

Microprocesorul RISC-V

Candidat: 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 ușor să ne pierdem în complexitățile 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 și 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, 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.

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 făcându-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 adății 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 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. 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 întregime 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 modelarea 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 ușor 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, detaliile de la un anumit nivel logic al unui sistem, sunt redată 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 și 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 părți 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 implementări 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 ABSTRAȚIA 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 în considerare faptul că 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 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 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 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 definește acest aspect este redată prin formula 1.

$$s = \log_r n \quad (1)$$

3.2.1 NUMERELE BINARE

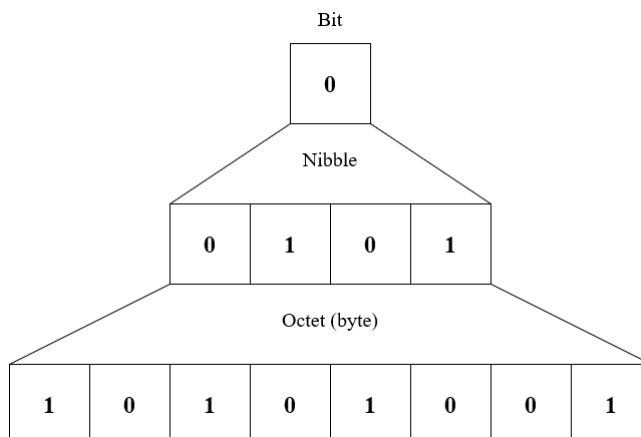
Modul de funcționare a dispozitivelor digitale este constituit pe oscilațiile rapide ale semnalelor electrice, semnale a căror valori se identifică cu unul dintre membri faimosului cuplu binar, 0 și 1. Prin urmare, datele vor avea o reprezentare care utilizează radixul binar.

Fiecare simbol dintr-un număr binar poartă 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 număr binar în octeți ajută reprezentarea acestora într-o bază numerică superioară, în special cea hexadecimală. Figura 1 prezintă clar părțile componente a unui octet și relația dintre acestea.

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



Octet-ul va reprezenta unitatea fundamentală și indivizibilă pentru microarhitectura microprocesorului 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 mărimea binară (numărul de biți folosiți în reprezentare) și cantitatea de date reprezentate prin plaja de valori adiacentă, ignorând existența numerelor negative.

Tabela 2: Plaja de valori asumând numere strict pozitive, de mărimi 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, înțelegerea bazei hexadecimale va fi cu atât mai simplă. Convertirea unui număr din binar în 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 folosite 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, 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

$$\begin{array}{r}
 1 \ 0 \ 0 \ 1 \ + \\
 0 \ 0 \ 1 \ 1 \\
 \hline
 1 \ 1 \ 0 \ 0
 \end{array}
 \qquad
 \begin{array}{r}
 1 \ 1 \ 1 \ 1 \ + \\
 1 \ 1 \ 1 \ 1 \\
 \hline
 \begin{array}{|c|} \hline \text{Carry} \\ \hline 1 \\ \hline \end{array}
 1 \ 1 \ 1 \ 0
 \end{array}
 \qquad
 \begin{array}{r}
 1 \ 0 \ 0 \ 1 \ + \\
 1 \ 0 \ 1 \ 1 \\
 \hline
 \begin{array}{|c|} \hline \text{Carry} \\ \hline 1 \\ \hline \end{array}
 0 \ 1 \ 0 \ 0
 \end{array}$$

$$\begin{array}{r}
 1 \ 0 \ 0 \ 0 \ 1 \ 1 \ 0 \ 0 \ + \\
 0 \ 0 \ 1 \ 1 \ 1 \ 0 \ 0 \ 1 \\
 \hline
 1 \ 1 \ 0 \ 0 \ 0 \ 1 \ 0 \ 1
 \end{array}
 \qquad
 \begin{array}{r}
 1 \ 0 \ 0 \ 0 \ 1 \ 1 \ 0 \ 0 \ + \\
 1 \ 0 \ 1 \ 1 \ 1 \ 0 \ 0 \ 1 \\
 \hline
 \begin{array}{|c|} \hline \text{Carry} \\ \hline 1 \\ \hline \end{array}
 0 \ 1 \ 0 \ 0 \ 0 \ 1 \ 0 \ 1
 \end{array}$$

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 număr 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 metodă 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.

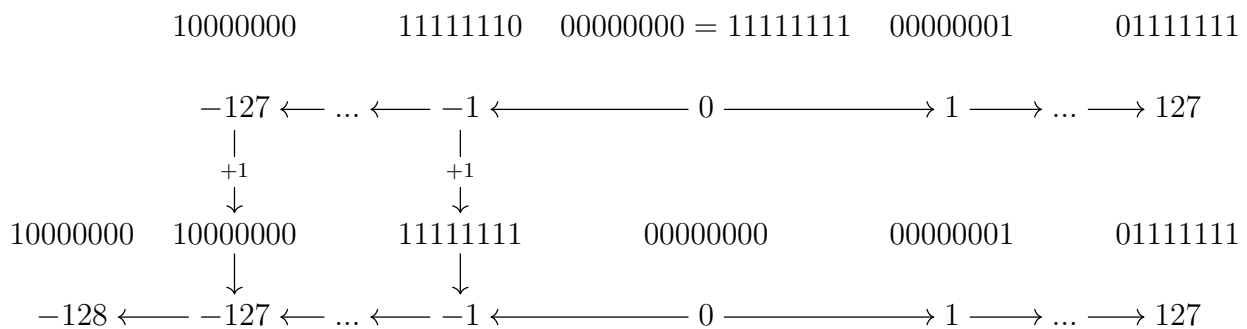
Tabela 6: Reprezentarea prin complement de 1

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 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. Efectul însumării unitare este cel mai bine explicat 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 architectural al sistemului și datele numerice, este necesară tratarea entităților digitale fundamentale cunoscute drept porți logice.

3.3 ABSTRAȚIA 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.

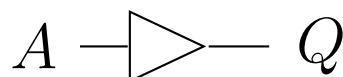
O poartă logică are rolul de a implementa o funcție matematică a algebrei Booleane. Comportamentul acestora este considerat ideal, întârzierile de propagare a impulsului astfel neglijabile. Pentru analiza funcției algebrice descrise se vor utiliza tabele logice, a căror rol este de a relaționa în format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

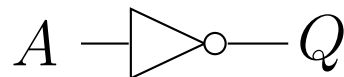
De asemenea cunoscut sub numele de *inversor*, rolul acestei porți logice este de a schimba polaritatea semnalului primit.

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 $Q = \bar{A}$.

Figura 5: Simbolul porții logice NOT



3.3.3 SAU

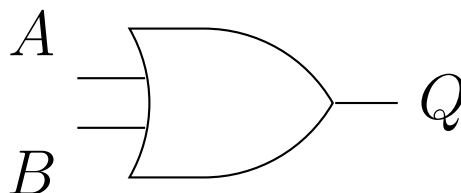
Poarta logică *sau* reprezintă implementarea operației de disjuncție matematică, a cărei rezultat este 0 doar atunci când ambele intrări logice sunt 0. Tabela 8 prezintă valorile operației raportate la intrările logice. Expresia matematică a operației *sau* este $Q = A + B$.

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

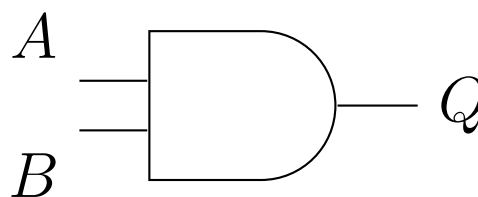
Poarta logică *și* reprezintă implementarea operației de conjuncție matematică, a cărei rezultat este 1 doar atunci când ambele intrări logice sunt 1. Tabela 9 prezintă valorile operației raportate la intrările logice. Expresia matematică a operației *și* este $Q = A \cdot B$.

Tabela 9: Funcția logică ȘI

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

Simbolul utilizat în diagramele circuitelor este prezentat prin Figura 7.

Figura 7: Simbolul porții logice ȘI



3.3.5 XOR

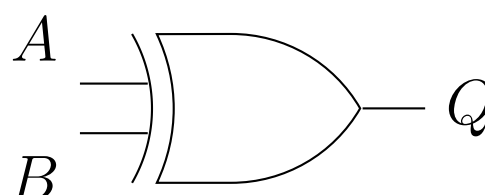
Poarta logică *xor*, de asemenea cunoscută ca *SAU exclusiv* setează semnalul de output pe 1 logic doar în cazul în care cel mult una dintre intrări este activă. Comportamentul acesteia este cel mai bine reprezentat de Tabela 10. Expresia matematică care descrie această poartă este $Q = A \oplus B$.

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 generală, 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.

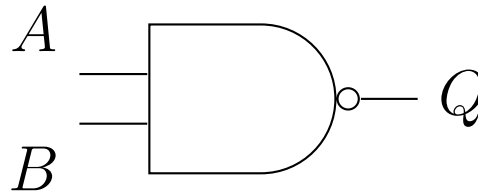
În mod firesc, semnalul de ieșire al acestei porți va fi complementarul operației *and*. La nivelul Tabelii 11 se poate observa acest lucru. Expresia matematică a acestei operații este $Q = \overline{A \cdot B}$.

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

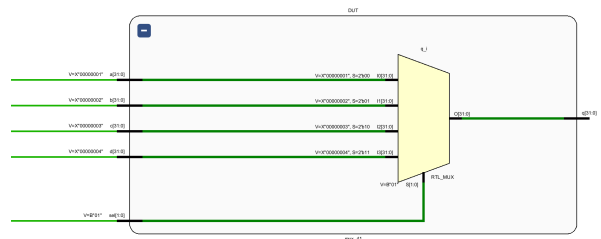
Pe lângă circuitele clasice reprezentate de porțile logice, un alt component digital, multiplexorul, merită studiat, acesta având o importanță deosebită. Prin multiplexare se înțelege selecția unui anumite intrări dintr-o lista de n posibilități. Selecția este dictată de un alt semnal de intrare numit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

Multiplexarea 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 $\log_2 n$ biți 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 detaliile arhitecturii, printre care se numără structura organizațională a registrelor, tipurile de instrucțiuni cât și 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 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.

3.4.2 INSTRUCȚIUNILE DE BAZĂ RISC-V

Operațiile 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 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.

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 operațiile aritmetice, precum adunarea și scăderea numerelor. Figura 11 prezintă formatul acestei instrucțiuni cât și câmpurile de date constitutive.

Figura 11: Instrucțiunea R și câmpurile de date

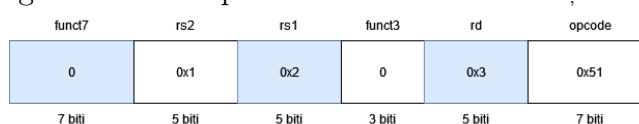


Câmpurile acestei instrucțiuni au următoarele semnificații:

- 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.

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

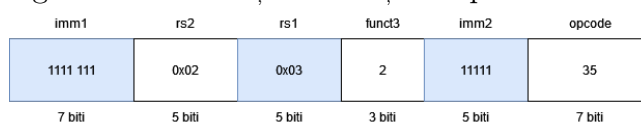


Semnificația câmpurilor este următoarea:

- Câmpul *opcode* indică tipul instrucțiunii de executat.
- Câmpul *rs1* conține adresa registrului în care se găsesc datele destinate stocării.
- Câmpul *rs2* conține adresa registrului față de a căruia valoare se va calcula offset-ul.
- Câmpul *funct3* conține 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 adăuga un offset de -1.

Figura 14: Instrucțiunea R și câmpurile de date



3.4.5 INSTRUCȚIUNEA I

Instrucțiunea I are rol în execuția operațiilor aritmetice cu valori imediate. Aceste valori sunt extinse la 32 de biți, 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*. 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 folosită 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

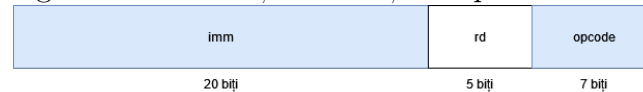
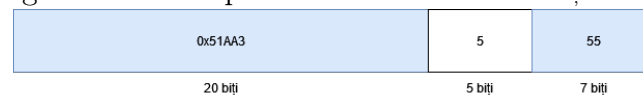


Figura 18 prezintă un exemplu de utilizare, în registrul 5 fiind încărcată valoarea imediată 0X51AA3.

Figura 18: Exemplu de utilizare al instrucțiunii U



4 IMPLEMENTARE

4.1 ABSTRAȚIA 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.

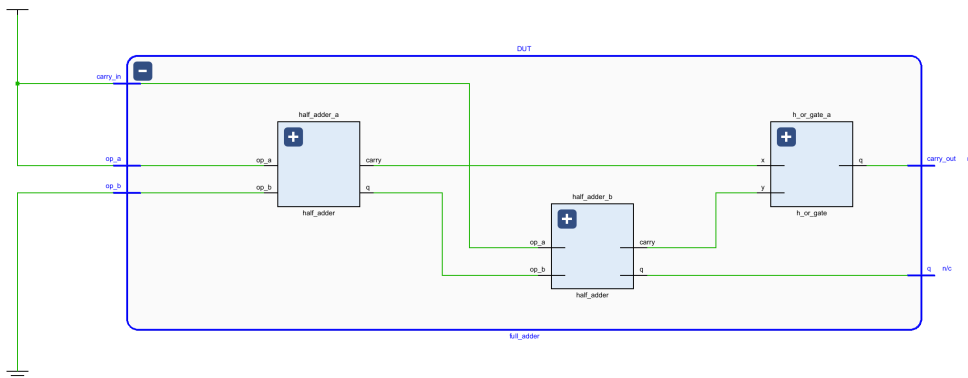
4.1.1 SUMATORUL ȘI ARITMETICA NUMERELOR

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 19. Acesta are rolul de a executa suma a 2 biți, precum s-a prezentat la nivelul abstracției numerice.

Figura 19: Sumator integral și structura sa internă



Implementarea VHDL a entității cât și a arhitecturii sumatorului integral este redată prin secvența de cod aferentă Figurii 20. Se observă faptul că această entitate este construită prin relaționarea a două sumatoare pe jumătăți și a unei porți logice *sau*.

Figura 20: Entitatea și arhitectura VHDL a sumatorului integral

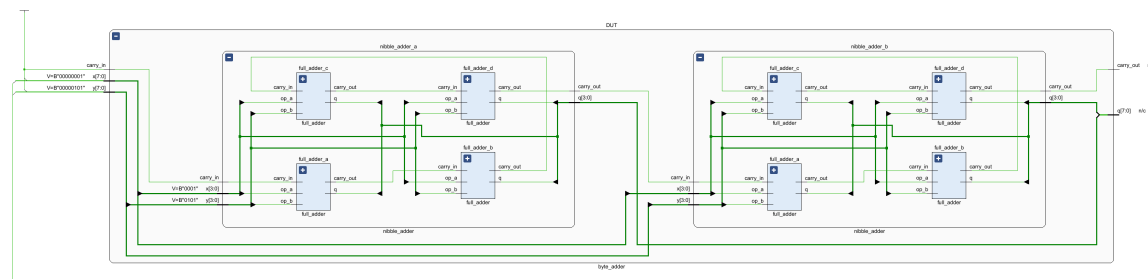
```

1  library IEEE;
2  use ieee.std_logic_1164.all;
3
4
5  entity h_or_gate is port(
6    x: in std_logic;
7    y: in std_logic;
8    q: out std_logic
9  );
10 end h_or_gate;
11
12 architecture arch of h_or_gate is
13
14 begin
15   q <= x or y;
16 end arch;
17
18 library arithmetic;
19 library IEEE;
20 use ieee.std_logic_1164.all;
21 use ieee.numeric_std.all;
22
23 entity full_adder is port(
24   op_a: in std_logic;
25   op_b: in std_logic;
26   carry_in: in std_logic;
27   q: out std_logic;
28   carry_out: out std_logic
29 );
30 end full_adder;
31
32 architecture arch of full_adder is
33   component half_adder port(
34     op_a: in std_logic;
35     op_b: in std_logic;
36     q: out std_logic;
37     carry: out std_logic
38   );
39   end component;
40
41   component h_or_gate port(
42     x: in std_logic;
43     y: in std_logic;
44     q: out std_logic
45   );
46   end component;
47
48   signal q_half_adder_a: std_logic := '0';
49   signal carry_half_adder_a: std_logic := '0';
50   signal carry_half_adder_b: std_logic := '0';
51
52 begin
53   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);
54   half_adder_b: half_adder port map(op_a => carry_in, op_b => q_half_adder_a, q => q, carry => carry_half_adder_b);
55   h_or_gate_a: h_or_gate port map(x => carry_half_adder_a, y => carry_half_adder_b, q => carry_out);
56 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 21. Se poate observa faptul că cele 8 sumatoare sunt împărțite în două grupuri de 4, fiecare numit *nibble adder*.

Figura 21: Sumator de octeți și elementele constitutive



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

Figura 22: Entitatea și arhitectura VHDL a sumatorului de octeți

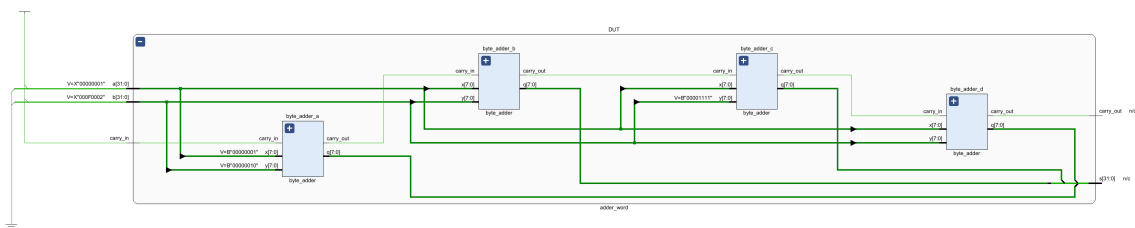
```

1  library IEEE;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4
5  entity byte_adder is port(
6      x: in std_logic_vector (7 downto 0);
7      y: in std_logic_vector (7 downto 0);
8      carry_in: in std_logic;
9      q: out std_logic_vector (7 downto 0);
10     carry_out: out std_logic
11 );
12 end byte_adder;
13
14 architecture arch of byte_adder is
15     component nibble_adder port(
16         x: in std_logic_vector (3 downto 0);
17         y: in std_logic_vector (3 downto 0);
18         q: out std_logic_vector (3 downto 0);
19         carry_in: in std_logic;
20         carry_out: out std_logic
21     );
22     end component;
23
24     signal sig_carry_out: std_logic := '0';
25 begin
26     nibble_adder_a: nibble_adder port map(x => x(3 downto 0), y=> y(3 downto 0),
27         carry_in => carry_in, q => q(3 downto 0), carry_out => sig_carry_out);
28     nibble_adder_b: nibble_adder port map(x => x(7 downto 4), y=> y(7 downto 4),
29         carry_in => sig_carry_out, q => q(7 downto 4), carry_out => carry_out);
30 end architecture;

```

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 23.

Figura 23: 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 operațiilor în funcție de selecția multiplexorului

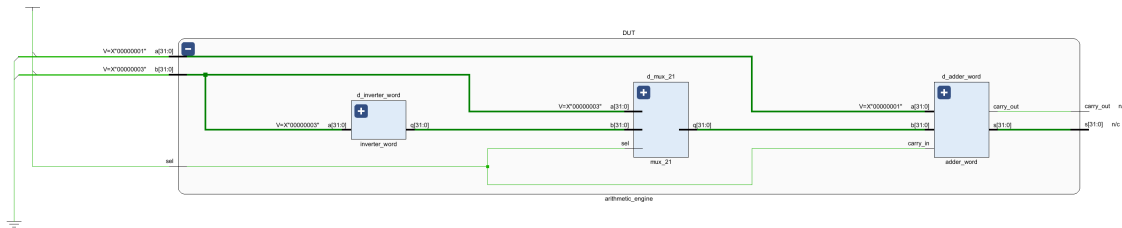
A	B	Carry	Sel	Operație
0x0406001F	0x031400A5	0	0	Adunare
0x0013121F	0x01144EB5	1	1	Scădere

Se poate observa faptul că semnalul de *carry* este setat pe 1 în cazul scăderii. Acest

lucru formează practic complementul de 2 necesar scăderii, numărul 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 24.

Figura 24: Unitatea aritmetică



Entitatea VHDL a unității aritmetice este redată prin codul prezentat în Figura 25.

Figura 25: Entitatea și arhitectura VHDL a unității aritmetice

```

1  library IEEE;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4
5  entity arithmetic_engine is generic(
6      word: integer := 31
7  );
8
9
10 port(
11     a: in std_logic_vector (word downto 0);
12     b: in std_logic_vector (word downto 0);
13     sel: in std_logic;
14     carry_out: out std_logic;
15     s: out std_logic_vector (word downto 0)
16 );
17 end arithmetic_engine;
18
19 architecture arch of arithmetic_engine is
20 >   component adder_word generic(...)
21 >   );
22 >   end component;
23
24 >   component inverter_word generic(...)
25 >   );
26 >   port(...)
27 >   );
28 >   end component;
29
30 >   component mux_21 generic(...)
31 >   );
32 >   port(
33 >     a: in std_logic_vector (word downto 0);
34 >     b: in std_logic_vector (word downto 0);
35 >     sel: in std_logic;
36 >     q: out std_logic_vector (word downto 0)
37 >   );
38 >   end component;
39
40   signal inverter_output: std_logic_vector (word downto 0) := (others => '0');
41   signal multiplexed_output: std_logic_vector (word downto 0) := (others => '0');
42
43 begin
44   d_inverter_word: inverter_word port map(a => b, q => inverter_output);
45   d_mux_21: mux_21 port map(a => b, b => inverter_output, sel => sel, q => multiplexed_output);
46   d_adder_word: adder_word port map(a => a, b => multiplexed_output, carry_in => sel, s => s, carry_out => carry_out);
47 end architecture;

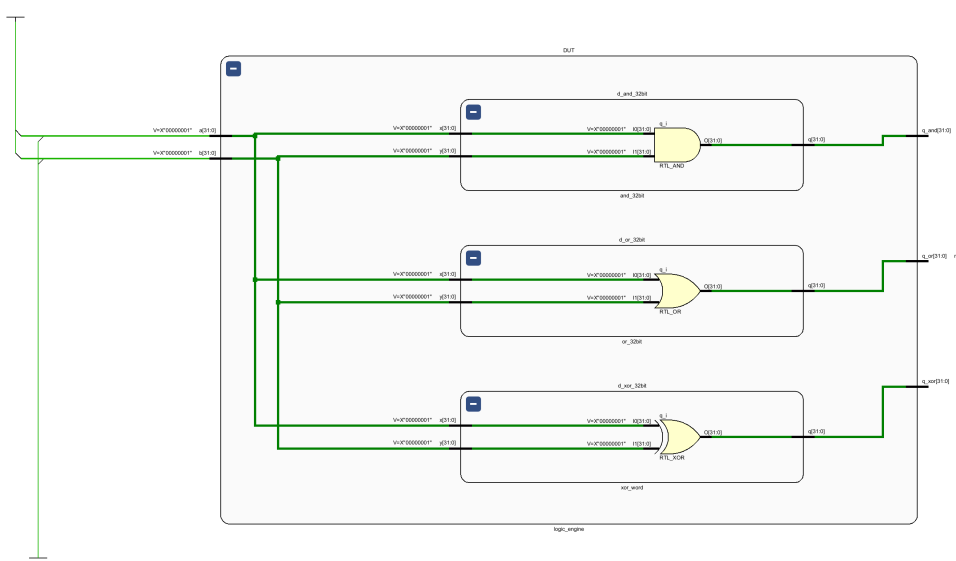
```

4.1.2 OPERAȚII LOGICE ȘI UNITATEA LOGIC

Pe lângă operațiile aritmetice, un procesor de asemenea are posibilitatea de executare a operațiilor 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 26 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 26: 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 27.

Figura 27: 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...
19 begin
20     d_and_32bit: and_32bit port map(x=> a, y=> b, q=> q_and);
21     d_or_32bit: or_32bit port map(x=> a, y=> b, q=> q_or);
22     d_xor_32bit: xor_word port map(x=> a, y=> b, q=> q_xor);
23 end architecture;
```

4.1.3 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 executa operația de scădere, semnul rezultatului acestei operații determinând care dintre cele 2 numere este mai mare. O magnitudine negativă indică faptul că 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 scăderii 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
$A \leq 0$	$B \geq 0$	$-2^{33} \leq A - B \leq 0$	Overflow posibil

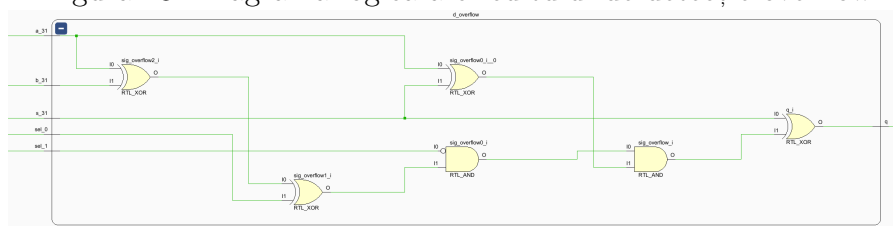
De asemenea, semnul diferenței este mereu diferit de semnul descăzutului, în cazul unui overflow. Astfel, acesta poate fi detectat prin verificarea bit-ului de semn al diferenței în cazul în care operandii sunt de magnitudini opuse. Tabela 15 prezintă cazurile în care se poate detecta overflow-ul, raportat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

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

Figura 28: Diagrama logică a circuitului de detecție overflow



Prin urmare, îmbinând toate elementele modelate anterior, rezultă schema digitală a unității aritmetice și logice din Figura 29. În Figura 30 se poate observa codul entității și al arhitecturii VHDL aferent acestei componente.

Figura 29: Unitatea aritmetică și logică

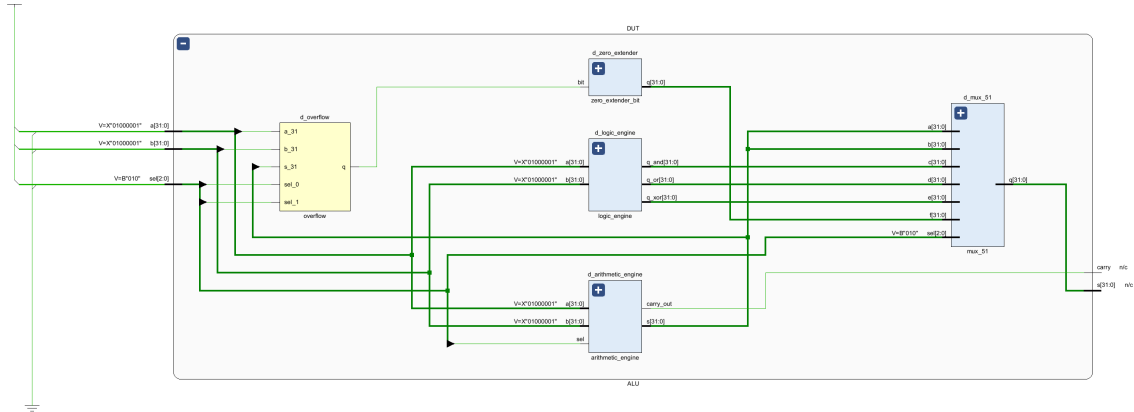


Figura 30: Unitatea aritmetică și logică

```

1  library IEEE;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4
5  entity ALU is generic (
6      word: integer := 31;
7      opt: integer := 2
8  );
9
10 port(
11     a: in std_logic_vector (word downto 0);
12     b: in std_logic_vector (word downto 0);
13     sel: in std_logic_vector (opt downto 0);
14     s: out std_logic_vector (word downto 0);
15     carry: out std_logic
16 );
17
18 end ALU;
19
20 architecture arch of ALU is
21 > component arithmetic_engine is generic(...)
22 end component;
23
24 > component logic_engine generic(...)
25 port(...)
26 );
27 end component;
28
29 > component overflow port(...)
30 );
31 end component;
32
33 > component zero_extender_bit generic(...)
34 );
35 end component;
36
37 > component mux_51 generic(...)
38 end component;
39
40 signal sig_arithmetic_engine_output: std_logic_vector (word downto 0) := (others => '0');
41
42 signal sig_logic_engine_and: std_logic_vector (word downto 0) := (others => '0');
43 signal sig_logic_engine_or: std_logic_vector (word downto 0) := (others => '0');
44 signal sig_logic_engine_xor: std_logic_vector (word downto 0) := (others => '0');
45
46 signal sig_overflow: std_logic := '0';
47
48 signal sig_zero_extender: std_logic_vector (word downto 0) := (others => '0');
49
50 begin
51     d_arithmetic_engine: arithmetic_engine port map(a=> a, b=> b, sel=> sel(0), s=> sig_arithmetic_engine_output, carry_out=> carry);
52     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);
53     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);
54     d_zero_extender: zero_extender_bit port map(bit=> sig_overflow, q=> sig_zero_extender);
55     d_mux_51: mux_51 port map(a=> sig_arithmetic_engine_output, b=> sig_arithmetic_engine_output,
56                               c=> sig_logic_engine_and, d=> sig_logic_engine_or, e=> sig_logic_engine_xor, f=> sig_zero_extender, sel=> sel, q=> s);
57 end architecture;

```

5 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), Computer Organization and Design RISC-V Edition: The Hardware Software Interface, Morgan Kaufmann.