_logic;
        a: in std_logic_vector(word downto 0):= x"00000000";
        q: out std_logic_vector(word downto 0)
);
end flipflopR;


architecture arch of flipflopR is

begin
    process(clk, reset, a) begin
        if reset = '1' then
            q<= x"00000000";
        elsif rising_edge(clk) then
            q <= a;
        end if;
    end process;
end architecture;
```

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

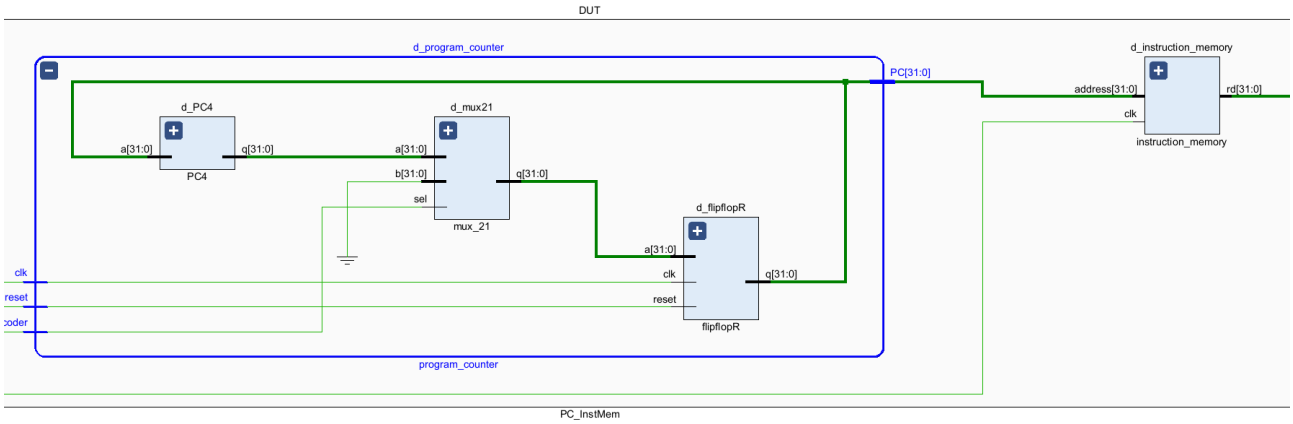
Figura 44: Bistabilul sincron ca entitate VHDL

```
entity program_counter is generic(
    word: integer := 31
    );
    port(
        sel_decoder: in std_logic;
        clk: in std_logic;
        reset:in std_logic;
        PC: buffer std_logic_vector(word downto 0)
    );
end program_counter;

architecture arch of program_counter is
    component flipflopR ···
    );
    end component;

    component PC4 generic(···
    );
    port(
        a: in std_logic_vector(word downto 0);
        q: out std_logic_vector(word downto 0)
    );
    end component;

    component mux_21 generic(···
    );
    end component;

    signal PCNext: std_logic_vector(word downto 0):= x"00000000";
    signal Plus4: std_logic_vector(word downto 0):= x"00000000";

begin
    d_flipflopR: flipflopR port map(clk=> clk, reset=> reset, a=> PCNext, q=> PC);
    d_PC4: PC4 port map(a=> PC, q=> Plus4);
    d_mux21: mux_21 port map(a=> Plus4, b=> x"00000000", sel=> sel_decoder, q=> PCNext);

end architecture;
```

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 secvente de executat.

Figura 45 prezintă o astfel de construcție logică, aceasta fiind practic entitatea de incărcare a instrucțiunilor în unitatea de control 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.

3.5.7 DECODORUL 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. Decodorul de instrucțiuni este elementul care va îndeplini un astfel de rol, trimițând biții de comanda mai departe elementelor digitale de execuție.

Decodorul 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 îm 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 acestă familie pe care decodorul 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 counter-ul 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, decodorul urmează să fie împărțit în două componente cu rol de decodificare. Primul component este practic decodorul principal, semnalând componentelor inferioare operațile care sunt pe cale de a fi executate. Decodorul secundar poartă numele de decodor aritmetic și are rolul de a orchestra modul de funcționare al unității aritmetice și logice. În mod practic, decodorul aritmetic va primi un cuvânt de comandă de la entitatea superioră, urmând ca acel cuvant să fie interpretat, comenzile aritmetice fiind trimise unității de calcul.

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

Tabela 16: Decorodul principal

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

Câmpurile trimie de către decodor spre componentele inferioare au următoarele semnificatii:

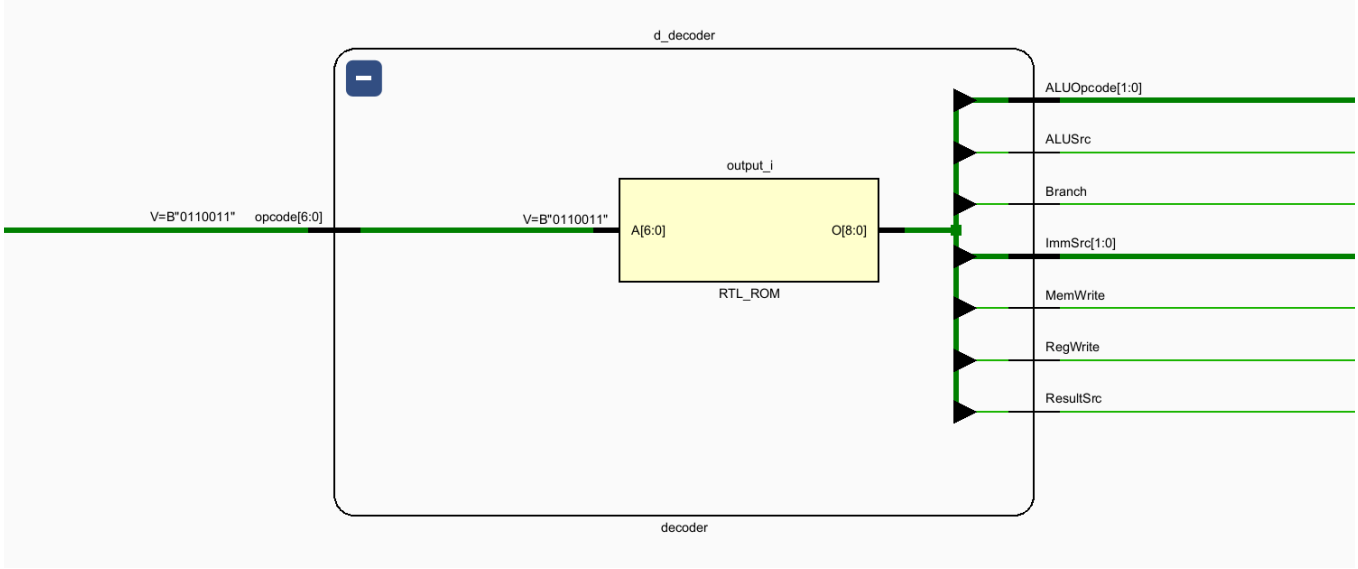
- *ALUOp* este cuvântul de comandă trimis decodorului aritmetic, acesta va indica în concordanță cu câmpul funct3 și funct7 operație pe care unitatea aritmetică și logică o va executa.
- *Reg Write* este bit-ul care indică fișerului de registre dacă se va realiza o scriere la nivelul său.
- *ImmSrc* 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 fisierul de registre.
- *Memwrite* înștiințează memoria principală ca urmează să se execute o scriere la nivelul său în momentul în care bit-ul 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. Valorea 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 decodor se poate consulta în Figura 46.

Figura 46: Diagrama digitală a decodorului aritmetic

```
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 47: Entitatea VHDL a decodorului principal



Decodorul aritmetic 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.

Tabela 17: Decorodul aritmetic

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

Pe lângă datele provenite de la unitatea superioară, decodorul 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.

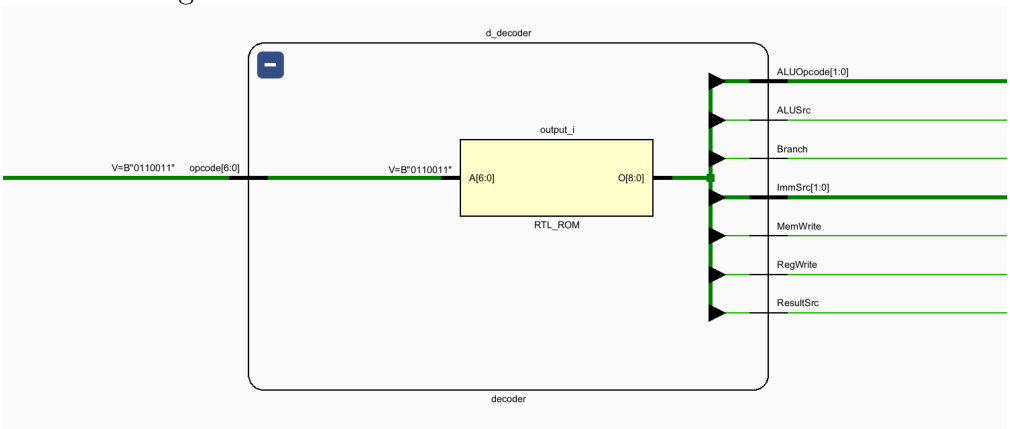
Codul VHDL al decodorului aritmetic este prezentat în Figura 49.

Figura 48: Entitatea VHDL a decodorului aritmetic

```
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;
```

Entitatea logică generată de sinteza VHDL pentru decodorul aritmetic se poate vedea în Figura 47.

Figura 49: Entitatea VHDL a decodorului aritmetic



4 BIBLIOGRAFIE

[1] Sarah Harris, David Harris (Octombrie, 2021), Digital Design and Computer Architecture, RISC-V Edition: RISC-V Edition, Morgan Kaufmann.

[2] Charles Petzold (Octombrie, 2000), Code: The Hidden Language of Computer Hardware and Software, Microsoft Press.

[3] David A. Patterson, John L. Hennessy (Aprilie, 2017), omputer Organization and Design RISC-V Edition: The Hardware Software Interface, Morgan Kaufmann.