

**Florin ONIGA**

# **DE LA BIT LA PROCESOR.**

## **Introducere în arhitectura calculatoarelor**



**Editura UTPRESS**  
**Cluj-Napoca, 2019**  
**ISBN 978-606-737-366-0**



Editura U.T.PRESS  
Str.Observatorului nr. 34  
400775 Cluj-Napoca  
Tel.:0264-401.999  
e-mail: [utpress@biblio.utcluj.ro](mailto:utpress@biblio.utcluj.ro)  
<http://biblioteca.utcluj.ro/editura>

Director: Ing. Călin D. Câmpean

Recenzia: Prof.dr.ing. Radu DĂNESCU  
Conf.dr.ing. Tiberiu MARIȚA

Copyright © 2019 Editura U.T.PRESS

Reproducerea integrală sau parțială a textului sau ilustrațiilor din această carte este posibilă numai cu acordul prealabil scris al editurii U.T.PRESS.

**ISBN 978-606-737-366-0**

Bun de tipar: 06.05.2019

## Cuprins

<b>1. Introducere .....</b>	<b>3</b>
<b>2. Reprezentarea numerelor. Sisteme de numerație .....</b>	<b>5</b>
2.1. Reprezentarea numerelor în baza 10. Sistemul zecimal .....	5
2.2. Reprezentarea numerelor în baza 2. Sistemul binar .....	7
2.2.1. Aritmetica numerelor în baza 2. Reprezentarea în complement față de 2 .....	9
2.3. Reprezentarea numerelor în baza 16. Sistemul hexazecimal.....	12
<b>3. Algebră booleană și porți logice .....</b>	<b>15</b>
3.1. Algebra booleană – noțiuni de bază.....	15
3.2. Teoreme/legi specifice .....	17
3.3. Reprezentarea cu termeni canonici .....	20
<b>4. Circuite combinaționale și secvențiale – fundamente .....</b>	<b>23</b>
<b>5. Circuite combinaționale.....</b>	<b>25</b>
5.1. Multiplexorul .....	25
5.2. Demultiplexorul .....	27
5.3. Decodificatorul.....	28
5.4. Sumatorul.....	29
5.5. Unitatea Aritmetică-Logică .....	34
<b>6. Circuite secvențiale .....</b>	<b>37</b>
6.1. Semnalul de ceas .....	37
6.2. Circuite secvențiale sincrone .....	38
6.2.1. Bistabilul D sincron, activ pe front crescător de ceas.....	38
6.2.2. Registrul pe n biți .....	40
6.3. Model ideal versus implementarea fizică.....	41
6.4. Un exemplu de circuit combinat.....	46
<b>7. Circuite de memorie.....</b>	<b>47</b>
7.1. Concepte de bază: adrese, date, organizare .....	47
7.2. Memoria RAM.....	49
7.3. Memoria ROM.....	50
7.4. Blocul de registre .....	51

<b>8. Aplicativ: construirea unor circuite mai complexe.....</b>	<b>53</b>
8.1. Sinteză de nivel înalt .....	53
8.1.1. Varianta 1: Fără constrângeri de resurse.....	53
8.1.2. Varianta 2: Cu constrângeri de resurse .....	54
8.2. O abordare mai generală, spre conceptul de procesor .....	58
8.2.1. Setul de instrucțiuni .....	58
8.2.2. Identificarea componentelor necesare + asamblare parțială .....	60
8.2.3. Asamblarea completă a căii de date.....	63
8.2.4. Unitatea de control.....	64
<b>9. Proiectarea procesorului MIPS, varianta ciclu unic.....</b>	<b>67</b>
9.1. Arhitectura setului de instrucțiuni – aspecte generale .....	67
9.2. Arhitectura setului de instrucțiuni – studiu de caz MIPS 32 .....	68
9.2.1. Registre .....	68
9.2.2. Organizarea memoriei.....	69
9.2.3. Tipuri de date / operanzi .....	69
9.2.4. Tipuri de instrucțiuni .....	69
9.2.5. Instrucțiuni reprezentative.....	71
9.2.6. Etape de execuție .....	74
9.3. Metodologia de proiectare bazată pe RTL .....	74
9.3.1. Analiza setului de instrucțiuni.....	75
9.3.2. Setul de componente .....	78
9.3.3. Asamblarea căii de date .....	80
9.3.4. Semnalele de control – definire completă.....	90
9.3.5. Unitatea de control.....	97
9.4. Funcționarea în implementarea fizică. Frecvența de ceas .....	100
<b>10. Programe, compilare, execuție .....</b>	<b>105</b>
10.1. Un program simplu. Din C în asamblare .....	105
10.2. Un program simplu. Din asamblare în cod mașină .....	107
10.3. Execuția programului .....	109
<b>11. Extinderea procesorului .....</b>	<b>111</b>
<b>Bibliografie.....</b>	<b>117</b>

## 1. Introducere

În această carte se prezintă concepte de bază din arhitectura calculatoarelor, cu scopul principal de a se înțelege cum este proiectat și cum funcționează un procesor. Pornind de la reprezentarea numerelor binare, se introduc, gradual, conceptele necesare pentru proiectarea procesorului: algebră booleană, proiectare logică, circuite combinaționale și secvențiale. După proiectarea pas cu pas a procesorului, se explică cum se transformă programele, din limbaje de nivel înalt în cod mașină, pentru a fi executate de către procesor.

Ca studiu de caz, se construiește un procesor bazat pe arhitectura RISC (Reduced Instruction Set Computer). Procesorul prezentat este MIPS (Micro-processor without Interlocked Pipeline Stages), în varianta *ciclu unic*, în care fiecare instrucțiune se execută pe o perioadă de ceas. Acesta este utilizat aproape universal ca material de studiu în domeniul academic, fiind propus și popularizat de John Hennessy și David Patterson, în seria de cărți *Computer organization and design: the hardware/software interface* (mai multe ediții, menționate punctual de-a lungul acestei cărți).

Cartea se adresează cititorilor dornici să înțeleagă cum se construiesc circuitele digitale și, mai ales, cum se construiește și cum funcționează “creierul” unui calculator. Nu sunt necesare cunoștințe anterioare despre circuite digitale. Obiectivul autorului este oarecum ambițios, dat fiind numărul nu foarte mare de pagini ale cărții. Pentru a atinge acest obiectiv, autorul a prezentat conceptele de bază din primele capitole insistând pe exemple și, selectiv, doar pe acele detalii care sunt relevante mai târziu, când este construit procesorul. Pentru unele subiecte se oferă explicații intuitive și se omit intenționat anumite caracteristici care pot ridica cititorului întrebări suplimentare și, implicit, pot crea confuzie. Pentru cititorii interesați de detalii suplimentare se oferă referințe spre capitolele specifice din cărțile fundamentale ale domeniului.

Complexitatea subiectelor crește cu fiecare capitol. Se recomandă cititorului o lectură atentă și activă, în sensul parcurgerii în scris (foaie/instrument de scris) a exemplurilor date în carte și desenarea diferitelor diagrame pentru circuitele construite.

Multe dintre circuitele prezentate sunt construite gradual, pe baza noțiunilor prezentate anterior în carte. Astfel, cititorul poate avea o vedere de ansamblu, cu o înțelegere mai clară a procesului prin care, pornind de la sistemul de numerație binar, se ajunge la circuite digitale complexe (inclusiv la un procesor).

În capitolul 2 se prezintă sisteme de reprezentare a numerelor (întregi), începând cu familiarul sistem zecimal, apoi sistemul binar (cu aritmetica asociată) specific circuitelor digitale, și terminând cu sistemul hexazecimal.

În capitolul 3 se prezintă noțiuni de algebră booleană care stau la baza proiectării logice a circuitelor digitale, pornind de la cele mai simple circuite: porțile logice.

În capitolul 4 se discută caracteristicile principale ale celor două tipuri de circuite: combinaționale, respectiv secvențiale.

În capitolul 5 sunt prezentate circuitele combinaționale cele mai utilizate, cu accent pe multiplexor, decodificator și sumator. Se explică cum acestea pot fi construite din porți logice, folosind noțiunile din capitolul 3. La finalul acestui capitol se proiectează o unitate aritmetică-logică, unitate care va fi folosită mai târziu ca parte a procesorului MIPS.

În capitolul 6 se introduce o noțiune de bază în calculatoare, și anume semnalul de ceas. Se prezintă al doilea tip de circuite, cele secvențiale, cu accent pe elementul standard de memorare a unui bit: bistabilul D sincron. Se discută atât funcționarea din punct de vedere logic, cât și funcționarea în implementarea fizică. Această paralelă este importantă,

deoarece, odată lămurită, ușurează mult înțelegerea funcționării unor circuite mai complexe.

În capitolul 7 se prezintă conceptele de bază pentru memorii, apoi principalele circuite de memorie care vor fi utilizate pentru construirea procesorului MIPS.

În capitolele 2-7 sunt prezentate toate fundamentele (metode și circuite de bază) necesare pentru a începe proiectarea procesorului. Capitolul 8 reprezintă un pas intermediar spre procesul de proiectare. Se permite cititorului familiarizarea cu conceptele anterioare, prin aplicarea lor pentru a construi circuite care rezolvă probleme specifice. În partea a doua a capitolului se proiectează un circuit care are multe caracteristici comune cu un procesor. Metodologia de proiectare este bazată pe analiza transferurilor dintre elementele de memorare din circuit (RTL – *Register Transfer Level*). Se introduc conceptele de cod mașină (instrucțiune în format binar), memorie de instrucțiuni, contor de program și bloc de registre.

Proiectarea procesorului MIPS, în varianta ciclu unic, este prezentată în capitolul 9. Se prezintă concepte generale despre arhitectura setului de instrucțiuni, apoi arhitectura setului de instrucțiuni MIPS, cu subsetul de instrucțiuni pentru care se proiectează procesorul. Se aplică metodologia de proiectare bazată pe transferul RTL pentru a construi complet procesorul MIPS, cu calea de date și unitatea de control. Abordarea este similară cu cea propusă original de Hennessy și Patterson, dar se pune accent pe interpretarea fiecărui detaliu din descrierea RTL a instrucțiunilor, cu scopul de a identifica ușor componentele procesorului și conexiunile dintre ele. În plus, se construiesc căi parțiale de date pentru toate instrucțiunile, căi care sunt gradual combinate pentru a obține calea de date finală a procesorului.

În capitolul 10 se explică concis traseul pe care îl urmează un program: descrierea folosind un limbaj de programare, compilarea, încărcarea în memorie și lansarea în execuție de către sistemul de operare. Ca studiu de caz, se consideră o secțiune de program scrisă în limbajul C, care este tradusă în limbajul de asamblare suportat de procesorul MIPS proiectat anterior. Apoi, instrucțiunile în limbaj de asamblare sunt traduse în cod mașină, care, odată încărcat în memoria de instrucțiuni a procesorului MIPS, poate fi executat.

În ultimul capitol se discută posibilități de extindere a procesorului, în special prin adăugarea de noi instrucțiuni pentru a suporta toate instrucțiunile din set. Ca exemplu, se alege o instrucțiune nouă și se prezintă cum, aplicând aceiași pași ca la proiectarea inițială, se extinde procesorul MIPS pentru a suporta și instrucțiunea selectată.

În Bibliografie se prezintă, selectiv, principalele referințe recomandate ca lectură suplimentară. Această carte se dorește a fi o introducere rapidă în arhitectura calculatoarelor și nu o alternativă pentru cărțile fundamentale din domeniu. Odată înțelese conceptele din prezenta carte, autorul recomandă ca studiu suplimentar cel puțin cărțile scrise de Harris (Harris & Harris, 2013) și Patterson și Hennessy, în edițiile actuale sau anterioare (Patterson & Hennessy, 2013), (Hennessy & Patterson, 2011) / (Hennessy & Patterson, 2017). Astfel, aceștia se vor familiariza cu concepte mai avansate, ca ierarhii de memorie, planificare dinamică și execuție speculativă, procesoare grafice și, mai ales, cu unele modele de procesoare reprezentative folosite în sistemele actuale de calcul.

Autorul vă dorește o lectură plăcută și utilă!

## 2. Reprezentarea numerelor. Sisteme de numerație

În acest capitol se prezintă aspecte legate de reprezentarea numerelor în sistemele de calcul și aritmetica numerelor binare, pentru numere întregi (cu sau fără semn). Sunt prezentate sistemele de numerație poziționale în baza 10, 2 și 16.

Sistemele de numerație prezentate sunt sisteme poziționale, unde un număr este format dintr-o succesiune ordonată de cifre, iar poziția unei cifre în număr este strâns legată de valoarea pe care o are cifra în cadrul numărului. Pozițiile cele mai semnificative sunt înspre stânga numărului, iar cele din dreapta sunt cele mai puțin semnificative.

Nu se prezintă deloc detalii legate de reprezentarea numerelor reale, cu formatul în virgulă mobilă. Pentru aceste subiecte se recomandă consultarea în detaliu a bibliografiei.

### 2.1. Reprezentarea numerelor în baza 10. Sistemul zecimal

Acesta este cel mai cunoscut sistem de reprezentare a numerelor. Pentru a ușura înțelegerea următoarelor reprezentări, în acest subcapitol se vor sublinia caracteristicile sistemului zecimal. Chiar dacă aceste caracteristici sunt triviale, fiind uzuale pentru toți cititorii acestei cărți, ele sunt prezentate în detaliu deoarece aceleași principii se aplică și pentru celelalte sisteme de numerație. Aceste caracteristici vor fi mai ușor de înțeles, făcând permanent analogia cu sistemul zecimal, în subcapitolele care urmează.

Baza sistemului zecimal este 10, iar cifrele sunt:

0, 1, 2, 3, 4, 5, 6, 7, 8, 9

Ca în orice bază de numerație, un număr de mai multe cifre conține cifre care au o pondere diferită în valoarea numărului. Ponderea unei cifre este dată de poziția ei: cu cât este mai la stânga, cu atât are o pondere mai mare. Ponderea unei cifre este calculată ca 10 ridicat la o putere egală cu poziția cifrei, de aici și denumirile uzuale folosite:

<i>Cifra</i>	<i>Pondere</i>
Unităților	$10^0 = 1$
Zecilor	$10^1 = 10$
Sutelor	$10^2 = 100$
Miilor	$10^3 = 1000$
Zecilor de mii	$10^4 = 10\ 000$
Sutelor de mii	$10^5 = 100\ 000$
...	...

Numerotarea poziției se face de la dreapta spre stânga, cifra cea mai din dreapta, a unităților, având poziția 0. În continuare, sunt prezentate exemple de numere scrise în baza 10 (*observație*: baza se scrie ca indice, în dreapta-jos lângă număr), cu contribuția fiecărei cifre la valoarea zecimală a numărului:

$$\begin{aligned}
 12345_{10} &= 1 * 10^4 + 2 * 10^3 + 3 * 10^2 + 4 * 10^1 + 5 * 10^0 \\
 7_{10} &= 7 * 10^0 \\
 5628_{10} &= 5 * 10^3 + 6 * 10^2 + 2 * 10^1 + 8 * 10^0 \\
 1001_{10} &= 1 * 10^3 + 0 * 10^2 + 0 * 10^1 + 1 * 10^0
 \end{aligned}$$

În cazul operației de adunare dintre două numere scrise în baza 10, se adună cifrele de pe pozițiile corespondente, pornind de la cifra unităților. Dacă pentru o pereche de cifre rezultatul este de două cifre  $cx$  (depășește cifra maximă 9), atunci la următoarea pereche de cifre se va aduna cifra zecilor  $c$  obținută pentru perechea curentă. Acest excedent care se adună la următoarea poziție de cifre poartă denumirea de *transport* (eng. carry).

În rândurile următoare este trasat un exemplu de adunare a două numere în baza 10:

$$\begin{array}{r} 3\ 6\ 4\ 9\ + \\ \underline{1\ 4\ 2\ 3} \\ ?\ ?\ ?\ ? \end{array}$$

*Pasul 1:*  $9 + 3 = 12$ , transport = 1

$$\begin{array}{r} \cdot \\ 3\ 6\ 4\ 9\ + \\ \underline{1\ 4\ 2\ 3} \\ ?\ ?\ ?\ 2 \end{array}$$

*Pasul 2:*  $4 + 2 + 1 = 7$

$$\begin{array}{r} 3\ 6\ 4\ 9\ + \\ \underline{1\ 4\ 2\ 3} \\ ?\ ?\ 7\ 2 \end{array}$$

*Pasul 3:*  $6 + 4 = 10$ , transport = 1

$$\begin{array}{r} \cdot \\ 3\ 6\ 4\ 9\ + \\ \underline{1\ 4\ 2\ 3} \\ ?\ 0\ 7\ 2 \end{array}$$

*Pasul 4, ultim:*  $3 + 1 + 1 = 5$

$$\begin{array}{r} 3\ 6\ 4\ 9\ + \\ \underline{1\ 4\ 2\ 3} \\ 5\ 0\ 7\ 2 \end{array}$$

În ciuda ușurinței cu care oamenii folosesc sistemul de numerație zecimal, acest sistem este greu de implementat fizic, în mod direct, în circuite digitale. Acest impediment este explicat simplist în următoarele idei. Un circuit digital de bază ar trebui să stocheze / reprezinte valorile unei cifre din sistemul de numerație folosit. O cifră este reprezentată într-un circuit ca o valoare particulară pentru una dintre proprietățile curentului electric, tensiunea/voltajul fiind proprietatea universal folosită. Realizarea unui circuit digital care să genereze 10 niveluri discrete de tensiune (pentru a reprezenta o cifră a sistemului zecimal) este mult prea complexă și costisitoare. Complexitatea crescută a circuitului de bază ar duce, în cascadă, la o complexitate și mai mare pentru circuitele de nivel înalt (de exemplu, circuitele pentru operații aritmetice). Ca urmare, în calculatoare este folosit cel mai simplu sistem de numerație, sistemul binar, pentru a reprezenta numere și pentru a efectua operații.



## 2.2. Reprezentarea numerelor în baza 2. Sistemul binar

În cazul sistemului de numerație binar, baza este 2, iar cifrele posibile sunt 0 sau 1. Se aplică același principiu ca la baza 10, și anume că ponderea unei cifre este dată de poziția ei: cu cât este mai la stânga, cu atât are o pondere mai mare. Diferența este la valoarea zecimală a ponderii, care este calculată ca 2 (baza) ridicat la o putere egală cu poziția cifrei. Cea mai din dreapta cifră are poziția 0.

Exemple de numere în sistem binar (sau simplu, binar):

Număr (în baza 2)
$1_2$
$0_2$
$101_2$
$10_2$
$11001011_2$
$10000000_2$

Pentru a calcula valoarea numărului binar în sistemul zecimal, se însumează valoarea fiecărei cifre înmulțite cu ponderea sa:

$$\begin{aligned}
 1_2 &= 1 * 2^0 = 1_{10} \\
 0_2 &= 0 * 2^0 = 0_{10} \\
 101_2 &= 1 * 2^2 + 0 * 2^1 + 1 * 2^0 = 4 + 1 = 5_{10} \\
 10_2 &= 1 * 2^1 + 0 * 2^0 = 2_{10} \\
 11001011_2 &= 1 * 2^7 + 1 * 2^6 + 1 * 2^3 + 1 * 2^1 + 1 * 2^0 = 128 + 64 + 8 + 2 + 1 \\
 &= 203_{10}
 \end{aligned}$$

În domeniul calculatoarelor, o cifră a sistemului binar este reprezentată de *bit* (pl. *biți*), acesta fiind unitatea de bază folosită pentru reprezentarea informației. Bitul, având doar două valori posibile, este reprezentat fizic de un circuit care are două stări. De aici rezultă o complexitate redusă a circuitului comparativ cu un sistem de reprezentare cu baza mai mare (de exemplu, sistemul zecimal). Cele două stări sunt date de tensiunea de ieșire a circuitului: valorile uzual cunoscute sunt 0V versus 5V. În prezent, pentru un consum cât mai redus de putere, nivelul mare de tensiune ajunge până la 1V sau chiar sub (pentru anumite circuite). Valorile folosite pentru cele două stări, în literatură și în această carte, sunt 0/1, LOW/HIGH sau, în contextul algebrei booleene, FALSE/TRUE (fals/adevărat).

Grupurile de biți consecutivi sunt denumite în mod specific, în general având dimensiuni care sunt puteri ale lui 2:

- *Semi-octet* (eng. half-byte sau nibble) – un grup de 4 biți consecutivi
- *Octet* (eng. byte) – grupare de 8 biți
- *Semi-cuvânt* (eng. half-word) – grupare de 2 octeți, respectiv 16 biți
- *Cuvântul* (eng. word) – grupare de 4 sau 8 octeți (sau mai mult), respectiv 32 sau 64 de biți.

*Cuvântul* este o noțiune importantă pentru o arhitectură de calcul, reprezentând unitatea de date care poate fi procesată în mod tipic de operațiile/instrucțiunile arhitecturii.

La procesoarele actuale, valorile uzuale pentru dimensiunea cuvântului sunt de 32 sau 64 de biți (coexistă pe multe arhitecturi).

Valorile care necesită un număr mai mic de biți pentru reprezentare, dar trebuie reprezentate pe un număr impus de biți, se completează pe cifrele mai semnificative (cele din stânga) cu biți nuli. În exemplul de mai jos, 13 este reprezentat pe diferitele grupuri de biți standard:

Baza 10	Baza 2	Semi-octet	Octet	Semi-cuvânt
13 <sub>10</sub>	1101 <sub>2</sub>	1101 <sub>2</sub>	0000 1101 <sub>2</sub>	0000 0000 0000 1101 <sub>2</sub>

Conversia unui număr din baza 10 în baza 2 se face prin împărțiri repetate cu valoarea bazei destinație, și anume 2. După fiecare împărțire se reține restul, acesta reprezentând una dintre cifrele binare ale numărului, și se repetă împărțirea pe câtul rămas, până când se ajunge la o valoare nulă a câtului. Resturile obținute (0 sau 1) se iau în ordine inversă și astfel se obține reprezentarea binară. Mai jos este prezentată procedura de conversie pentru numărul zecimal 94, rezultatul obținut fiind 94<sub>10</sub>=1011110<sub>2</sub>.

	Număr	Cât	Rest	Rezultat în baza 2
<i>pasul 1</i>	94 : 2 =	47	0	<b>1 0 1 1 1 1 0</b>
<i>pasul 2</i>	47 : 2 =	23	1	
<i>pasul 3</i>	23 : 2 =	11	1	
<i>pasul 4</i>	11 : 2 =	5	1	
<i>pasul 5</i>	5 : 2 =	2	1	
<i>pasul 6</i>	2 : 2 =	1	0	
<i>pasul 7</i>	1 : 2 =	0	1	
stop				

În cazul numerelor de ordinul zecilor/sutelor, cu o bună cunoaștere a primelor puteri ale lui 2 (2<sup>0</sup>=1, 2<sup>2</sup>=4, 2<sup>3</sup>=8, 2<sup>4</sup>=16, 2<sup>5</sup>=32, 2<sup>6</sup>=64, 2<sup>7</sup>=128, 2<sup>8</sup>=256), se pot descompune numerele în sume ale puterilor lui 2, rezultând reprezentarea binară. Fiecare termen din sumă corespunde câte unei cifre de 1 în reprezentarea binară, având poziția dată de puterea lui 2 din termenul respectiv (poziția 0 este cea mai din dreapta). Exemplu (se începe cu cel mai mare termen, iar apoi restul sumei se completează cu următorii termeni):

$$\begin{aligned}
 94_{10} &= 64 + 30 = \dots = 64 + 16 + 8 + 4 + 2 = \\
 &= 1 * 2^6 + 1 * 2^4 + 1 * 2^3 + 1 * 2^2 + 1 * 2^1 \Rightarrow (1 \text{ pe pozițiile } 6, 4, 3, 2, 1) \Rightarrow \\
 94_{10} &= \mathbf{1011110_2}
 \end{aligned}$$

Ca interval de reprezentare, pentru numere pozitive, pe  $n$  biți, se pot reprezenta 2<sup>n</sup> numere distincte, din intervalul [0, 2<sup>n</sup>-1]. Pe un semi-octet se pot reprezenta numerele de la 0 la 15 inclusiv, pe octet de la 0 la 255, iar pe cuvântul de 32 de biți numerele de la 0 la 4,294,967,295.

### 2.2.1. Aritmetica numerelor în baza 2. Reprezentarea în complement față de 2

În această secțiune se vor prezenta în detaliu operațiile de adunare și scădere pentru numere binare, precum și o reprezentare particulară a numerelor binare necesară pentru numerele cu semn, și anume *complementul față de 2*.

În cazul numerelor în baza 2, adunarea se face la fel ca în sistemul zecimal, ținând cont de adunările elementare ale cifrelor binare:

$$\begin{array}{rclcl} 0 & + & 0 & = & 0_2 \\ 0 & + & 1 & = & 1_2 \\ 1 & + & 0 & = & 1_2 \quad \text{comutativitate!} \\ 1 & + & 1 & = & 10_2 \end{array}$$

În cazul ultimei adunări de mai sus, justificarea este simplă: dacă efectuăm operația în sistemul zecimal, atunci  $1 + 1 = 2$ , rezultat care transformat în sistemul binar este  $10_2$ .

Pentru numere reprezentate binar pe mai mulți biți se adună cifrele de pe pozițiile corespondente, pornind de la poziția cea mai puțin semnificativă (cea mai din dreapta). Dacă pentru o pereche de cifre rezultatul adunării este de două cifre binare (depășește valoarea 1), atunci la următoarea pereche de cifre se va aduna transportul obținut (1 în cazul adunării  $1 + 1$ ).

În rândurile următoare este trasat un exemplu de adunare a două numere în baza 2, reprezentate pe 4 biți:

$$\begin{array}{r} 0\ 1\ 1\ 1\ + \\ \underline{0\ 0\ 1\ 0} \\ \text{? ? ? ?} \end{array}$$

*Pasul 1:*  $1 + 0 = 1$

$$\begin{array}{r} 0\ 1\ 1\ 1\ + \\ \underline{0\ 0\ 1\ 0} \\ \text{? ? ? } 1 \end{array}$$

*Pasul 2:*  $1 + 1 = 1\ 0$ , transport = 1

$$\begin{array}{r} \cdot \\ 0\ 1\ 1\ 1\ + \\ \underline{0\ 0\ 1\ 0} \\ \text{? ? } 0\ 1 \end{array}$$

*Pasul 3:*  $1 + 0 + 1$  (transportul) =  $1\ 0$ , transport = 1

$$\begin{array}{r} \cdot \\ 0\ 1\ 1\ 1\ + \\ \underline{0\ 0\ 1\ 0} \\ \text{? } 0\ 0\ 1 \end{array}$$

*Pasul 4, ultim:*  $0 + 0 + 1 = 1$

$$\begin{array}{r} 0\ 1\ 1\ 1\ + \\ \underline{0\ 0\ 1\ 0} \\ 1\ 0\ 0\ 1 \end{array}$$

În cazul în care se obține transport pe poziția de bit cea mai semnificativă (cel mai din stânga bit) apare o situație de depășire (eng. *overflow*). Practic, rezultatul adunării nu poate fi reprezentat pe numărul de biți ai operanzilor și ar fi necesari biți suplimentari pentru reprezentarea corectă.

În exemplele discutate până acum nu au fost incluse numere întregi negative. Prima problemă de discutat este cum pot fi reprezentate numerele negative în sistemul binar.

Informația de semn trebuie codificată tot prin intermediul biților din reprezentare. Modalitatea directă, cunoscută ca *semn/magnitudine*, este de a alege un bit care să codifice semnul, 0 pentru numere pozitive și 1 pentru numere negative. Acest bit poartă numele de *bit de semn* și este situat pe poziția cea mai semnificativă. Restul biților reprezintă *magnitudinea* numărului (valoarea în modul). Mai jos se prezintă exemple de numere întregi, pozitive sau negative, reprezentate pe 8 biți, în varianta *semn/magnitudine*.

Număr <sub>10</sub>	În binar, <i>semn/magnitudine</i>
1	00000001
33	00100001
-1	10000001
-33	10100001
0	00000000 sau 10000000

Intervalul de reprezentare este dat de cei  $n-1$  biți rămași pentru magnitudine. De exemplu, în cazul numerelor pe 8 biți, intervalul de reprezentare este dat de cei 7 biți ai magnitudinii:  $[-127...+127]$ . Se pot reprezenta 255 de numere distincte, în loc de 256 cât permit cei 8 biți, 0 având o reprezentare redundantă cu două combinații posibile (+0/-0). Pentru a elimina acest neajuns, dar și din alte motive care urmează a fi prezentate, în domeniul calculatoarelor se folosește în mod universal o altă reprezentare pentru numere binare, și anume *complementul față de 2*.

### 2.2.1.1. Reprezentarea în complement față de 2

În reprezentarea în complement față de 2 se folosește de asemenea bitul de semn, pe poziția cea mai semnificativă. Pentru numere pozitive, reprezentarea se face exact ca în baza 2, bitul de semn având valoarea 0. Intervalul de reprezentare pentru numere pozitive, pe  $n$  biți, este  $[0...+2^{n-1}-1]$ . De exemplu, pe 8 biți se pot reprezenta numere pozitive între 0 și +127, inclusiv.

Pentru un număr negativ  $-a$  ( $a$  este modulul numărului) reprezentarea în complement față de 2, pe  $n$  biți, este egală cu reprezentarea binară a numărului  $2^n - a$  (de aici și ideea de complement). Din motive care vor deveni clare în capitolele următoare (simplitate pentru implementarea fizică), reprezentarea se obține în doi pași simpli:

1. Se reprezintă numărul pozitiv  $a$  pe  $n$  biți și se neagă biții (0 devine 1, și viceversa)
2. Se adună 1 la rezultatul de la primul pas. Dacă în urma adunării se obține valoarea 0 pentru bitul de semn înseamnă că numărul  $-a$  necesită mai mulți biți pentru reprezentare corectă în complement față de 2.

În Tabel 2.1 sunt prezentate exemple de conversie, pe 8 biți. Se folosește indicele  $c_2$  pentru a sublinia că o reprezentare binară este în complement față de 2.

Tabel 2.1. Procesul de obținere a reprezentării în complement față de 2, exemple pe 8 biți

Număr	$+8_{10}$	$-1_{10}$	$-34_{10}$	$-128_{10}$	$-129_{10}$
<i>Pasul 1.a</i>	0000 1000 <sub>2</sub>	0000 0001 <sub>2</sub>	0010 0010 <sub>2</sub>	1000 0000 <sub>2</sub>	1000 0001 <sub>2</sub>
<i>Pasul 1.b</i>	-	1111 1110 <sub>2</sub> +	1101 1101 <sub>2</sub> +	0111 1111 <sub>2</sub> +	0111 1110 <sub>2</sub> +
<i>Pasul 2</i>	-	1	1	1	1
<i>Rezultat</i>	0000 1000 <sub>c2</sub>	1111 1111 <sub>c2</sub>	1101 1110 <sub>c2</sub>	1000 0000 <sub>c2</sub>	0111 1111 <sub>c2</sub>
<i>Observații</i>	nr. pozitiv, nu se aplică cei 2 pași				bit de semn = 0 în c2, nu se poate reprezenta pe 8 biți

Legat de exemplul din tabelul anterior, se poate observa ideea de complement pentru numerele negative: reprezentarea în c2 a lui  $-a$  este echivalentă cu reprezentarea binară a numărului  $256 - a$  (ex. pentru  $-34$ , rezultatul pe biți 1101 1110 reprezintă numărul pozitiv  $222 = 256 - 34$ !). În mod particular, reprezentarea c2 a lui  $-128$  este identică cu a lui 128 pentru că  $-128$  are complementul  $256 - 128 = 128$ . Acest aspect nu implică nicio problemă de suprapunere a reprezentărilor, întrucât pe 8 biți, în c2, se reprezintă doar numerele pozitive până la  $+127$  inclusiv.

Intervalul de numere negative reprezentabile în c2 pe  $n$  biți este  $[-2^{n-1} \dots -1]$ . Față de reprezentarea *semn/magnitudine* apare în plus un număr negativ care poate fi reprezentat, și anume  $-2^{n-1}$ , pe poziția eliberată prin eliminarea reprezentării dublate a lui 0.

În cazul numerelor întregi pozitive/negative, pe semi-octet se poate reprezenta intervalul  $[-8 \dots +7]$ , pe octet  $[-128 \dots +127]$ , iar pe cuvânt de 32 biți  $[-2^{31} \dots +2^{31}-1] \Leftrightarrow$  intervalul  $[-2,147,483,648 \dots +2,147,483,647]$ .

Transformarea inversă, pornind de la reprezentarea în c2 a unui număr negativ pentru a obține numărul zecimal în modul, se face exact cu aceiași pași folosiți la transformarea directă: se neagă biții reprezentării, se adună 1 și se obține numărul pozitiv în binar. Exemple mai jos, pe 8 biți:

Tabel 2.2. Transformarea din reprezentarea în complement față de 2 în sistemul zecimal

Număr	$-9$	$-1$	$-34$
<i>Reprezentare c2</i>	1111 0111 <sub>c2</sub>	1111 1111 <sub>c2</sub>	1101 1110 <sub>c2</sub>
<i>Pasul 1.b (negare)</i>	0000 1000 +	0000 0000 +	0010 0001 <sub>2</sub> +
<i>Pasul 2 (+1)</i>	1	1	1
<i>Rezultat</i>	0000 1001 <sub>2</sub>	0000 0001 <sub>2</sub>	0010 0010 <sub>2</sub>
	$= 9_{10}$	$= 1_{10}$	$= 34_{10}$

Generalizând, dacă numărul întreg  $x$  (pozitiv sau negativ) este reprezentat în c2 pe  $n$  biți, atunci reprezentarea în c2 a numărului  $-x$  se obține din reprezentarea în c2 a lui  $x$  prin (1) negarea biților urmată de (2) adunarea cu 1.

### 2.2.1.2. Adunarea și scăderea în complement față de 2

*Adunarea* se face la fel ca pentru numerele binare, bit cu bit, indiferent de semnul numerelor, acesta fiind un avantaj al reprezentării în complement față de 2. Rezultatul este corect doar dacă poate fi reprezentat pe numărul de biți disponibili (la fel ca termenii/operanzii sumei), în caz contrar apare o depășire. Situația de depășire, spre deosebire de cazul adunării numerelor pozitive (vezi mai sus), nu poate fi detectată strict din analiza transportului care apare de pe poziția bitului de semn (necesită o logică mai complexă, care nu se va detalia în această carte).

În tabelul următor sunt prezentate exemple de adunare pe 4 biți, inclusiv cu două situații de depășire (ultimele 2 coloane).

Tabel 2.3. Exemple pentru adunarea de numere (cu semn) pe 4 biți

Suma	$3+(-4)$	$(-4)+(-4)$	$7+7$	$(-7)+(-7)$
<i>Operand 1</i>	$0011_{c2} +$	$1100_{c2} +$	$0111_{c2} +$	$1001_{c2} +$
<i>Operand 2</i>	$1100_{c2}$	$1100_{c2}$	$0111_{c2}$	$1001_{c2}$
<i>Rezultat</i>	$1111_{c2}$	$1000_{c2}$	$1110_{c2}$	$0010_{c2}$
	$= -1_{10}$	$= -8_{10}$	$= 2_{10}$	$= 2_{10}$

În cazul celor două situații de depășire (ultimele două coloane), rezultatul este interpretat ca număr cu semn, reprezentat în complement față de 2. Dacă s-ar interpreta numerele binare ca numere fără semn, atunci prima operație ar fi corectă ( $7 + 7 = 14$ ), doar a doua situație ar avea depășire ( $9 + 9 = 2$ ). Acesta este un alt avantaj al reprezentării în complement față de 2: operanzii și rezultatul pot fi interpretate fie ca numere cu semn, fie fără semn. Rezultatul este corect în ambele interpretări, cu condiția să nu existe depășiri. În limbajul C, de exemplu, pe 8 biți se pot declara numere cu semn, de tipul *char*, sau numere fără semn, de tipul *unsigned char*.

Pentru a efectua scăderea se apelează la un truc simplu, observat deja în tabelul anterior. Operația de scădere este rescrisă ca o adunare cu negativul scăzătorului (care se obține printr-o negare a biților și o adunare cu 1, vezi mai sus):

$$a - b = a + (-b)$$

Acesta este principalul avantaj pentru reprezentarea în complement față de 2: se refolosește logica (circuitul) de la adunare. Astfel, nu sunt necesare două circuite fizice distincte, rămânând valabile regulile pentru depășire de la adunare (rezultatul e corect doar dacă se poate reprezenta pe același număr de biți).

Exemple de scădere cu numere reprezentate pe 4 biți în  $c2$ :

Tabel 2.4. Scăderea, rescrisă ca adunare, în complement față de 2

Diferența	$3 - 4 = 3 + (-4)$	$-4 - 2 = (-4) + (-2)$	$7 - 7 = 7 + (-7)$
<i>Operand 1</i>	$0011_{c2} +$	$1100_{c2} +$	$0111_{c2} +$
<i>Operand 2</i>	$1100_{c2}$	$1110_{c2}$	$1001_{c2}$
<i>Rezultat</i>	$1111_{c2}$	$1010_{c2}$	$0000_{c2}$
	$= -1_{10}$	$= -6_{10}$	$= 0_{10}$

### 2.3. Reprezentarea numerelor în baza 16. Sistemul hexazecimal

Reprezentarea în binar este strâns legată de circuitele fizice, însă pentru depanatorul/programatorul de calculatoare sunt greu de vizualizat și interpretat șirurile lungi de biți consecutivi.

O soluție elegantă pentru vizualizare este utilizarea unui sistem de numerație cu o bază mai mare, care să permită reprezentarea cu o singură cifră a unui grup de biți. Selecția bazei se face printr-un compromis, astfel încât să se reducă semnificativ lungimea în cifre pentru reprezentarea unui șir de biți, dar, în același timp, să se păstreze o granularitate suficient de mică pentru interpretarea/reconversia ușoară înapoi la binar. Baza 16 este cea

care răspunde acestui compromis și permite reprezentarea completă a unui semi-octet (4 biți) pe o cifră hexazecimală.

Cifrele (sau simbolurile) bazei 16 sunt:

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F

Cifrele de la A, B, C, D, E și F au valoarea echivalentă în sistemul zecimal de 10, 11, 12, 13, 14, respectiv 15.

Conversia din baza 16 în baza 10 se face aplicând aceleași principii ca în subcapitolele anterioare. Pentru a calcula valoarea unui număr hexazecimal în sistemul zecimal, se însumează valorile cifrelor înmulțite cu ponderile lor (16 la puterea dată de poziția cifrei):

$$321_{16} = 3 * 16^2 + 2 * 16^1 + 1 * 16^0 = 801_{10}$$

$$10_{16} = 1 * 16^1 + 0 * 16^0 = 16_{10}$$

$$ABC_{16} = 10 * 16^2 + 11 * 16^1 + 12 * 16^0 = 2748_{10}$$

Echivalența cifrelor hexazecimale în sistemul binar este prezentată mai jos, cu observația că reprezentarea binară este ușor de calculat folosind a doua metodă de conversie în binar prezentată în pagina 8 (exemplu:  $D = 13_{10} = 8+4+1 = 1101_2$ ):

Cifră hexazecimală	Reprezentare binară, 4 biți
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
A	1010
B	1011
C	1100
D	1101
E	1110
F	1111

Data fiind echivalența dintre semi-octet și cifra hexazecimală, este ușor de calculat numărul de cifre hexazecimale necesare pentru grupurile standard de biți. Astfel, valoarea unui octet se poate reprezenta cu 2 cifre hexazecimale (intervalul de reprezentare  $0...255_{10} \Leftrightarrow 0..FF_{16}$ ), semi-cuvântul cu 4 cifre (intervalul  $0..FFFF_{16}$ ), iar cuvântul de 32 de biți cu 8 cifre (intervalul  $0..FFFF FFFF_{16}$ ).

Conversia unui număr din baza 10 în baza 16 se face similar cu metoda descrisă pentru baza 2, prin împărțiri repetate cu valoarea bazei destinație, și anume 16. Resturile obținute la fiecare pas vor fi una dintre cifrele hexazecimale 0, 1, 2, ..., A, B, C, D, E, F și se

vor lua în ordine inversă pentru a obține reprezentarea hexazecimală. Procedura de conversie pentru numărul zecimal 2988 este prezentată mai jos, rezultatul obținut fiind  $2988_{10} = \text{BAC}_{16}$ .

	Număr		Cât	Rest	Rezultat în baza 16
<i>pasul 1</i>	2988	:	16 =	186	<b>12=C</b>
<i>pasul 2</i>	186	:	16 =	11	<b>10=A</b>
<i>pasul 3</i>	11	:	16 =	0	<b>11=B</b>
Stop					

Conversia unui număr din baza 2 în baza 16 și invers se face simplu, ținând cont de reprezentarea binară pe 4 biți a unei cifre hexazecimale.

Pentru conversia din baza 2 în baza 16, biții consecutivi se grupează câte 4, de la dreapta la stânga, completând la nevoie cu biți nuli pentru biții cei mai semnificativi. Fiecare grup de 4 biți este transformat apoi în cifra hexazecimală echivalentă. Exemplul următor prezintă o astfel de conversie:

$$10001011001011_2 = 10\ 0010\ 1100\ 1011_2 = 0010\ 0010\ 1100\ 1011_2 = \mathbf{2\ 2\ C\ B}_{16}$$

Viceversa, pentru conversia din baza 16 în baza 2, fiecare cifră hexazecimală se înlocuiește cu reprezentarea ei binară pe 4 biți. Se poate vizualiza exemplul anterior, parcurs de la dreapta spre stânga.



### 3. Algebra booleană și porți logice

Algebra booleană este un tip particular de algebră care are ca subiect variabilele care au două valori posibile,  $1 = \text{adev\c{a}rat}$  (eng. true) sau  $0 = \text{fals}$  (eng. false), și operațiile posibile între astfel de variabile. Folosind regulile algebrei booleene, se pot proiecta, descrie și optimiza circuite digitale. Acest lucru este posibil deoarece principalele operații ale algebrei booleene au asociate porți logice, care de fapt sunt circuite de bază care implementează operația asociată.

În acest capitol se vor prezenta mai întâi operațiile de bază, cu porțile logice asociate, după care se vor discuta legile (teoremele) algebrei booleene. Acestea pot fi folosite pentru a simplifica și rescrie, într-o formă convenabilă, ecuații booleene complexe (implicit se vor obține circuite optimizate). Noțiunile prezentate sunt fundamentale pentru *proiectarea logică* (eng. *logic design*).

#### 3.1. Algebra booleană – noțiuni de bază

Se disting două componente principale: *variabile* și *operații*.

*Variabilele* se denumesc cu simboluri literale (eventual cu indici, ex: A, B, A<sub>1</sub>, C<sub>3</sub> etc.) și pot avea doar una dintre cele două valori permise de algebră. Valorile binare de 0 și 1 se folosesc de obicei în loc de *fals* / *adev\c{a}rat*.

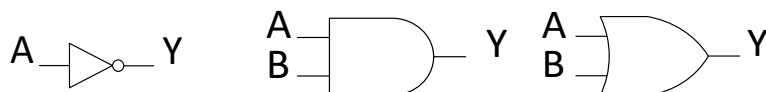
*Operațiile* de bază sunt fie unare (cu un singur operand de intrare), fie binare (cu doi operanzi de intrare – cu posibilitate de extindere spre mai mulți). În formalismul următor se vor folosi variabilele A și B ca operanzi de intrare și Y pentru rezultatul operației. Cele 3 operații de bază sunt:

1. NEGARE (eng. NOT):  $Y = \bar{A}$ 
  - Se alege valoarea opusă a operandului de intrare
2. SUMA, cunoscută și ca SAU logic (eng. OR):  $Y = A + B$ 
  - Rezultatul este 1 dacă oricare dintre operanzii de intrare este 1 (dacă A=1 sau B=1), altfel 0
3. PRODUSUL, cunoscut și ca ȘI logic (eng. AND):  $Y = A \cdot B$ 
  - Rezultatul este 1 dacă ambii operanzi de intrare sunt 1 (dacă A=1 și B=1), altfel 0

În cazul expresiilor cu mai multe operații, prioritatea standard a acestor operații este: prima dată se evaluează negările, apoi produsele, și la final sumele. Prioritatea se poate modifica folosind paranteze. De exemplu:

$Y = A \cdot \bar{B} + C$	$Y = A \cdot (\bar{B} + C)$
ordinea evaluării: (1) B negat, (2) produsul, (3) suma	ordinea evaluării: (1) B negat, (2) suma, (3) produsul

*Porțile logice* (eng. logic gates) pentru aceste operații de bază sunt prezentate în tabelul următor. În contextul porților logice, care au și implementare sub formă de circuite fizice (nivelurile logice sunt reprezentate de niveluri de tensiune diferite), operanzii de intrare se vor referi ca *intrări* sau *semnale de intrare* (eng. input signals), iar rezultatul ca *ieșire* sau *semnal de ieșire* (eng. output signal).



Inversor (negare)\*

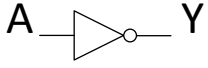
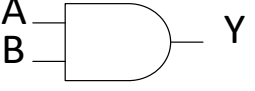
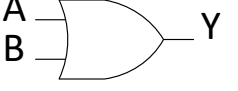
Poartă ȘI/AND

Poartă SAU/OR

\* Observație: *negarea* este subliniată de prezența 'o' pe ieșirea porții

Pentru a descrie rezultatul unei ecuații booleene sau, echivalent, funcționarea unei porți logice, se folosesc *tabele de adevăr*. În aceste tabele se specifică, pentru fiecare combinație de valori a operanzilor de intrare, care este valoarea rezultatului ecuației, respectiv a semnalului de ieșire pentru poarta logică:

Tabel 3.1. Tabele de adevăr pentru porțile logice de bază

NOT		AND			OR		
							
A	Y	A	B	Y	A	B	Y
0	1	0	0	0	0	0	0
1	0	0	1	0	0	1	1
		1	0	0	1	0	1
		1	1	1	1	1	1

Alte operații booleene / porți logice frecvent utilizate sunt:

1. SAU EXCLUSIV (eng. eXclusive OR – XOR):

$$Y = A \oplus B$$

- Rezultatul este 1 dacă operanzii de intrare sunt diferiți (unul e 0, celălalt e 1), altfel 0. Utilă pentru a detecta valori distincte pe intrări.

2. SAU NEGAT (eng. Not OR – NOR):

$$Y = \neg(A + B) \text{ sau } Y = \overline{A + B}$$

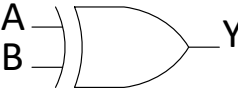
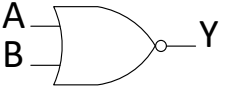
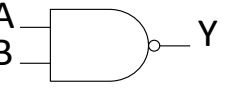
- Rezultatul este 1 dacă operanzii de intrare sunt nuli, altfel 0. Este obținută dintr-o poartă SAU cu rezultatul negat. Utilă pentru a detecta valori nule pe toate intrările.

3. ȘI NEGAT (eng. Not AND – NAND):

$$Y = \neg(A \cdot B) \text{ sau } Y = \overline{A \cdot B}$$

- Rezultatul este 1 dacă cel puțin un operand de intrare este nul, altfel 0. Este obținută dintr-o poartă ȘI cu rezultatul negat.

Tabel 3.2. Tabele de adevăr

XOR			NOR			NAND		
								
A	B	Y	A	B	Y	A	B	Y
0	0	0	0	0	1	0	0	1
0	1	1	0	1	0	0	1	1
1	0	1	1	0	0	1	0	1
1	1	0	1	1	0	1	1	0

Transformarea unei ecuații într-un circuit format din porți logice se face ținând cont de prioritatea operațiilor: se începe cu porțile asociate cu operațiile prioritare, ieșirea (rezultatul) acestor porți se leagă la intrările următoarelor porți ce se adaugă la schemă, în ordinea priorității ș.a.m.d. Exemplu:

$$Y = \bar{A} + A \cdot (A + B) \quad (3.1)$$

Se începe (desenul următor, de urmărit de la stânga la dreapta) cu un circuit inversor pentru  $\bar{A}$  și cu o poartă SAU pentru  $A + B$ . Se adaugă o poartă ȘI pentru a calcula  $A \cdot (A + B)$ , după care ieșirea din inversor și cea din poarta ȘI se leagă la o poartă SAU pentru a-l calcula pe Y.

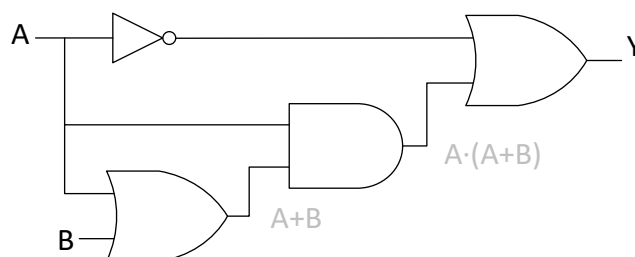


Fig. 3.1. Circuitul echivalent pentru ecuația (3.1)

### 3.2. Teoreme/legi specifice

Aceste teoreme reprezintă identități/relații cu variabile și operații booleene care pot fi folosite pentru a simplifica/optimiza ecuațiile booleene și, implicit, circuitele rezultate. Relațiile sunt de diferite niveluri de complexitate. Majoritatea sunt intuitive, însă unele sunt mai puțin evidente. În toate cazurile, relațiile se pot demonstra ușor folosind tabele de adevăr (cititorul este încurajat să facă asta pentru o mai bună înțelegere și aprofundare).

În continuare sunt prezentate majoritatea teoremelor, primele cinci fiind teoreme cu o singură variabilă, iar ultimele fiind teoreme cu două sau mai multe variabile. Pentru fiecare teoremă se oferă o scurtă explicație (uneori sub forma unei întrebări la care teorema răspunde). Unde e cazul, se prezintă ambele forme ale teoremei, pentru cele două operații binare de bază (sumă și produs).

1. *Identitatea* – dată fiind variabila A, cu ce valoare se poate efectua un produs sau o sumă astfel încât rezultatul să fie tot A?

$$A \cdot 1 = A \quad A + 0 = A$$

2. *Elementul nul* – dată fiind variabila  $A$ , cu ce valoare se poate efectua un produs sau o sumă astfel încât rezultatul să fie același, indiferent de valoarea lui  $A$ ?

$$A \cdot 0 = 0 \qquad A + 1 = 1$$

3. *Idempotența* – ce se întâmplă când se efectuează un produs sau o sumă având aceeași variabilă pe ambele intrări?

$$A \cdot A = A \qquad A + A = A$$

4. *Involuția* – ce se întâmplă dacă se aplică succesiv, de două ori, negarea pe o variabilă?

$$\bar{\bar{A}} = A$$

5. *Complementul* – care este rezultatul unui produs sau al unei sume dintre o variabilă și aceeași variabilă negată?

$$A \cdot \bar{A} = 0 \qquad A + \bar{A} = 1$$

6. *Comutativitatea* – similar cu conceptul din algebra clasică

$$A \cdot B = B \cdot A \qquad A + B = B + A$$

7. *Asociativitatea* – similar cu conceptul din algebra clasică

$$A \cdot (B \cdot C) = (A \cdot B) \cdot C \qquad (A + B) + C = A + (B + C)$$

8. *Distributivitatea* – similar cu conceptul din algebra clasică

$$A \cdot (B + C) = A \cdot B + A \cdot C$$

9. *Acoperirea* – apare o simplificare interesantă atunci când se face produsul dintre o variabilă  $A$  și o sumă în care variabila  $A$  apare ca operand.

$$A \cdot (A + B) = A$$

Demonstrație: Aplicăm *distributivitatea*, apoi, din *idempotență*, simplificăm  $A \cdot A = A$

$$A \cdot (A + B) = A \cdot A + A \cdot B = A + A \cdot B$$

Fiind un SAU între  $A$  și  $A \cdot B$ , în cazul în care  $A=1$ , expresia devine 1 (adică  $=A$ ) indiferent de valoarea lui  $B$ , iar dacă  $A$  este 0, expresia va deveni 0 ( $=A$ ) indiferent de valoarea lui  $B$ . De aici conceptul de acoperire:  $A$  acoperă toate cazurile pentru rezultat, fără să conteze valoarea lui  $B$ .

10. *Teorema lui De Morgan* – Prin negarea unui produs/sume de variabile se obține operația complementară sumă/produs aplicată pe variabilele negate. Nu vom intra în mai multe detalii deoarece în restul cărții nu se dau exemple de utilizare (exceptând o demonstrație în subcapitolul 5.4 pentru construcția unui sumator).

$$\overline{A_0 \cdot A_1 \cdot A_2 \cdot \dots \cdot A_n} = \overline{A_0} + \overline{A_1} + \overline{A_2} + \dots + \overline{A_n}$$

$$\overline{\overline{A_0} + \overline{A_1} + \overline{A_2} + \dots + \overline{A_n}} = \overline{A_0} \cdot \overline{A_1} \cdot \overline{A_2} \cdot \dots \cdot \overline{A_n}$$

Ca exemplu de minimizare, se consideră următoarea ecuație booleană. Fără minimizare, circuitul echivalent din porți logice ar necesita un inversor, trei porți SAU și două porți ȘI.

$$Y = (\bar{A} + A \cdot (A + B)) \cdot (A + B) \quad (3.2)$$

Se observă un caz direct de aplicare pentru teorema de *acoperire*, pentru subexpresia  $A \cdot (A + B)$ . Ecuația devine:

$$Y = (\bar{A} + A) \cdot (A + B) \quad (3.3)$$

Aplicând mai departe teorema *complementului* pentru  $\bar{A} + A$ , se obține:

$$Y = 1 \cdot (A + B) \quad (3.4)$$

La final, se aplică teorema *identității*:

$$Y = A + B \quad (3.5)$$

În această formă simplificată, circuitul se implementează simplu cu o singură poartă SAU. Astfel, se obține o optimizare pentru (avantajele următoare sunt universale, nu strict pentru exemplul dat):

1. *Resursele necesare*. Fiind necesare mai puține circuite componente (porți, în acest caz) rezultă un cost mai redus și un spațiu mai mic necesar pentru circuit
2. *Puterea consumată* de circuit. Tot datorită numărului redus de elemente rezultă un consum redus de putere, adică un cost redus de funcționare. În același timp, puterea consumată se disipă prin căldură, rezultând o temperatură scăzută în timpul funcționării circuitului
3. *Viteza* circuitului. Fiind mai puține porți logice înlănțuite, timpul de propagare pentru semnalul electric se reduce și rezultatul apare mai repede. Se va reveni la acest concept în capitolele următoare, pentru circuite mai complexe.

Un exemplu cotidian, familiar cititorilor, unde avantajele enumerate mai sus sunt evidente (chiar dacă nu se datorează doar acestui tip de optimizare!), este telefonul mobil inteligent. Acesta înglobează o putere mare de calcul într-un volum redus, putere consumată mică (timp crescut de funcționare strict pe baterie) și viteză mare de procesare (măsurată de obicei, ușor simplist, prin frecvența procesorului, în GigaHertzi - GHz).

### 3.3. Reprezentarea cu termeni canonici

În contextul unei funcții sau ecuații booleene, *termenul canonic* reprezintă o combinație a variabilelor de intrare, în forma inițială sau negată, ca produs sau sumă. Sunt două reprezentări cu termeni canonici pentru funcțiile booleene: *sumă de produse* și *produs de sume*.

În cazul reprezentării ca *sumă de produse* (eng. sum of products) pe care o vom detalia în continuare, termenul canonic (eng. minterm pentru suma de produse) este reprezentat de un produs între variabilele de intrare (forma normală sau negată). În continuare sunt prezentate exemple de ecuații sub formă de sume de produse, cu două sau trei variabile.

$$Y = \bar{A} \cdot B + A \cdot B + A \cdot \bar{B}$$

$$Y = \bar{A} \cdot \bar{B}$$

$$Y = \bar{A} \cdot B \cdot C + A \cdot B \cdot C + A \cdot B \cdot \bar{C}$$

Acest tip de reprezentare permite deducerea, cu o metodă ușor de aplicat, a unei ecuații booleene pornind de la tabela de adevăr. Se aplică pașii următori:

1. Se identifică liniile din tabelă (combinațiile de valori de intrare) pentru care rezultatul ecuației trebuie să fie 1 și se derivă termenul canonic pentru fiecare linie. Termenul canonic este un produs între variabilele de intrare (forma normală sau negată). Dacă valoarea unei variabile este 1 pentru linia respectivă, ea va fi folosită sub forma normală în produs, iar dacă este 0, se va folosi forma negată (complementară) a variabilei
2. Se deduce ecuația ca suma dintre termenii canonici identificați la punctul 1.

Urmează un exemplu de aplicare a acestei metode. Tabela de adevăr, pentru o ecuație cu 3 variabile (valorile pentru Y sunt alese aleatoriu), este prezentată în continuare.

A	B	C	Y
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	1

Pasul 1. Se identifică combinațiile de valori de intrare pentru care rezultatul e 1 și se scriu termenii canonici asociați. Pentru generalitate, în tabelul următor sunt prezentați toți termenii canonici, chiar dacă nu sunt necesari. Termenul canonic  $T$  pentru o linie trebuie să producă un rezultat de 1 dacă variabilele de intrare au ca valori combinația de pe linia respectivă. Din acest motiv se neagă în produsul respectiv variabilele de intrare care au valoarea 0 pentru linia asociată (rezultă un produs între valori de 1). Pentru orice altă combinație, de pe celelalte linii, termenul canonic  $T$  produce rezultatul 0.

Tabel 3.3. Tabela de adevăr cu termenii canonici pentru fiecare linie

A	B	C	Y	Termen canonic
0	0	0	0	$\bar{A} \cdot \bar{B} \cdot \bar{C}$
0	0	1	0	$\bar{A} \cdot \bar{B} \cdot C$
0	1	0	1	$\bar{A} \cdot B \cdot \bar{C}$
0	1	1	0	$\bar{A} \cdot B \cdot C$
1	0	0	1	$A \cdot \bar{B} \cdot \bar{C}$
1	0	1	1	$A \cdot \bar{B} \cdot C$
1	1	0	0	$A \cdot B \cdot \bar{C}$
1	1	1	1	$A \cdot B \cdot C$

Pasul 2. Ecuația se scrie ca o sumă între termenii canonici identificați la pasul 1:

$$Y = \bar{A} \cdot B \cdot \bar{C} + A \cdot \bar{B} \cdot \bar{C} + A \cdot \bar{B} \cdot C + A \cdot B \cdot C$$

Din acest moment, ecuația se poate optimiza folosind legile specifice ale algebrei booleene sau se pot folosi metode mai avansate (ex. diagrame Karnaugh etc.), pentru care cititorul este invitat să parcurgă bibliografia suplimentară pentru proiectare logică (Văcariu & Creț, 2012). Pentru cititorii interesați de detalii suplimentare legate de algebra booleană, porți logice, precum și implementarea și funcționarea fizică (la nivel de tranzistori) a porților logice, se recomandă studierea primelor două capitole din (Harris & Harris, 2013).

Se va aplica din nou această metodă în capitolul următor, pentru a construi gradual circuite mai complexe, utilizate frecvent în sistemele de calcul. Printre altele, aritmetica numerelor binare (descrisă în capitolul anterior) va fi dusă spre o implementare (circuit) folosind principiile de bază ale algebrei booleene și porți logice.





## 4. Circuite combinaționale și secvențiale – fundamente

Există două tipuri de circuite digitale care stau la baza unui procesor / sistem de calcul: circuite combinaționale și circuite secvențiale. Pentru a înțelege următoarele capitole este esențială discutarea proprietăților fundamentale ale acestor circuite.

Modelul teoretic discutat până acum, cu nivelurile logice 0 și 1, se transpune în implementarea fizică a circuitelor ca un model cu 2 niveluri diferite de tensiune, tensiunea LOW (scăzută) și HIGH (înalță). Concret, 0 logic este asociat cu 0 V, iar 1 logic cu 5 V (în funcție de circuit și de destinația lui, nivelul de tensiune folosit pentru 1 logic poate să fie chiar mai mic decât 1 V, pentru un consum scăzut de putere).

În cazul circuitelor *combinaționale*, orice modificare a semnalelor de intrare se propagă instantaneu (modelul ideal) în valoarea semnalului de ieșire. Altfel spus, ieșirea unui circuit combinațional depinde doar de valoarea curentă a semnalelor de intrare. În implementarea fizică, schimbarea semnalelor de intrare înseamnă de fapt apariția unui nivel diferit de tensiune (dintre cele două posibile) pe una sau mai multe intrări, eveniment care produce un nivel de tensiune diferit (sau nu – depinde de funcția logică implementată) pe semnalul de ieșire (rezultatul). Circuitele prezentate deja în capitolul anterior, porțile logice, sunt combinaționale.

Circuitele *secvențiale* sunt circuite care au proprietatea de memorie sau *stare*, proprietate care lipsește pentru circuitele combinaționale. Valoarea vizibilă pe ieșire, reprezentând *starea* circuitului, se schimbă doar în anumite condiții în funcție de semnalele de intrare. Există două feluri de circuite secvențiale: *sincrone*, care își schimbă starea doar în momente bine definite de timp, cu o cadență dată de un semnal special numit *semnal de ceas* (eng. clock), și *asincrone*, care nu au un semnal de ceas dedicat și își schimbă starea doar în anumite condiții, în funcție de semnalele de intrare (exemplu: apariția anumitor combinații de valori pe unul sau mai multe dintre semnalele de intrare). Aproape în mod universal în calculatoare se folosesc circuitele secvențiale sincrone, acestea fiind detaliate în continuare în această carte. Cele asincrone au anumite avantaje, în special de viteză, însă necesită multă atenție la proiectare și implementare, fiind utile pentru aplicații particulare.

În capitolele următoare, pentru proiectarea diverselor circuite, se va folosi în general descrierea din punct de vedere logic a circuitelor: valoarea logică de pe ieșire depinde doar de valorile curente de pe intrări pentru circuite combinaționale, iar la cele secvențiale valoarea de pe intrare se memorează (devenind vizibilă pe ieșire) în funcție de semnalul de ceas. În anumite etape de proiectare se va explica și comportamentul în implementarea fizică, pentru o înțelegere mai bună a funcționării. Implementarea fizică a unui circuit depinde de tehnologia folosită. Semnalul electric care reprezintă nivelurile logice nu se propagă instantaneu printr-un circuit (combinațional sau secvențial), existând anumite întârzieri specifice. Însă, după ce valoarea semnalelor devine stabilă, circuitul respectă specificațiile din punct de vedere logic.



## 5. Circuite combinaționale

Circuitele *combinaționale* sunt circuite care nu au *stare*, ieșirea unui circuit combinațional depinzând doar de valoarea curentă a semnalelor de intrare, conform tabelii de adevăr care descrie funcțional circuitul.

Un circuit combinațional este la rândul lui format din alte circuite combinaționale legate între ele prin *noduri* (conexiuni, fizic fiind cablaje/fire, eng. wires). Cele mai simple circuite combinaționale sunt porțile logice discutate deja în capitolul 3.

Un circuit combinațional nu trebuie să aibă bucle, adică să aibă ieșirea conectată la una sau mai multe intrări (Fig. 5.1). Astfel de bucle sunt interzise, deoarece duc la un comportament imprevizibil al circuitului.

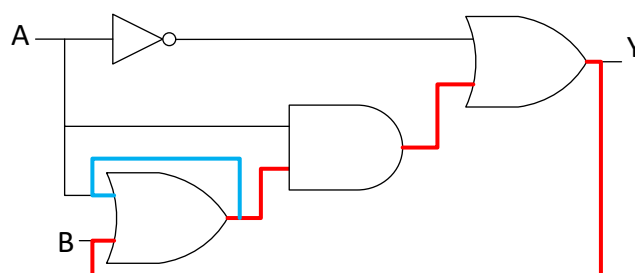


Fig. 5.1. Exemple de bucle într-un circuit (traseele cu roșu, respectiv albastru), interzise în circuitele combinaționale

În continuare se vor prezenta circuite combinaționale de bază, care, pe lângă porțile logice, pot fi utilizate pentru a construi circuite mai complexe (până la nivelul unui procesor). Pentru fiecare circuit se va descrie funcționalitatea, cu tabela de adevăr asociată, și schema bloc cu semnalele de intrare și ieșire. Se vor pune în practică metodele din capitolele anterioare pentru a construi majoritatea circuitelor descrise, pornind de la porți logice.

Se vor descrie și construi în detaliu multiplexorul, demultiplexorul, decodificatorul, sumatorul, iar la finalul capitolului se va construi o unitate fundamentală pentru orice procesor, și anume unitatea aritmetică-logică.

### 5.1. Multiplexorul

*Multiplexorul* (eng. multiplexer) sau, pe scurt, *mux*, este un circuit combinațional cu rol de selecție care are  $N$  intrări, un semnal de selecție  $S$  și o singură ieșire. Denumirea uzuală este de multiplexor  $N:1$  (sau mux  $N:1$ , citit *mux N la 1*). Pe ieșirea mux-ului se propagă valoarea uneia dintre cele  $N$  intrări, intrarea de propagat fiind selectată de semnalul de selecție  $S$ . Semnalul de selecție are rol de index pentru intrări, el având valori între 0 și  $N-1$ . Simbolul pentru multiplexor este un trapez, cu latura mare pe partea intrărilor, ca în Fig. 5.2.

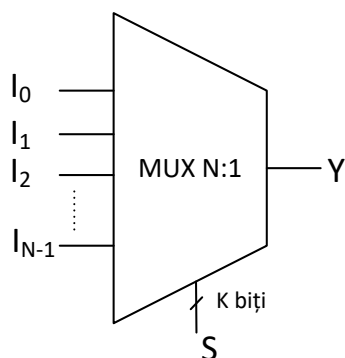


Fig. 5.2. Schema bloc pentru un multiplexor  $N:1$ , cu intrările  $I_0, I_1, I_2, \dots, I_N$ , ieșirea  $Y$  și semnalul de selecție  $S$  reprezentat pe  $k$  biți (*remarcă*: acest lucru este reprezentat pe schemă printr-o tăietură obișnuită pe linia asociată semnalului, lângă tăietură scriindu-se numărul de biți)

În mod uzual se folosesc mux-uri cu un număr de intrări egal cu  $2^k$ , unde  $k$  este numărul de biți pentru semnalul de selecție: mux 2:1 – 1 bit pentru semnalul de selecție, mux 4:1 – 2 biți pentru selecție, mux 8:1 – 3 biți pentru selecție etc. Atât intrările, cât și ieșirea unui multiplexor pot reprezenta semnale pe mai mulți biți. În figura următoare sunt prezentate exemple particulare de mux-uri.

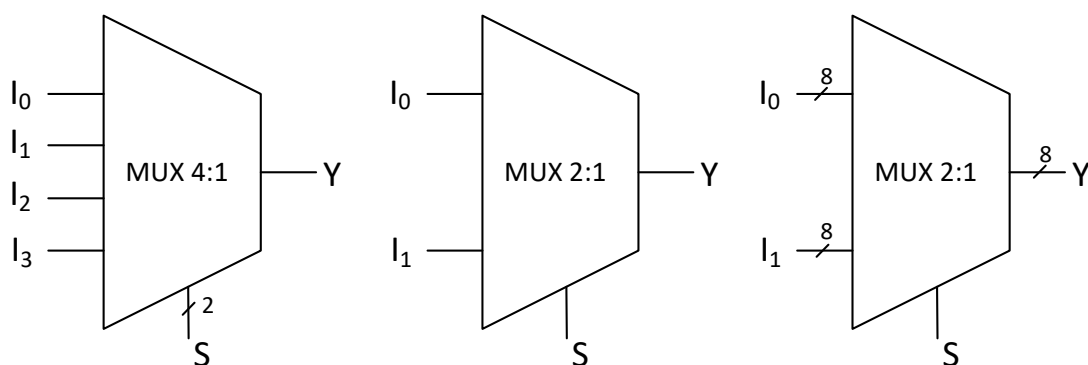


Fig. 5.3. Exemple particulare de multiplexoare, de la stânga la dreapta: mux 4:1, mux 2:1, mux 2:1, având intrările și ieșirea pe 8 biți

Pentru a răspunde la întrebarea *Ce este în interiorul unui multiplexor?*, în continuare se prezintă o implementare directă cu porți logice a unui multiplexor 2:1. Se aplică metoda bazată pe sumă de produse, pornind de la tabela de adevăr.

Tabel 5.1. Tabela de adevăr pentru multiplexorul 2:1

$I_0$	$I_1$	$S$	$Y$	Termen canonic
0	0	0	0	-
0	1	0	0	-
1	0	0	1	$I_0 \cdot \bar{I}_1 \cdot \bar{S}$
1	1	0	1	$I_0 \cdot I_1 \cdot \bar{S}$
0	0	1	0	
0	1	1	1	$\bar{I}_0 \cdot I_1 \cdot S$
1	0	1	0	
1	1	1	1	$I_0 \cdot I_1 \cdot S$

Tabela de adevăr din Tabel 5.1 descrie funcționarea multiplexorului: dacă selecția  $S$  este 0, atunci ieșirea  $Y$  va lua valoarea de pe intrarea  $I_0$ , iar dacă  $S$  este 1, pe  $Y$  trece valoarea lui  $I_1$ . Pentru fiecare valoare posibilă a selecției, tabela conține cele 4 combinații posibile pentru intrări. Se identifică liniile unde  $Y$  este 1 și se scriu termenii canonici pentru acele linii (produs de variabilele de intrare  $I_0$ ,  $I_1$ ,  $S$ , cele care sunt 0 pe linia respectivă fiind negate).

Termenii canonici identificați sunt însumați pentru a obține ecuația multiplexorului:

$$Y = I_0 \cdot \bar{I}_1 \cdot \bar{S} + I_0 \cdot I_1 \cdot \bar{S} + \bar{I}_0 \cdot I_1 \cdot S + I_0 \cdot I_1 \cdot S$$

Folosind asociativitatea, între primii doi, respectiv ultimii doi termeni canonici se pot da factori comuni:

$$Y = I_0 \cdot \bar{S} \cdot (\bar{I}_1 + I_1) + I_1 \cdot S \cdot (\bar{I}_0 + I_0)$$

Suma dintre o variabilă și aceeași variabilă negată este 1 (teorema complementului):

$$Y = I_0 \cdot \bar{S} \cdot 1 + I_1 \cdot S \cdot 1$$

$$\Leftrightarrow$$

$$Y = I_0 \cdot \bar{S} + I_1 \cdot S \quad (5.1)$$

Ecuația se implementează simplu, cu două porți ȘI și o poartă SAU, ca în figura următoare. Se remarcă rolul semnalului de selecție: prin intermediul porților ȘI, el inhibă trecerea unuia dintre semnale și permite trecerea celuilalt spre rezultat.

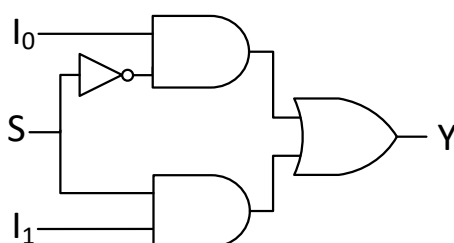


Fig. 5.4. Implementarea cu porți logice pentru un multiplexor 2:1

Pentru multiplexoare mai complexe, se pot construi ierarhii de multiplexoare 2:1.

## 5.2. Demultiplexorul

*Demultiplexorul* (eng. demultiplexer) este un circuit combinațional cu funcție opusă multiplexorului: are o singură intrare  $I$ , care este propagată pe una dintre cele  $N$  ieșiri,  $Y_0 \dots Y_{N-1}$ , în funcție de valoarea semnalului de selecție  $S$ . Este mai puțin utilizat și nu va fi descris în detaliu, dar are aplicații pentru comunicație de date, conversie din serial în paralel etc.

### 5.3. Decodificatorul

*Decodificatorul* (eng. decoder) este un circuit combinațional care are  $N$  intrări și  $M$  ieșiri de un bit, valorile de pe cei  $N$  biți de intrare fiind transformate în valori corespundente pe cei  $M$  biți de ieșire.

Forma standard are  $N$  intrări  $I_0, I_1, \dots, I_{N-1}$  și  $2^N$  ieșiri  $Y_0, Y_1, \dots, Y_{2^N-1}$ . Se activează doar una dintre ieșiri (valoare 1), celelalte fiind inactive (valoare 0). Valoarea întreagă  $k$  prezentă pe cei  $N$  biți de intrare este între 0 și  $2^N-1$  și reprezintă indexul ieșirii care va fi activată:  $Y_k$ . Simbolul folosit pentru decodificator este un dreptunghi, ca în figura următoare.

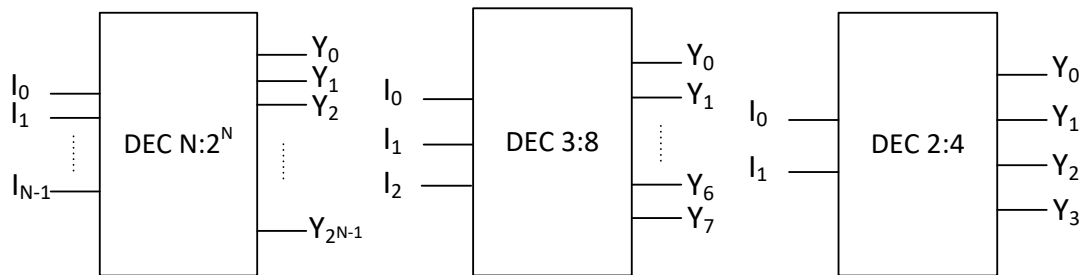


Fig. 5.5. Schema bloc pentru decodificator (de la stânga la dreapta): forma generală  $N:2^N$ , decodificator 3:8, decodificator 2:4

Tabela de adevăr pentru decodificatorul binar 2:4 este prezentată în continuare, valoarea întreagă a indexului fiind dată de combinația de intrări.

$I_0$	$I_1$	Valoare index	$Y_0$	$Y_1$	$Y_2$	$Y_3$
0	0	0	1	0	0	0
0	1	1	0	1	0	0
1	0	2	0	0	1	0
1	1	3	0	0	0	1

Următorul pas este de identificare a termenilor canonici asociați cu liniile unde rezultatul este 1. De data aceasta, metoda cu sumă de produse este replicată pentru fiecare dintre cei 4 biți de ieșire.

$I_0$	$I_1$	$Y_0$	T. can. $Y_0$	$Y_1$	T. can. $Y_1$	$Y_2$	T. can. $Y_2$	$Y_3$	T. can. $Y_3$
0	0	1	$\bar{I}_0 \cdot \bar{I}_1$	0	-	0	-	0	-
0	1	0	-	1	$\bar{I}_0 \cdot I_1$	0	-	0	-
1	0	0	-	0	-	1	$I_0 \cdot \bar{I}_1$	0	-
1	1	0	-	0	-	0	-	1	$I_0 \cdot I_1$

Pentru fiecare ieșire se face suma termenilor canonici rezultați (în mod particular există un singur termen asociat fiecărei ieșiri).

$$\begin{aligned}
 Y_0 &= \bar{I}_0 \cdot \bar{I}_1 \\
 Y_1 &= \bar{I}_0 \cdot I_1 \\
 Y_2 &= I_0 \cdot \bar{I}_1 \\
 Y_3 &= I_0 \cdot I_1
 \end{aligned} \tag{5.2}$$

Implementarea cu porți logice este prezentată în figura următoare. Pentru fiecare ieșire  $Y$  este necesară câte o poartă ȘI. La fiecare intrare se leagă câte un inversor, pentru a avea disponibilă și valoarea negată a intrării.

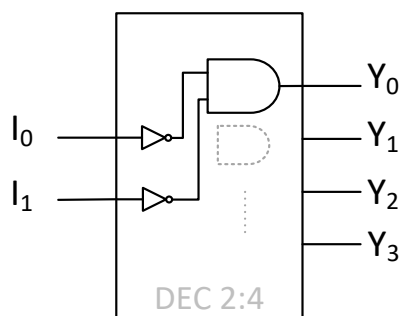


Fig. 5.6. Detalii de implementare internă pentru decodificatorul 2:4. Din motive de spațiu, pe schemă este prezentată doar implementarea pentru prima ieșire,  $Y_0$

#### 5.4. Sumatorul

Sumatorul este un circuit care adună numere binare. Se începe cu construcția unui sumator pe 1 bit (eng. *1-bit adder*). Remarcă: în acest subcapitol, de obicei, prin sumă ne referim la operația aritmetică de adunare, nu la conceptul de sumă (=SAU logic) din algebra booleană. Se îmbină practic cunoștințele de aritmetică binară cu cele din algebra booleană prezentate în capitolele anterioare, cu scopul de a construi circuite sumatoare din porți logice.

Pentru un circuit sumator pe 1 bit, intrările sunt cei doi biți care se adună,  $A$  și  $B$ , iar ieșirile sunt bitul de sumă  $S$  și bitul de transport de ieșire  $C_{out}$  (eng. carry out). Acest tip de sumator se numește semi-sumator (eng. half adder). Tabela de adevăr este prezentată în continuare și este construită pe baza regulilor de adunare în sistemul binar.

Tabel 5.2. Tabela de adevăr pentru semi-sumator

$A$	$B$	$S$	$C_{out}$
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Pentru a deduce ecuațiile booleene pentru cele două semnale de ieșire se poate aplica metoda bazată pe sume de produse. Ca o soluție alternativă, pornind de la experiența acumulată în capitolele anterioare, putem observa că:

- Semnalul  $S$  coincide cu ieșirea unei porți SAU exclusiv, XOR, cu intrările  $A$  și  $B$

*Observație:* folosind reprezentarea ca sumă de produse, cu termeni canonici extrași din tabela de adevăr, rezultă  $S = A \cdot \bar{B} + \bar{A} \cdot B = A \oplus B$ , aceasta fiind o identitate cunoscută!

- Semnalul  $C_{out}$  se poate genera direct cu o poartă ȘI, cu intrările  $A$  și  $B$ .

Circuitul din porți logice care rezolvă adunarea a doi biți este prezentat în figura următoare.

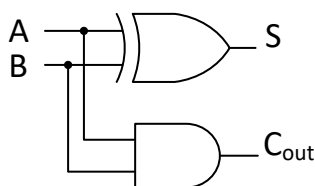


Fig. 5.7. Semi-sumator construit din porți logice, A și B sunt semnalele de intrare, S este suma și  $C_{out}$  transportul de ieșire, toate semnalele fiind pe 1 bit \*

\* Convenție pentru scheme: atunci când două trasee se intersectează, se consideră că nu există contact fizic între ele. Când există o bifurcație (un semnal pleacă mai departe pe două trasee), atunci intersecția este marcată cu un cerc (conector).

Pentru a construi un sumator pe N biți se pot folosi N sumatoare pe 1 bit (câte unul pentru fiecare pereche de biți care se adună).

Semi-sumatorul construit anterior rezolvă corect adunarea unor biți singuri. În cazul adunării numerelor pe mai mulți biți, acesta poate fi utilizat doar pentru perechea de biți de pe poziția cea mai puțin semnificativă (cea mai din dreapta). Ținând cont de aritmetica adunării pe mai mulți biți, pentru restul pozițiilor de biți trebuie adunat și transportul de la perechea anterioară de biți. Acest transport poartă numele de transport de intrare  $C_{in}$  (eng. carry in), iar un sumator care permite și acest tip de transport se numește *sumator complet* (eng. full adder).

Tabela de adevăr pentru sumatorul complet se obține pornind de la cea a semi-sumatorului, prin adăugarea noului semnal de intrare  $C_{in}$ , adunarea pentru fiecare linie efectuându-se între cele 3 variabile de intrare:

Tabel 5.3. Tabela de adevăr pentru sumatorul complet

A	B	$S_{half}$	$C_{in}$	S	$C_{out}$
0	0	0	0	0	0
0	1	1	0	1	0
1	0	1	0	1	0
1	1	0	0	0	1
0	0	0	1	1	0
0	1	1	1	0	1
1	0	1	1	0	1
1	1	0	1	1	1

Probabil nu este clar de ce în tabelă apare și termenul  $S_{half}$ , acesta fiind suma fără transport de intrare (generată cu poarta XOR, vezi semi-sumatorul). Dacă se analizează coloanele S,  $S_{half}$  și  $C_{in}$ , se constată că S este 1 doar în situațiile când  $S_{half}$  și  $C_{in}$  sunt diferite, adică definiția directă a unei porți XOR! Prin urmare, circuitul care permite calculul lui S este format din 2 porți XOR înlanțuite, ca în figura următoare:

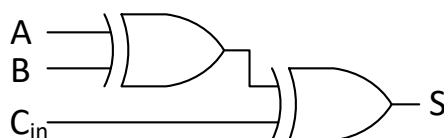


Fig. 5.8. Generarea sumei pentru un sumator complet pe 1 bit



Dacă se merge pe abordarea clasică cu termeni canonici, după mai mulți pași se poate ajunge la același rezultat, după cum se va arăta în continuare. Se extrag și se însumează (SAU logic!) termenii canonici pentru liniile din tabelă unde semnalul de ieșire  $S$  este 1 și se obține:

$$S = \bar{A} \cdot B \cdot \overline{C_{in}} + A \cdot \bar{B} \cdot \overline{C_{in}} + \bar{A} \cdot \bar{B} \cdot C_{in} + A \cdot B \cdot C_{in}$$

$$\Leftrightarrow$$

$$S = (\bar{A} \cdot B + A \cdot \bar{B}) \cdot \overline{C_{in}} + (\bar{A} \cdot \bar{B} + A \cdot B) \cdot C_{in}$$

$$\Leftrightarrow$$

$$S = (A \oplus B) \cdot \overline{C_{in}} + (\bar{A} \cdot \bar{B} + A \cdot B) \cdot C_{in} \quad (5.3)$$

Pentru a forma a doua poartă XOR ar trebui ca termenul  $\bar{A} \cdot \bar{B} + A \cdot B$  să fie egal cu  $\overline{A \oplus B}$ . Pentru a demonstra echivalența se pot calcula tabelele de adevăr și compara rezultatele expresiilor sau folosi teoremele booleene. Mai jos se prezintă o demonstrație cu a doua abordare, cu scopul de a exersa teoremele booleene. Se începe cu o dublă negare (involuția) (5.4), apoi se aplică teorema lui De Morgan pentru suma negată (termenii sunt evidențiați prin paranteze) (5.5), apoi din nou teorema lui De Morgan pentru fiecare dintre cele două produse negate obținute (5.6), se desfac parantezele, se elimină produsele redundante și se obține poarta XOR căutată (5.7)-(5.10).

$$\bar{A} \cdot \bar{B} + A \cdot B = \overline{\overline{\bar{A} \cdot \bar{B} + A \cdot B}} \quad (5.4)$$

$$\Leftrightarrow$$

$$\bar{A} \cdot \bar{B} + A \cdot B = \overline{(\bar{A} \cdot \bar{B}) \cdot (A \cdot B)} \quad (5.5)$$

$$\Leftrightarrow$$

$$\bar{A} \cdot \bar{B} + A \cdot B = \overline{(A + B) \cdot (\bar{A} + \bar{B})} \quad (5.6)$$

$$\Leftrightarrow$$

$$\bar{A} \cdot \bar{B} + A \cdot B = \overline{A \cdot (\bar{A} + \bar{B}) + B \cdot (\bar{A} + \bar{B})} \quad (5.7)$$

$$\Leftrightarrow$$

$$\bar{A} \cdot \bar{B} + A \cdot B = \overline{A \cdot \bar{A} + A \cdot \bar{B} + B \cdot \bar{A} + B \cdot \bar{B}} \quad (5.8)$$

$$\Leftrightarrow$$

$$\bar{A} \cdot \bar{B} + A \cdot B = \overline{A \cdot \bar{B} + \bar{A} \cdot B} \quad (5.9)$$

$$\Leftrightarrow$$

$$\bar{A} \cdot \bar{B} + A \cdot B = \overline{A \oplus B} \quad (5.10)$$

Se înlocuiește termenul din (5.10) în relația (5.3) și se obține:

$$S = (A \oplus B) \cdot \overline{C_{in}} + (\overline{A \oplus B}) \cdot C_{in} \quad (5.11)$$

$$\Leftrightarrow$$

$$S = (A \oplus B) \oplus C_{in} \quad (5.12)$$

Pentru determinarea ecuației transportului de ieșire  $C_{out}$ , se identifică termenii canonici din tabela de adevăr (Tabel 5.3). Suma termenilor canonici este:

$$C_{out} = A \cdot B \cdot \overline{C_{in}} + \bar{A} \cdot B \cdot C_{in} + A \cdot \bar{B} \cdot C_{in} + A \cdot B \cdot C_{in} \quad (5.13)$$

grupăm convenabil termenul 1 cu 4 și 2 cu 3

$$C_{out} = A \cdot B \cdot (\overline{C_{in}} + C_{in}) + (\bar{A} \cdot B + A \cdot \bar{B}) \cdot C_{in} \quad (5.14)$$

aplicăm teorema complementului și recunoaștem o poartă XOR

$$C_{out} = A \cdot B \cdot 1 + (A \oplus B) \cdot C_{in} \quad (5.15)$$

$$C_{out} = A \cdot B + (A \oplus B) \cdot C_{in} \quad (5.16)$$

În acest moment, ambele semnale de ieșire ale sumatorului sunt exprimate sub formă de ecuații booleene și se poate construi circuitul complet. Se extinde circuitul prezentat în Fig. 5.8 cu implementarea ecuației booleene (5.16), pentru calculul transportului de ieșire  $C_{out}$ . Se obține astfel sumatorul complet implementat cu porți logice. Sunt necesare în plus pe schemă: două porți ȘI, o poartă SAU și conexiunile corespunzătoare.

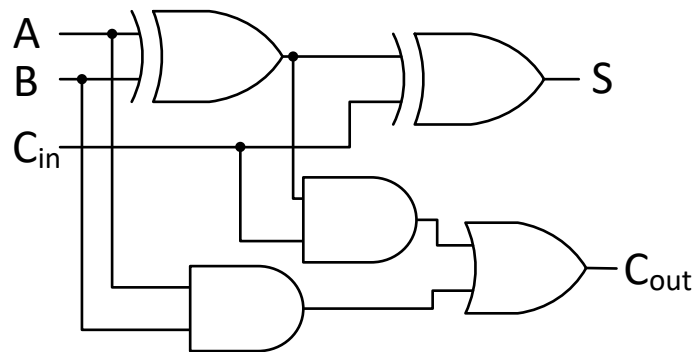


Fig. 5.9. Circuitul sumator complet pe 1 bit

Diagrama bloc a sumatorului complet pe 1 bit este prezentată în figura următoare, cu o dispunere convenabilă a semnalelor pentru construirea sumatorului pe mai mulți biți.

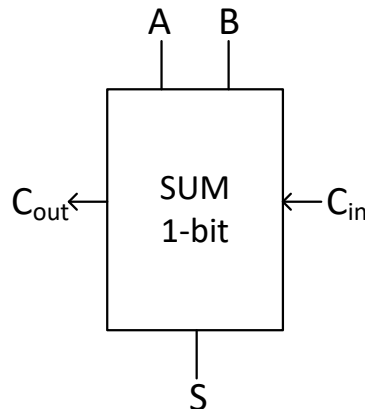


Fig. 5.10. Diagrama bloc pentru sumatorul complet pe 1 bit. În mod particular pe această schemă s-a evidențiat cu săgeți sensul semnalelor de transport (de intrare sau ieșire)

În cazul adunării numerelor binare pe  $n$  biți soluția este simplă: se înlanțuie  $n$  sumatoare complete pe 1 bit. Adunarea pentru fiecare pereche de biți se face cu câte un sumator complet pe 1 bit. Transportul de la perechea curentă la perechea următoare este transmis prin legarea ieșirii  $C_{out}$  de la sumatorul curent la intrarea  $C_{in}$  a sumatorului următor.

Dacă numerele de intrare pe  $n$  biți sunt  $A_{n-1}...A_1A_0$  și  $B_{n-1}...B_1B_0$ , iar suma  $S_{n-1}...S_1S_0$ , atunci diagrama sumatorului pe  $n$  biți arată ca în Fig. 5.11. De remarcă că pe transportul de intrare al sumatorului pentru bitul 0 trebuie pusă valoarea 0 pentru a funcționa corect (pe bitul cel mai puțin semnificativ nu există transport de intrare).

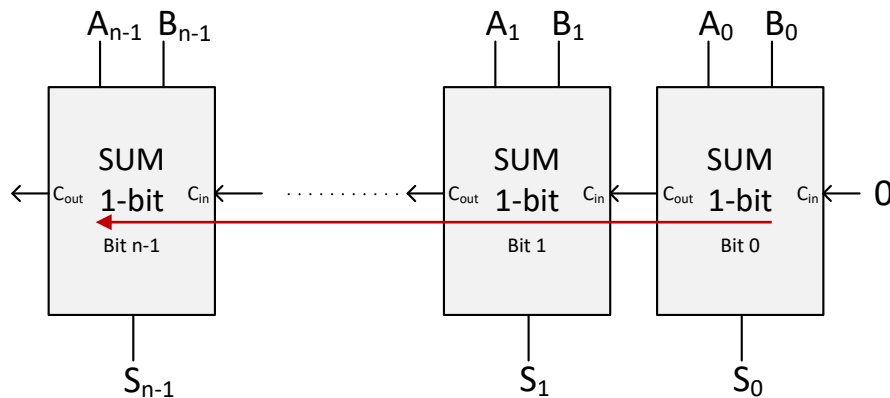


Fig. 5.11. Sumator pe  $n$  biți obținut prin înlanțuirea a  $n$  sumatoare complete pe 1 bit. Calea critică, pe unde durează cel mai mult propagarea fizică a semnalului, este evidențiată cu roșu

În implementarea fizică, sumatorul pe  $n$  biți este un circuit combinațional care produce rezultatul corect după propagarea completă a semnalelor de intrare prin circuit. La o analiză mai atentă, cel mai târziu se vor stabili semnalele de ieșire ale sumatorului  $n-1$  care depind de transportul de intrare venit de la sumatorul anterior, care la rândul lui depinde de sumatoarele anterioare etc. Acest traseu combinațional, pe care durează cel mai mult propagarea fizică a semnalelor și stabilizarea rezultatelor, poartă numele de *cale critică*. Se va reveni la acest concept în capitolele următoare.

Cu scopul de a merge tot mai sus cu nivelul de abstractizare, în Fig. 5.12 sunt prezentate diagramele bloc uzuale pentru un sumator pe  $n$  biți.

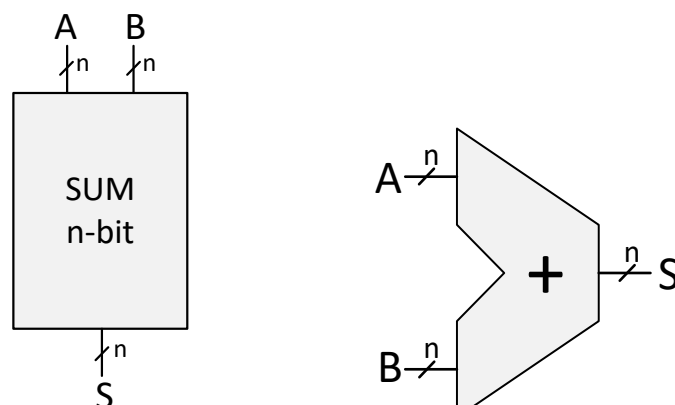


Fig. 5.12. Diagramele bloc folosite în mod uzual pentru sumatorul pe  $n$  biți. Transportul de ieșire  $C_{out}$  de la bitul cel mai semnificativ se pune pe schemă doar dacă este folosit în mod explicit în exterior (de exemplu: pentru detecția depășirii)

### 5.5. Unitatea Aritmetică-Logică

Unitatea aritmetică-logică, acronim UAL (eng. Arithmetic-Logic Unit – ALU), este o componentă de bază a unui procesor, fiind acea parte din procesor care execută operațiile aritmetice și logice.

În acest subcapitol se va proiecta o UAL care poate executa operații aritmetice de adunare, scădere și operații logice de ȘI/SAU pe biți. Unitatea are 2 operanzi de intrare pe  $n$  biți și produce un rezultat de ieșire tot pe  $n$  biți. Unitatea se va folosi ca bloc constructiv al procesorului de tip MIPS prezentat în capitolele următoare (este similară cu descrierea din literatură, anexa B.5 din (Patterson & Hennessy, 2013), cu anumite modificări).

În mod similar cu abordarea pentru sumator, se va construi mai întâi o UAL care să funcționeze pe operanzi de 1 bit. Unitatea UAL pe 1 bit se va proiecta inițial pentru operațiile de adunare, ȘI și SAU, după care, exploatând avantajele complementului față de 2, se va extinde pentru scădere.

Pentru cele 3 operații, circuitele necesare au fost deja construite: sumator complet pe 1 bit, poarta ȘI și poarta SAU. Problema este de a selecta ce se trimite ca rezultat la ieșirea UAL. Sună cunoscut? Mai multe intrări (rezultatele de la fiecare circuit individual) și o singură ieșire a UAL... Se va folosi un multiplexor, unde semnalul de selecție alege operația al cărei rezultat este dorit la ieșirea UAL. Schema UAL pe 1 bit este prezentată în figura următoare.

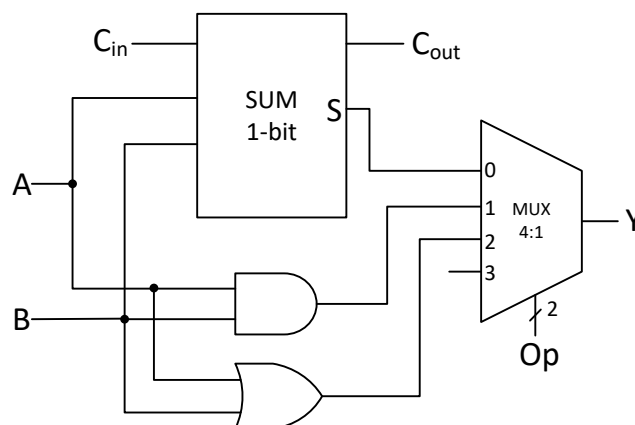


Fig. 5.13. Diagrama pentru UAL 1 bit, cu operații de adunare, ȘI logic și SAU Logic. Ieșirea este Y, iar semnalul de selecție pentru UAL este  $Op$  (Operație)

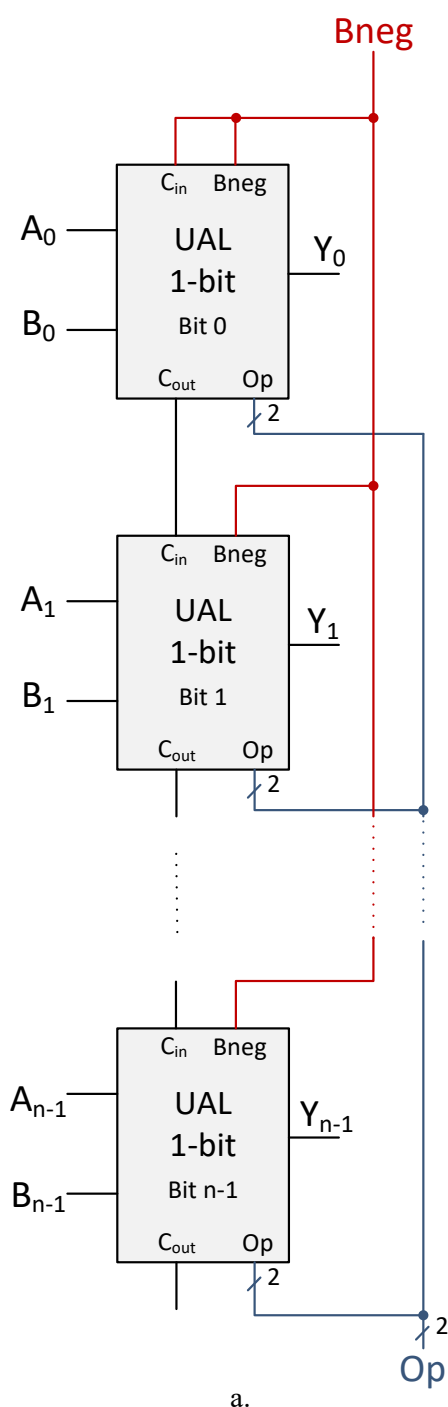
Semnalul de control  $Op$  va selecta ce rezultat să apară pe ieșirea UAL: 0 pentru adunare, 1 pentru ȘI, 2 pentru SAU, 3 rezervat pentru adăugarea unei noi operații. Ca o paranteză, dacă cititorul are cunoștințe de programare, ideea că UAL execută una dintre operații poate induce o confuzie. În implementarea fizică, hardware, toate cele 3 operații sunt executate, în sensul că semnalul se propagă combinațional prin circuitele individuale, producând cele 3 rezultate. Doar unul dintre ele apare la ieșirea UAL, în funcție de operația dorită.

Similar cu abordarea de la sumator, pentru a obține o UAL pe  $n$  biți ar trebui înlănțuite  $n$  UAL pe 1 bit, notate ca  $UAL_{n-1}, \dots, UAL_1, UAL_0$ , prin legarea corespunzătoare a semnalelor de transport.

Pentru operația de scădere, se profită de avantajele reprezentării în complement față de 2. O scădere  $A - B$  este rescrisă ca  $A + (-B)$  și se ține cont de proprietatea că negativul



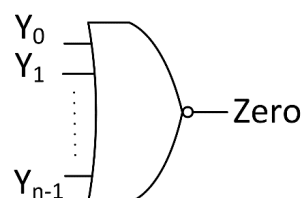
Comportamentul UAL este controlat prin intermediul celor 3 biți de control din semnalul *AluCtrl*. Valorile semnalului de control *AluCtrl* pentru cele 4 operații ale UAL sunt prezentate în Tabel 5.4.



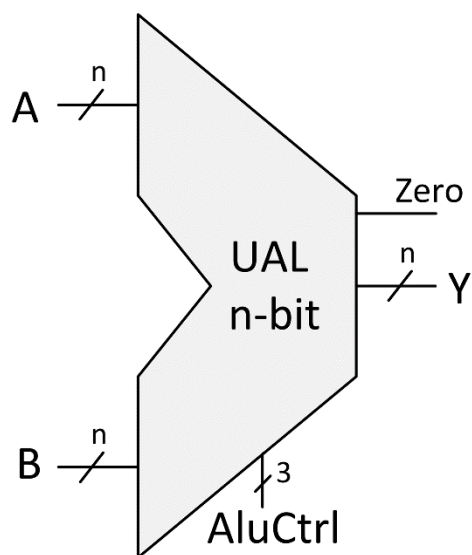
a.

Tabel 5.4

Operație UAL	AluCtrl <sub>2...0</sub>	Observație
Adunare	0 0 0	Bneg=0, Op=0
Scădere	1 0 0	Bneg=1, Op=0
ȘI	0 0 1	Bneg=0, Op=1
SAU	0 1 0	Bneg=0, Op=2



b.



c.

Fig. 5.15. a. UAL pe  $n$  biți asamblată din  $n$  UAL pe 1 bit, cu semnalele de control **Op** și **Bneg**, b. Circuitul de detecție a rezultatului 0 pe ieșirea UAL, c. Diagrama bloc include circuitele din subfigurile a și b, semnalul *AluCtrl* (eng. ALU Control) fiind format din **Bneg** și cei 2 biți ai semnalului **Op**

## 6. Circuite secvențiale

În acest capitol se vor prezenta cele mai uzuale tipuri de *circuite secvențiale*, folosite în mod universal în calculatoare, și anume cele *sincrone*. În general, în restul cărții, prin circuit secvențial se va înțelege implicit un circuit secvențial sincron.

Circuitele secvențiale sunt circuite care au memorie, sau stare, care nu se schimbă prin simpla modificare a intrărilor. Starea circuitului este vizibilă pe semnalul de ieșire (în paginile următoare aceste două noțiuni se vor considera echivalente). În cazul celor sincrone, starea se modifică doar la momente de timp indicate de un semnal special de sincronizare, denumit semnal de ceas (eng. clock signal).

Urmează câteva noțiuni generale legate de semnalul de ceas, după care se vor prezenta cele mai utilizate circuite secvențiale (cu accent pe cele necesare mai departe pentru proiectarea procesorului). În ultima parte se discută cum se pot combina circuitele combinatoriale și secvențiale de bază în circuite mai complexe, cu aspecte legate de funcționarea fizică.

### 6.1. Semnalul de ceas

Semnalul de ceas este un semnal oscilator periodic, care este generat de un circuit special (eng. clock generator). Oscilația are loc între cele două niveluri de tensiune, asociate cu nivelurile logice 0 și 1. Forma cea mai uzuală este cea de undă dreptunghiulară (eng. square wave), care arată în mod ideal ca în figura următoare.

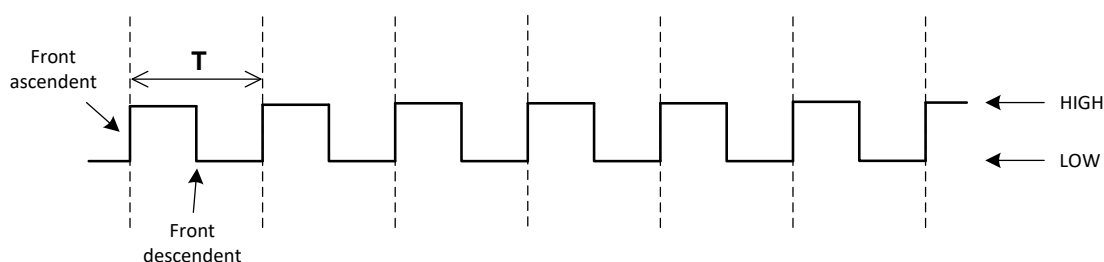


Fig. 6.1. Semnalul de ceas, forma de undă dreptunghiulară

Proprietățile importante ale semnalului de ceas sunt cele două fronturi unde are loc tranziția între niveluri, *ascendent* sau *crescător* (eng. rising edge) și *descendent* sau *descrescător* (eng. falling edge), și perioada  $T$ . Frontul crescător apare la tranziția din 0 (LOW) în 1 (HIGH), iar frontul descrescător invers, la tranziția din 1 în 0. Circuitele secvențiale devin active pe aceste fronturi ale semnalului de ceas (execută transferul intrării spre ieșiri, cu eventuale transformări). Forma frecventă de utilizare este cu sincronizare pe frontul crescător, așa cum e cazul pentru circuitele secvențiale din această carte.

Perioada  $T$  arată durata unui *ciclu* complet *de ceas*, adică durata dintre două fronturi crescătoare consecutive. Cu cât este mai mică perioada, cu atât circuitul funcționează mai repede, producând mai multe transferuri în unitatea de timp. O noțiune des întâlnită pentru sistemele de calcul (calculatoare, telefoane etc.) este frecvența procesorului, o proprietate măsurată în gigahertzi (simbol GHz, unde 1 Hertz = 1 ciclu / secundă). Frecvența  $f$  arată câte cicluri pe secundă are semnalul de ceas, fiind inversa perioadei  $T$ :

$$f = \frac{1}{T}$$

Un semnal de ceas cu frecvența de 1 GHz are 1 miliard de cicluri pe secundă, iar perioada este de 1 nanosecundă:

$$T = \frac{1}{10^9} s = 1 \text{ ns}$$

Mergând mai departe cu exemplele, pentru un semnal de ceas cu frecvența de 2 GHz sunt 2 miliarde de cicluri de ceas pe secundă, iar perioada este de 500 picosecunde, iar pentru 4 GHz se obține o perioadă de 250 picosecunde.

Din punct de vedere ideal, transferul are loc pe frontul crescător (sau descrescător). Aici se pot ridica mai multe întrebări, mai ales dacă se analizează și comportamentul circuitului fizic. În secțiunile care urmează se discută acest subiect.

## 6.2. Circuite secvențiale sincrone

Deși sunt multe variante de circuite secvențiale sincrone de bază, în acest subcapitol se va prezenta doar bistabilul D sincron, varianta activă pe front crescător de ceas (eng. D flip-flop, positive edge-triggered). Acesta este folosit frecvent în circuite de tip procesor. Înțelegerea modului în care funcționează și cum poate fi utilizat este suficientă pentru proiectarea procesorului MIPS.

Pornind de la bistabilul D sincron, se vor prezenta registrul pe  $n$  biți și circuitul de tip numărător.

### 6.2.1. Bistabilul D sincron, activ pe front crescător de ceas

Simbolul pentru bistabilul D sincron (pentru economie de cuvinte se va subînțelege „activ pe front crescător de ceas”) este un dreptunghi cu semnalele aferente, ca în figura următoare.

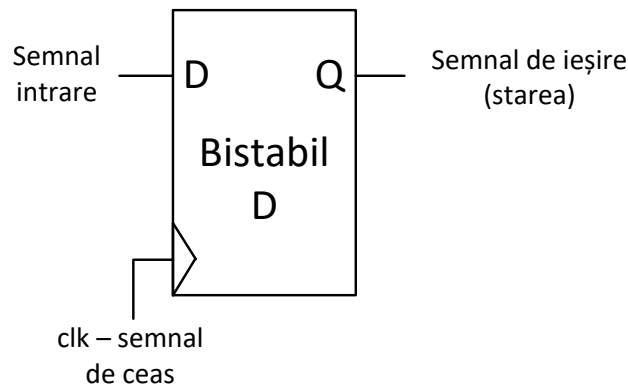


Fig. 6.2. Diagrama bloc pentru bistabilul D sincron (pe front crescător): D semnalul de intrare, Q semnalul de ieșire, pe care este vizibilă starea circuitului, iar clk este semnalul de ceas marcat pe diagramă cu forma triunghiulară (stânga jos)

Pentru a descrie funcționarea, se folosesc tabele de adevăr sau diagrame de timp. Tabelele de adevăr sunt similare cu cele folosite anterior la circuite combinaționale, dar se ține cont de frontul crescător de ceas (desenat simbolic ca  $\uparrow$ ). În tabela următoare (Tabel 6.1) se prezintă valorile semnalului Q înainte ( $Q_{init}$ ) și după (Q) frontul de ceas. Se marchează cu „x” valorile care pot fi 0 sau 1 (nu contează ce valoare are semnalul respectiv pentru a stabili valoarea semnalului de ieșire).



Tabel 6.1. Tabela de adevăr pentru bistabilul D sincron („x” = nu contează)

D	clk	Q <sub>init</sub>	Q
x	0	0	0
x	1	1	1
0	↑	x	0
1	↑	x	1

Singurele situații când valoarea de pe intrare se propagă pe ieșire (se modifică starea la valoarea de pe intrare) sunt când apare un front crescător de ceas. Nu contează ce valoare are intrarea D atunci când ceasul este pe nivelul 0 sau 1, deoarece starea bistabilului nu se modifică.

Pentru a înțelege circuite mai complexe, **regula de propagare** este următoarea: când apare un front crescător de ceas, valoarea existentă pe semnalul de intrare *anterior* apariției frontului de ceas se propagă pe ieșirea bistabilului. Aplicând această regulă (o vom denumi în continuare ca model ideal/logic de propagare) se poate trasa funcționarea unor circuite complexe care conțin circuite secvențiale și combinaționale cu diferite interconexiuni.

Diagrama de timp arată în mod grafic tranziția semnalelor între cele 2 niveluri logice (fizic voltaje), de-a lungul mai multor perioade de ceas succesive.

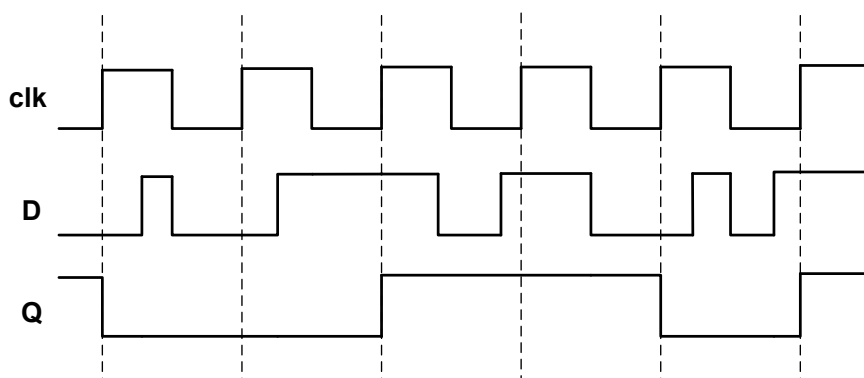


Fig. 6.3. Diagrama de timp pentru bistabilul D sincron. În acest exemplu, se presupune că starea (Q) avea valoarea 1 înainte de primul front crescător de pe diagramă

Bistabilul D sincron are practic rol de memorare, deoarece valoarea preluată de pe intrare, pe frontul crescător de ceas, este menținută pe ieșire (stare) pe întreaga perioadă de ceas.

În aplicații practice, uneori este necesară controlarea momentului de scriere (să nu se facă la fiecare front crescător de ceas) sau este necesară resetarea valorii bistabilului (starea/ieșirea) la valoarea 0. În figura următoare, Fig. 6.4, sunt prezentate două variații ale bistabilului D sincron, prima cu semnal de activare a scrierii (eng. enable, write enable etc.), iar a doua cu activarea scrierii și semnal de resetare (eng. reset).

Activarea scrierii are rolul de a valida scrierea de pe următorul front crescător, iar resetarea forțează valoarea bistabilului la 0, indiferent de valoarea semnalului de intrare. Aceste operații se pot face fie sincron, fie asincron. În exemplele care urmează în carte se vor folosi variantele cu activare și resetare sincrone.

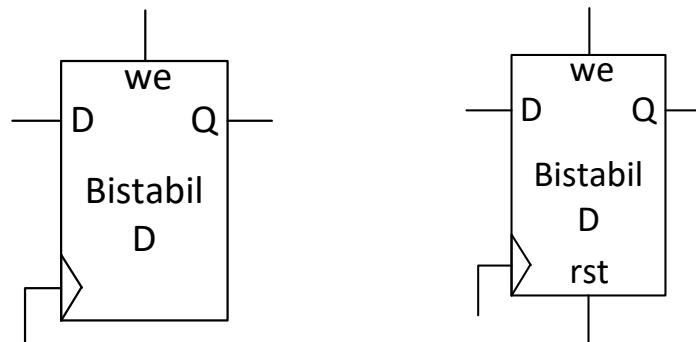


Fig. 6.4. Bistabil D sincron cu semnal de activare a scrierii (we – eng. write enable), respectiv cu activare a scrierii și reset (rst). Denumirea semnalelor nu respectă strict standardul din literatură, dar funcționarea este similară

Diagrama de timp, pentru bistabilul D sincron cu semnal de validare/activare a scrierii, este prezentată în Fig. 6.5.

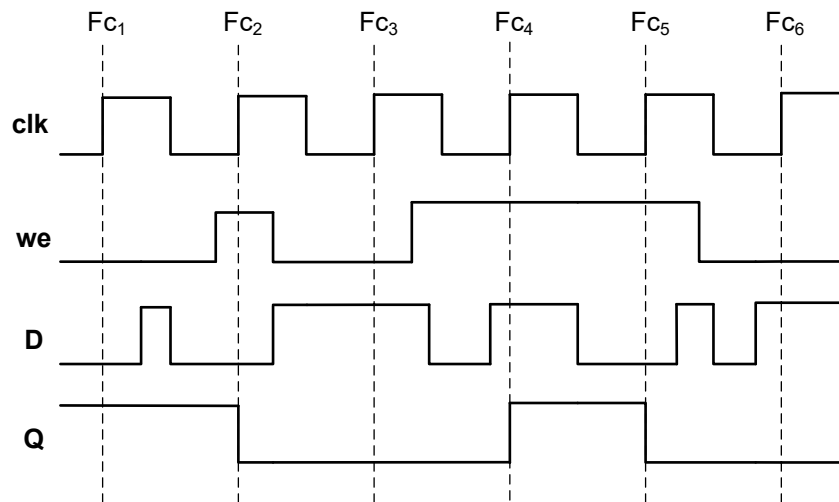


Fig. 6.5. Diagrama de timp pentru bistabilul D sincron (aceeași secvență de intrare ca în Fig. 6.3) cu semnalul de activare (sincron) a scrierii we

În diagrama de timp din figura anterioară se observă efectul de anulare a scrierilor pe front crescător atunci când semnalul de activare e 0. Față de diagrama din Fig. 6.3, pe fronturile crescătoare  $Fc_1$ ,  $Fc_3$  și  $Fc_6$  nu se execută scrieri, valoarea bistabilului rămânând neschimbată.

Implementarea fizică a unui bistabil D sincron se face folosind porți logice. Spre deosebire de circuitele combinaționale unde buclele sunt interzise, bistabilul conține în interior astfel de bucle. Aceste bucle sunt făcute în așa fel încât scrierea să se facă pe frontul de ceas. Mai multe detalii despre acest subiect se pot afla din (Harris & Harris, 2013), capitolul 3.

### 6.2.2. Registrul pe $n$ biți

După cum s-a discutat în primul capitol, numerele sunt reprezentate pe grupuri de biți (8 biți, 16 biți etc.). Un registru pe  $n$  biți este un circuit secvențial, cu rol de memorare, construit din  $n$  bistabili D sincroni care au același semnal de ceas. Fiecare bistabil component are rolul de a memora un bit din cei  $n$  ai numărului.

În funcție de tipul bistabilului D folosit, se poate discuta de registru cu scriere pe fiecare front crescător, sau de registru cu semnal de activare a scrierii, și/sau cu semnal de resetare (activare/resetare sincronă, pentru exemplele discutate mai departe). Câteva exemple în figura următoare.

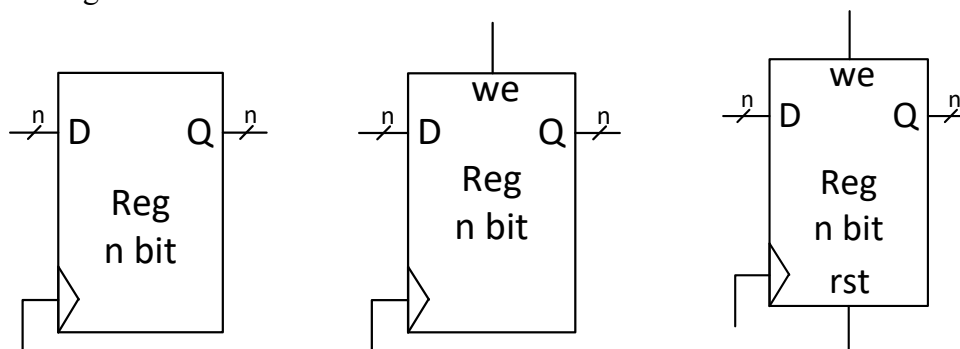


Fig. 6.6. Schema bloc pentru registrul pe  $n$  biți (sincron, front crescător): varianta simplă, cea cu semnal de activare/validare a scrierii  $we$  și cea care are inclusiv semnal de resetare  $rst$

Pentru a descrie funcționarea unui registru se folosesc diagrame de timp, cu specificarea valorii pe fiecare ciclu de ceas (valori binare/zecimale/hexazecimale după conveniență). Se va prezenta un exemplu în subcapitolul 6.4.

### 6.3. Model ideal versus implementarea fizică

În acest subcapitol se discută cum are loc transferul semnalelor aplicând modelul ideal/logic (regula de propagare descrisă anterior) pentru circuite formate din cele enumerate până acum, atât secvențiale, cât și combinaționale. De asemenea, se prezintă câteva detalii legate de ce se întâmplă în implementarea fizică cu transferul semnalelor, pentru a înțelege că, de fapt, modelul ideal de propagare este suficient pentru a trasa funcționarea circuitelor (ideal nu înseamnă incomplet/nepractic).

În implementarea fizică, cele două niveluri logice se regăsesc ca două niveluri diferite de tensiune. Pentru ambele tipuri de circuite (combinaționale și secvențiale), propagarea valorilor de intrare spre ieșire nu este instantanee, ci întârziată. Întârzierea este dată de limitările fizice de propagare/stabilizare a tensiunilor prin circuit, aceste caracteristici de timp fiind cunoscute pentru fiecare circuit fizic.

În figura următoare se prezintă un circuit format din 2 bistabili D înlănțuiți, legați la același semnal de ceas.

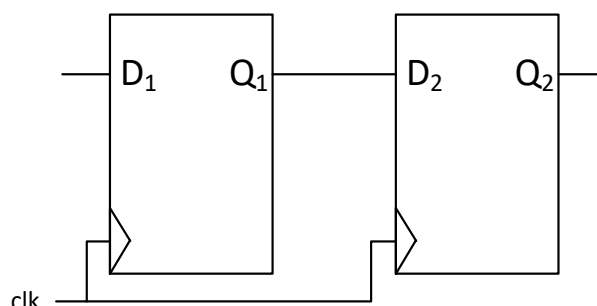


Fig. 6.7. Înlănțuirea circuitelor secvențiale, în cazul a doi bistabili D (sincron, pe front crescător)

Diagrama de timp următoare este construită presupunând că pe primul ciclu de ceas (ciclul  $T_0$  anterior frontului  $F_{c1}$ ) valoarea pe intrarea  $D_1$  este 0, semnalul  $Q_1$  este 1, iar  $Q_2$

este 0. Când ne referim la valoarea pe *ciclul  $n$  de ceas*, ne referim la valoarea stabilă a semnalelor înainte să apară frontul crescător de la finalul ciclului de ceas. De asemenea, nu am specificat valoarea lui  $D_2$ , deoarece  $D_2$  este considerat identic cu  $Q_1$  fiind legate direct.

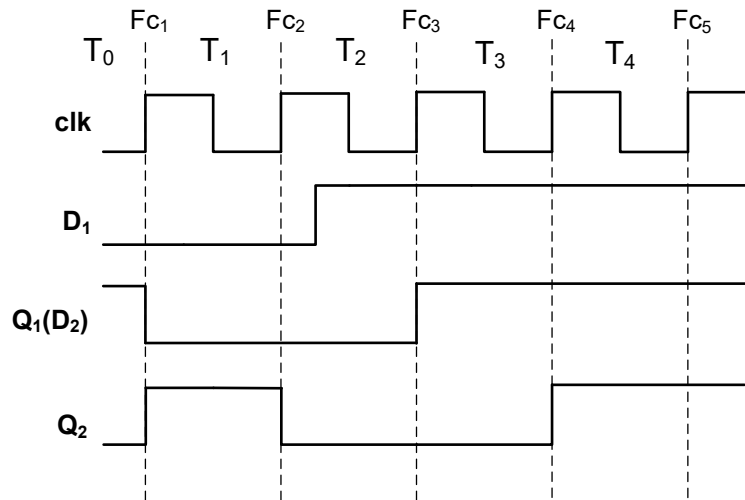


Fig. 6.8. Diagrama de timp pentru circuitul din Fig. 6.7 pornind de la o anumită configurație inițială a semnalelor și o tranziție din 0 în 1 a lui  $D_1$  pe durata ciclului  $T_2$ . Se remarcă întârzierea de un ciclu de ceas (dacă și intrarea a fost generată sincron) dintre forma semnalelor de intrare, respectiv ieșire a bistabililor

Pentru a construi diagrama de sus, la fiecare front crescător de ceas s-a aplicat regula de propagare: valoarea unui semnal de ieșire *după* frontul crescător va fi valoarea semnalului de intrare *anterioară* frontului crescător.

Urmează un alt exemplu, derivat din cel discutat prin intercalarea unui inversor între cei doi bistabili. Ieșirea primului bistabil este negată înainte de a ajunge la intrarea celui de-al doilea.

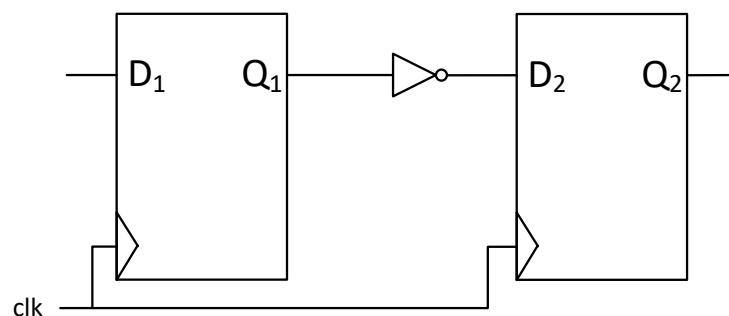


Fig. 6.9. Înlănțuirea circuitelor secvențiale și combinaționale în cazul a doi bistabili D (sincron, pe front crescător) și un circuit inversor

Pentru a construi diagrama de timp se aplică regula de propagare. Valoarea de pe ieșirea inversorului combinațional, echivalentă cu  $D_2$ , va fi valoarea negată a intrării, și anume  $Q_1$  negat. În cazul circuitelor combinaționale, pornind de la definiția lor (ieșirea depinde doar de valoarea curentă a intrărilor), semnalul de ieșire va urma întocmai semnalele de intrare conform funcției de transformare din tabela de adevăr. Se pornește de la aceleași valori inițiale, respectiv intermediare pentru  $D_1$ , ca în exemplul anterior (Fig. 6.8).

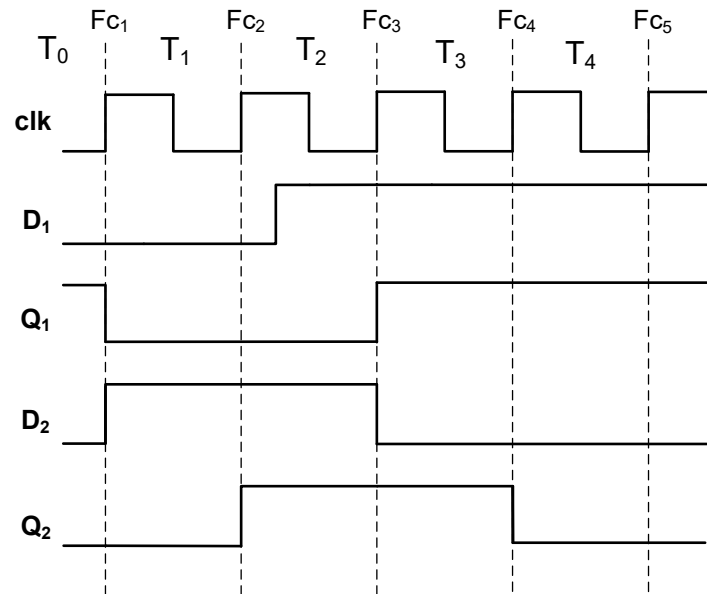


Fig. 6.10. Diagrama de timp pentru circuitul din Fig. 6.9 pornind de la o anumită configurație inițială a semnalelor și o tranziție din 0 în 1 a lui  $D_1$  pe durata ciclului  $T_2$

O potențială problemă/nelămurire: ce se întâmplă dacă semnalul de intrare variază strict anterior și pe durata frontului crescător de ceas (adică în acel moment când valoarea lui este scrisă în starea bistabilului)? Care valoare a semnalului de intrare se propagă?

Răspunsul, pe scurt, la aceste întrebări, este că situația descrisă este impredictibilă și circuitele sunt construite fizic în așa fel încât se evită astfel de situații.

Pentru a lămurii mai bine aceste aspecte este necesară o scurtă discuție legată de funcționarea circuitelor fizice. În modelul teoretic, semnalul de ceas are o tranziție instantanee pe frontul crescător. În realitate, frontul crescător nu are o formă perfect verticală și, implicit, există o durată nenulă a acestui front (durată care este oricum suficient de mică față de perioada de ceas).

Pentru un circuit secvențial, în implementarea fizică (urmăm exemplul bistabilului D sincron, dar noțiunile sunt valabile universal) există niște constrângeri de timp specifice legate de funcționare, altfel circuitul nu ar funcționa corect. Aceste constrângeri, relevante pentru discuția începută, sunt:

1. Semnalul de intrare trebuie să fie stabil (să nu se schimbe) pentru un interval de timp înainte și după frontul crescător de ceas
  - Intervalul anterior este de inițializare, cunoscut ca  $t_{\text{setup}}$  (eng. setup time)
  - Intervalul de după este de menținere, cunoscut ca  $t_{\text{hold}}$  (eng. hold time)
2. Semnalul de ieșire se stabilizează pe noua valoare (preia valoarea de pe intrare) după un anumit interval de timp, cunoscut ca timp de propagare  $t_{\text{prop}}$  (eng. propagation delay).

Constrângerile sunt indicate în diagrama următoare, Fig. 6.11, unde se prezintă scenariu standard pentru un bistabil D. Semnalul de intrare D se modifică din 0 în 1, dar modificarea apare cu cel puțin  $t_{\text{setup}}$  înaintea frontului crescător (mijlocul frontului e ales ca referință), și rămâne stabil pe noua valoare cel puțin pe intervalul de timp  $t_{\text{hold}}$  după frontul

crescător. Semnalul de ieșire Q se stabilizează la noua valoare după intervalul  $t_{prop}$  de la frontul crescător.

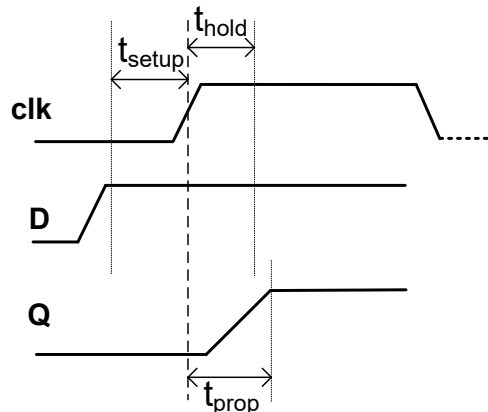


Fig. 6.11. Constrângerile de timp specifice pentru funcționarea normală în cazul transferului sincron, la un bistabil D, în implementarea fizică

Prin semnal stabil se înțelege o valoare egală (cu o anumită marjă de eroare, specifică circuitului) cu unul dintre cele două niveluri de tensiune asociate cu nivelurile logice. Ideea de stabilizare a semnalului este următoarea: când au loc tranziții între cele două niveluri de tensiune apar implicit valori de tensiune intermediare pentru o scurtă perioadă tranzitorie. Aceste valori intermediare nu corespund niciunui nivel logic, dar asta nu influențează funcționarea circuitelor, cât timp se respectă constrângerile de timp.

Revenim la exemplul celor doi bistabili înlanțuiți din Fig. 6.7. În modelul ideal, am considerat ca  $D_2$  este identic cu  $Q_1$ , fiind legate direct. Dacă se consideră modelul real, cu constrângerile de timp (Fig. 6.12), poate să apară o nouă dilemă. Când apare frontul crescător, semnalul  $Q_1$  începe să își schimbe valoarea și ajunge la noua valoare după  $t_{prop}$ . Dar  $Q_1$  este legat direct la intrarea  $D_2$  a următorului bistabil. Valoarea de pe  $D_2$  trebuie să fie valoarea veche a lui  $Q_1$  pentru o perioadă de  $t_{setup}$  înainte și  $t_{hold}$  după frontul de ceas. Perioadele  $t_{hold}$  și  $t_{prop}$  sunt suprapuse (cel puțin parțial) și asta ar viola constrângerile de timp în cazul bistabilului 2: semnalul  $D_2$  să fie stabil inclusiv pe perioada  $t_{hold}$ . În implementarea fizică problema se rezolvă simplu: traseul dintre  $Q_1$  și  $D_2$  este construit în așa fel încât să întârzie suficient propagarea noii valori a lui  $Q_1$  pe  $D_2$ . Întârzierea pe traseu se poate realiza inclusiv prin intercalarea de circuite care doar transmit mai departe semnalul. În Fig. 6.12, se poate observa rolul întârzierii  $t_h$  pe  $D_2$ , pentru fronturile  $F_{c1}$  și  $F_{c3}$ . Astfel, se asigură stabilitatea valorii de intrare ( $D_2$ ) în bistabilul al doilea, pe durata scrierii, atunci când sursa  $Q_1$  își modifică valoarea.

Diagrama de timp pentru modelul real, folosind același scenariu, este construită ținând cont de întârzierile minime necesare care se introduc în implementarea fizică pentru funcționarea identică cu modelul ideal. Pentru simplitate, frontul crescător a fost desenat tot ca în cazul ideal (pe poziția mijlocului frontului real). De data aceasta, semnalele  $Q_1$  și  $D_2$  sunt desenate independent, ele având aceeași formă, dar translatate relativ cu o durată cel puțin egală cu timpul  $t_{hold}$  ( $t_h$  pe diagramă).

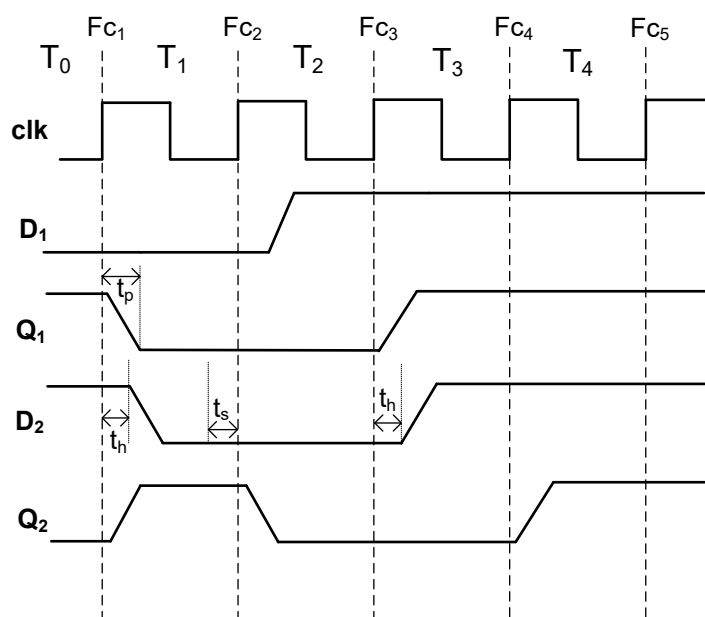


Fig. 6.12. Diagrama de timp cu constrângerile de întârziere impuse de implementarea fizică ( $t_s=t_{\text{setup}}$ ,  $t_h=t_{\text{hold}}$ ,  $t_p=t_{\text{prop}}$ ). Explicații în text

O altă posibilă nelămurire (și ultima discutată) este legată de stabilizarea semnalului de ieșire dintr-un circuit secvențial, care are loc după un interval  $t_{\text{prop}}$  de la frontul de ceas. Pe exemplul din diagrama anterioară, ce se întâmplă (vezi ciclul T1) dacă  $t_p$  asociat cu Fc1 ar fi atât de mare încât s-ar suprapune peste intervalul de inițializare  $t_s$  de la următorul front, Fc2? Răspunsul este că circuitul nu ar funcționa corect deoarece nu ar fi respectate constrângerile de temporizare pentru al doilea bistabil. Soluția este, de fapt, de a folosi un semnal de ceas cu perioada mai mare. Dacă între cei doi bistabili ar fi introdus un circuit combinațional care, la rândul lui, ar aduce o întârziere, atunci durata perioadei de ceas ar trebui să acopere inclusiv această întârziere, în plus față de constrângerile pentru bistabili.

Concluzii rezultate din discuția de până acum, precum și ceea ce este important de reținut pentru capitolele următoare:

1. Circuitele fizice sunt implementate astfel încât să respecte funcționarea conform modelului ideal, logic, prin diferite tehnici: întârzieri suplimentare pe trasee pentru a respecta constrângerile de timp ale circuitelor secvențiale, folosirea unei perioade de ceas adecvate pentru propagarea și stabilizarea semnalelor (tot pentru respectarea constrângerilor) etc.
2. (*De reținut*) Pentru descrierea circuitelor mai complexe este suficientă aplicarea regulilor de propagare a valorilor din punct de vedere logic (modelul ideal discutat), pentru fiecare circuit component:
  - I. *Circuite combinaționale*: semnalul de ieșire se modifică la orice modificare a semnalelor de intrare, conform tabelului de adevăr
  - II. *Circuite secvențiale* (ex. bistabilul D sincron): semnalul de ieșire (=starea) se modifică la fiecare front de ceas (dacă e cazul, se testează și semnalul de validare a scrierii), noua valoare (de după front) fiind valoarea prezentă pe semnalul de intrare anterior sosirii frontului de ceas. Între fronturile de ceas consecutive, valoarea stării se consideră stabilă.

#### 6.4. Un exemplu de circuit combinat

În acest subcapitol vom construi un circuit simplu care numără perioadele de ceas, prin combinarea a două circuite, unul secvențial și unul combinațional. Un astfel de circuit se numește numărător. Implementarea prezentată în continuare nu este standard, dar oferă posibilitatea de a pune în practică noțiunile discutate anterior și de a continua cu abordarea graduală prin care se crește complexitatea circuitelor prezentate.

Intuitiv, deoarece trebuie numărate perioadele de ceas, avem nevoie de un circuit care să stocheze valoarea curentă a contorului. Vom folosi un registru pe  $n$  biți ( $n$  se alege convenabil în funcție de valoarea maximă până unde dorim să numărăm), cu reset (pentru a putea inițializa cu 0 registrul). Pentru incrementarea valorii curente se poate folosi un circuit de tip sumator. Ieșirea registrului se va lega la una dintre intrările sumatorului, iar pe cealaltă intrare se va pune valoarea 1. Pentru a finaliza circuitul de numărare, ieșirea din sumator se leagă la intrarea registrului. Rezultă circuitul din figura următoare.

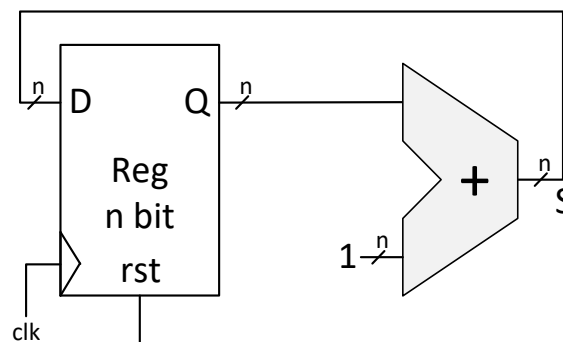


Fig. 6.13. Diagrama circuitului de numărare

Pentru a înțelege funcționarea acestui circuit, e suficient să considerăm concluziile din subcapitolul anterior. Sumatorul, fiind combinațional, va avea pe ieșire valoarea  $Q+1$ , de fiecare dată când se schimbă  $Q$ , valoarea incrementată apare pe ieșirea  $S$ . Registrul își va menține starea (ieșirea)  $Q$  pe durata perioadei de ceas. La fiecare front crescător registrul va prelua valoarea stabilă de pe intrarea  $D$ , și anume  $S=Q+1$ , care va deveni noua valoare a registrului. Diagrama de timp pentru circuitul realizat este prezentată în continuare. Deoarece semnalele  $D$  (egal cu  $S$ ) și  $Q$  sunt pe  $n$  biți, pe diagramă se vor scrie direct valorile echivalente în sistemul zecimal (pentru simplitate). Se presupune că valoarea registrului a fost resetată la 0 înainte de  $T_0$ , prin activarea semnalului  $rst$ .

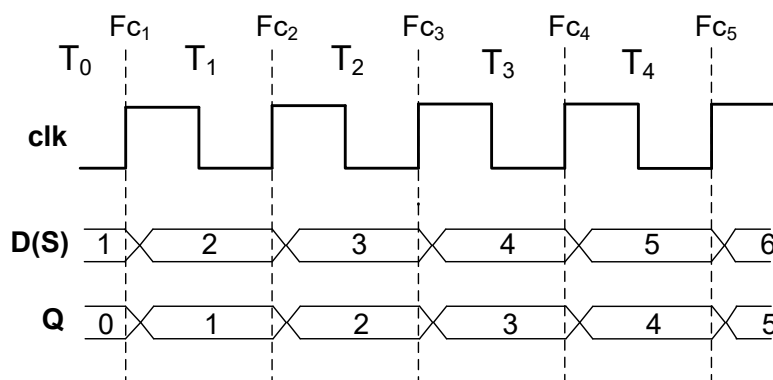


Fig. 6.14. Diagrama de timp pentru circuitul de numărare. Valoarea registrului de pe ieșirea  $Q$  reprezintă contorul care ne arată câte fronturi crescătoare ( $F_c$ ) au apărut de la perioada  $T_0$ .

Tranzițiile pentru  $D$  și  $Q$  au fost desenate ușor întârziate față de frontul crescător



## 7. Circuite de memorie

Circuitele de memorie sunt circuite care pot stoca/memora date în format binar. Un exemplu simplu de astfel de circuit, discutat în capitolul anterior, este registrul (respectiv bistabilul D, care poate memora un bit de date).

În acest capitol se vor discuta aspecte generale legate de circuitele de memorie folosite într-un calculator (de tip RAM, respectiv ROM). Se va pune accent pe înțelegerea modului în care funcționează și cum pot fi utilizate astfel de circuite, în contextul proiectării procesorului MIPS. Pentru detalii suplimentare privind implementarea fizică a circuitelor de memorie, funcționare, subtipuri existente și performanță, se recomandă consultarea bibliografiei suplimentare: (Harris & Harris, 2013), subcapitolul 5.5, respectiv (Patterson & Hennessy, 2013) subcapitolul 5.2.

Memoriile RAM (Random Access Memory) sunt memorii cu acces aleatoriu, care permit operații de citire, respectiv de scriere a datelor în locațiile din memorie. Ideea de *aleatoriu* se referă la posibilitatea accesării directe a oricărui element, fără ca timpul de acces să depindă de poziția unde se află în memorie. În contrast, un mediu de memorie cu acces *secvențial* ar implica trecerea succesivă prin mai multe locații situate între locația accesată anterior și locația curentă de accesat (ex. mediu de stocare pe bandă magnetică).

Memoria RAM este o memorie volatilă, în care datele sunt persistente doar pe durata alimentării cu energie.

În funcție de tehnologia folosită pentru implementarea fizică a celulei de memorie (care stochează un bit de date), există două categorii principale de memorie RAM: statică sau dinamică. Memoria RAM statică (Static RAM – SRAM) are o celulă de bit mai complexă (realizată din mai mulți tranzistori – depinde de implementare), dar are avantajul vitezei crescute de acces (permite funcționarea la frecvențe mari de ceas). Memoria RAM dinamică (Dynamic RAM – DRAM) are o celulă de bit mult mai simplă (formată dintr-un tranzistor și un condensator), dar cu viteză de acces mai scăzută față de SRAM. În schimb, costul de fabricație este mai redus la DRAM față de SRAM.

Memoriile de tip ROM (Read Only Memory) sunt memorii non-volatile (datele persistă și atunci când circuitul de memorie nu este alimentat), protejate la scriere. Pentru memoriile ROM se pot face doar operații de citire a datelor, scrierea nefiind posibilă în condiții normale de utilizare (scrierea se poate face doar cu echipamente speciale). De asemenea, memoriile ROM permit accesarea aleatorie a datelor.

### 7.1. Concepte de bază: adrese, date, organizare

Principiul de indexare, sau numerotare a elementelor/locațiilor, este specific memoriilor care permit acces aleatoriu.

O analogie ar fi un raft cu sute de dosare, cu două situații ipotetice. În prima, dosarele nu prezintă vizibil nicio indicație legată de poziția lor (ordinea relativă față de primul dosar). În acest caz, o sarcină simplă ca “Găsește dosarul de pe poziția 51” implică numărarea dosarelor, începând cu primul dosar, până se ajunge la dosarul căutat. Aceasta este situația de accesare secvențială. În a doua situație, fiecare dosar are atașată vizibil poziția (indexul). Extragerea dosarului căutat este directă, fără nevoia de a număra (trece secvențial prin) dosarele anterioare. Această variantă este echivalentul accesului aleatoriu.

O altă posibilă analogie, ușor de înțeles pentru cititorii care au cunoștințe de programare / structuri simple de date, este comparația dintre o lista înlănțuită și un șir. În

cazul listei, se trece secvențial prin toate elementele anterioare celui căutat. Pentru a accesa un element dintr-un șir, se folosește direct indexul elementului (relativ la începutul șirului).

Pentru circuitele de memorie, indexul este echivalent cu *adresa*.

Elementele stocate în diferite locații de memorie reprezintă valori binare care vor fi referite ca *date* în acest subcapitol. *Observație:* termenul nu este identic cu conceptul de date dintr-un program, deoarece în memorie se stochează, pe lângă datele folosite de programe, și reprezentarea binară a codului programelor – la acest subiect vom reveni.

Organizarea memoriei se referă la aranjarea succesivă a locațiilor, precum și la dimensiunea elementelor standard din memorie care pot fi accesate. Chiar dacă celula de bază a unui circuit de memorie este de 1 bit, elementele care pot fi accesate sunt în general grupuri de biți consecutivi, multipli de 8 (a se vedea capitolul 1). Astfel, conținutul unei memorii poate fi văzut ca o înșiruire de octeți (sau multipli), fiecare având un index (adresă) egal cu poziția sa în memorie (primul are adresa 0). În figura următoare sunt prezentate mai multe exemple, pentru diferite dimensiuni ale memoriei.

Adresa	Date	Adresa	Date	Adresa	Date
0	8 biți	0	8 biți	0	8 biți
1	8 biți	1	8 biți	1	8 biți
2	8 biți	2	8 biți	2	8 biți
3	8 biți	3	8 biți	3	8 biți
.	.	.	.	.	.
.	.	.	.	.	.
.	.	.	.	.	.
1022	8 biți	1,048,574	8 biți	1,073,741,822	8 biți
1023	8 biți	1,048,575	8 biți	1,073,741,823	8 biți

Fig. 7.1. Organizarea adrese/date pentru trei variante de memorie (de la stânga la dreapta) de dimensiune  $2^{10}$  (1 Kilobyte),  $2^{20}$  (1 Megabyte), respectiv  $2^{30}$  octeți (1 Gigabyte). Intervalul de adresă este  $0..dimensiune-1$

Dimensiunea în octeți a unui circuit de memorie este o putere a lui 2, din motive care țin de implementarea fizică a mecanismului de adresare. Unitățile de măsură uzuale sunt multipli de octeți: Kilo, Mega, Giga etc. În cazul circuitelor de memorie, prin acești multipli se înțeleg valori apropiate de valorile clasice (o mie, un milion, un miliard), dar care sunt puteri ale lui 2, conform tabelului următor. Pentru generalitate, se va folosi simbolul din engleză pentru octet (B – *Byte*, litera B mare, pentru a evita confuzia cu bit).

Tabel 7.1. Unitățile uzuale folosite pentru dimensiunea unui circuit de memorie

1 KB (Kilobyte – Kilo-octet)	=	$2^{10} = 1024 \text{ B}$
1 MB (Megabyte)	=	$2^{20} = 1,048,576 \text{ B}$
1 GB (Gigabyte)	=	$2^{30} = 1,073,741,824 \text{ B}$
...	...	...

Adresa este un semnal de intrare pentru circuit, pe un număr de biți care să permită reprezentarea binară a tuturor adreselor (valorilor de index) posibile. De aici apare și constrângerea ca dimensiunea memoriei să fie  $2^n$  locații, unde  $n$  este numărul de biți de adresă. Exemple numerice: (a) o memorie de 64 de locații are 6 biți de adresă; (b) o memorie de  $16 \text{ KB} = 2^4 * 2^{10} = 2^{14}$  octeți are 14 biți de adresă. Se folosește frecvent noțiunea de linii de adresă (câte o linie/bit) sau de date.

Pentru a accesa locația  $X$  dintr-o memorie care are adresa pe  $n$  biți, se convertește  $X$  în sistemul binar, pe  $n$  biți, și valoarea respectivă este trimisă pe liniile de adresă. Exemplu: pentru o memorie de 512 locații, sunt 9 biți (linii) de adresă; pentru a accesa locația cu adresa 234, aceasta se convertește în binar ( $234=011101010_2$ ) și se trimite pe cele 9 linii de adresă. Selecția propriu-zisă a locației, cu scopul citirii sau scrierii, este realizată printr-o serie de circuite suplimentare cu rol de multiplexare/demultiplexare, dar aceste detalii nu vor fi prezentate în continuare.

Un ultim aspect, legat de organizare, este noțiunea de *cuvânt* pentru memorie. Chiar dacă memoria permite adresarea individuală a fiecărui octet, pentru a avea o comunicație rapidă cu procesorul, este optimizat accesul cuvintelor aflate la adrese multiple de dimensiunea cuvântului. În acest caz, memoria este organizată pe *cuvânt*, adresabilă pe octet. Ca exemplu, în figura următoare se prezintă organizarea pentru o memorie de 4 GB, adresabilă pe octet, cu un cuvânt de 32 biți (4 octeți). Fiecare cuvânt din memorie este situat la adrese multiple de 4 octeți. Memoria conține 1 Giga ( $2^{30}$ ) cuvinte de câte 4 octeți. Fiind multiple de 4 octeți, adresele cuvintelor vor avea tot timpul ultimii 2 biți egali cu 0.

Adresă	Date			
$0 = 00...0000_2$	8 biți	8 biți	8 biți	8 biți
$4 = 00...0100_2$	8 biți	8 biți	8 biți	8 biți
$8 = 00...1000_2$	8 biți	8 biți	8 biți	8 biți
$12 = 00...1100_2$	8 biți	8 biți	8 biți	8 biți
.	.	.	.	.
.	.	.	.	.
.	.	.	.	.
$2^{32}-8 = 11...1000_2$	8 biți	8 biți	8 biți	8 biți
$2^{32}-4 = 11...1100_2$	8 biți	8 biți	8 biți	8 biți

Fig. 7.2. Organizarea memoriei de 4 GB adresabilă pe octet, cu cuvântul pe 32 biți

În continuare, se vor prezenta exemple de memorie RAM, ROM și un bloc de registre (uzual, în domeniul calculatoarelor, se folosește și pluralul *registri*, chiar dacă acesta poate fi acceptat cel mult ca neologism). Descrierile care urmează sunt particulare, ca proprietăți, pentru tipul de circuite folosite la proiectarea unui procesor MIPS, varianta cu ciclu unic (Patterson & Hennessy, 2013). Se vor explica semnalele de intrare/ieșire și control (se vor folosi de obicei denumirile în engleză, asemănător felului în care apar în literatura de specialitate) și modul cum funcționează, fără multe detalii legate de implementarea fizică și structura hardware internă.

## 7.2. Memoria RAM

Circuitul de memorie RAM este un circuit de memorie cu  $n$  linii de adresă care permite citire asincronă și scriere sincronă, adresabil pe octet, cu un cuvânt de 32 biți. Schema bloc este prezentată în Fig. 7.3. *Observație:* pentru circuitele de memorie descrise în această carte, termenul de asincron (la citire) este echivalent cu combinațional.

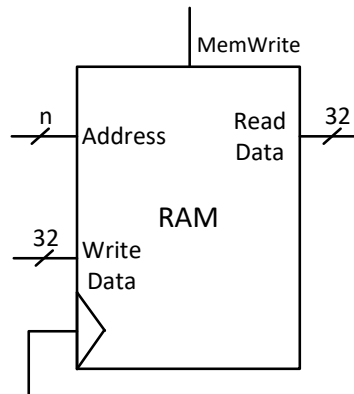


Fig. 7.3. Schema bloc pentru memoria RAM, cu semnalele de adresă (*Address*, intrare), date citite (*Read Data*, ieșire), intrarea de date pentru scriere (*Write Data*), semnalul de control pentru validarea scrierii *MemWrite* și semnalul de ceas

În cazul procesorului MIPS, circuitul va avea  $n=32$  de linii de adresă, deci va fi o memorie RAM de 4 GB, care conține 1 Giga de cuvinte de 32 de biți.

Din punct de vedere constructiv, pe lângă celulele care stochează efectiv valoarea biților, circuitul conține intern și toată logica necesară pentru efectuarea operațiilor de citire și de scriere.

Operația de citire este asincronă, fiind independentă de semnalul de ceas. Funcționarea este la fel ca la circuite combinaționale: ieșirea *Read Data* depinde doar de valoarea curentă a intrării *Address*. De fiecare dată când apare o nouă combinație de biți pe *Address* (practic, o adresă dorită), elementul de 32 de biți de la adresa furnizată apare pe *Read Data*.

Operația de scriere se desfășoară similar cu scrierea în bistabilul D sincron, cu semnal de activare a scrierii:

1. Se pun și mențin stabile valorile dorite (adresa, respectiv valoarea care se va memora) pe câmpul *Address* și pe *Write Data*, se pune valoarea 1 pe semnalul de activare a scrierii *MemWrite*
2. Scrierea efectivă în locația selectată are loc pe primul front crescător de ceas.

### 7.3. Memoria ROM

Circuitul de memorie ROM este un circuit de memorie protejat la scriere, cu  $n$  linii de adresă care permite citire asincronă (este un model de ROM idealizat, fără semnal de ceas, particular pentru procesorul MIPS). Schema bloc este prezentată în figura următoare.

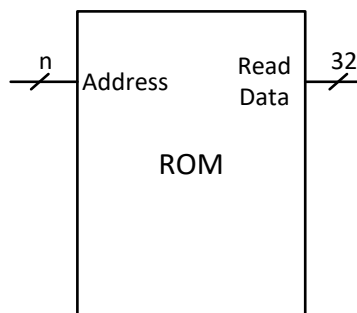


Fig. 7.4. Schema bloc pentru memoria ROM, cu semnalele de adresă (*Address*, intrare) și date citite (*Read Data*, ieșire)

Operația de citire se face la fel ca la memoria RAM descrisă anterior, funcționând ca un circuit combinațional: ieșirea *Read Data* depinde doar de valoarea curentă a intrării *Address*.

În cazul procesorului MIPS, memoria ROM are  $n=32$  de linii de adresă, organizată pe cuvânt de 32 de biți, dar adresabilă pe octet.

#### 7.4. Blocul de registre

Un bloc de registre (Register File – RF, acronim care va fi folosit frecvent în paragrafele următoare) este un circuit care conține mai multe registre individuale, organizat de obicei ca o memorie de mici dimensiuni. Blocul de registre este o componentă specifică a unui procesor, fiind spațiu de memorare a datelor aflate în curs de procesare sau procesate frecvent. Este foarte rapid (realizat cu tehnologie SRAM) în comparație cu memoria principală (tehnologie DRAM). Față de un circuit standard de memorie, un bloc de registre poate permite accesări simultane pentru elementele conținute prin existența mai multor *porturi* de citire și/sau scriere.

Blocul de registre folosit pentru procesorul MIPS, versiunea cu ciclu unic, conține 32 de registre de câte 32 de biți fiecare. Blocul are două porturi de citire asincronă și un port de scriere sincronă. În acest context, prin *port* se înțelege o combinație de linii de adresă (care selectează registrul dorit) și linii de date (pe care apar datele din registrul sau prin care se transmit datele care se vor memora). Fiind  $32 = 2^5$  registre în bloc, vor fi necesare 5 linii de adresă (semnal de adresă pe 5 biți) pentru fiecare port. Fiecare registru este identificat în mod unic prin indexul lui din bloc: r0, r1, r2, ..., r30, r31 sau RF[0], RF[1], RF[2], ..., RF[30], RF[31] – dacă RF este privit simplu, ca un șir.

Schema standard pentru blocul de registre este prezentată în Fig. 7.5.

Citirile din RF sunt asincrone (combinaționale), fiind permise două citiri simultane. Adresele registrelor dorite sunt puse pe liniile de adresă *Read Address 1*, respectiv *Read Address 2*, iar valorile stocate în cele două registre vor apărea pe liniile de date *Read Data 1*, respectiv *Read Data 2*.

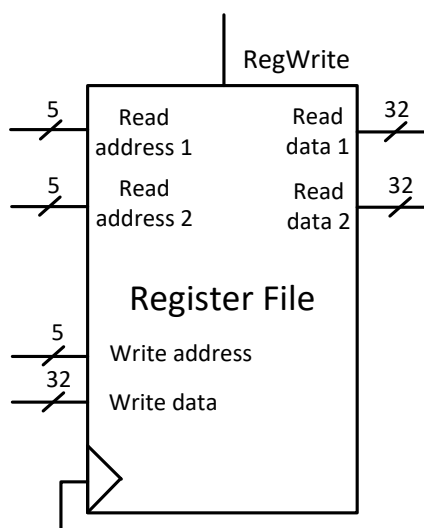


Fig. 7.5. Schema blocului de registre folosit pentru procesorul MIPS, cu cele două porturi de citire asincronă (Read Address 1/Read Data 1 și Read Address 2/Read Data 2) și portul de scriere sincronă (Write Address / Write Data, cu semnalul de activare a scrierii RegWrite)

Scrierea sincronă implică (la fel cu scrierea într-un registru cu semnal de activare a scrierii):

1. Se pun și mențin stabile valorile dorite (adresa, respectiv valoarea care se va memora) pe câmpurile *Write Address* și pe *Write Data*, iar pe semnalul de activare a scrierii *RegWrite* se pune valoarea 1
2. Scrierea efectivă în locația selectată are loc pe primul front crescător de ceas.

## 8. Aplicativ: construirea unor circuite mai complexe

În acest capitol se vor prezenta două tehnici prin care, aplicând anumiți pași standard de proiectare, se pot obține circuite complexe care rezolvă anumite probleme. Tipul de probleme rezolvate sunt formule algebrice. De asemenea, se introduce ideea de separare a unui circuit în *calea de date* (eng. datapath) și *unitatea de control* (eng. control unit).

Cele două tehnici sunt sinteza de nivel înalt (subcapitolul 8.1) și metoda bazată pe transferul dintre registre (subcapitolul 8.2).

### 8.1. Sinteza de nivel înalt

Sinteza de nivel înalt, High Level Synthesis – HLS (Coussy, Gajski, Meredith, & Takach, 2009), este o tehnică prin care, pornind de la descrierea comportamentală a circuitului (ce trebuie să facă?), se obține o descriere structurală a circuitului (cu elementele componente și legăturile dintre ele). În acest subcapitol se va prezenta o versiune (mult) simplificată a metodei HLS, aplicată pentru calculul expresiei algebrice  $Y = a * X + b$ , unde  $a$ ,  $X$  și  $b$  sunt variabile de intrare, iar  $Y$  este rezultatul care trebuie calculat.

Circuitul care rezultă prin metoda HLS va fi format din *calea de date* și *unitatea de control*.

*Calea de date* reprezintă totalitatea elementelor de memorare (ex. registre, memorii), a unităților funcționale (unitățile care fac calcule pe datele din circuit – ex. sumator, UAL etc) și a conexiunilor dintre ele, necesare pentru rezolvarea problemei.

*Unitatea de control* este un circuit care controlează calea de date prin intermediul *semnalelor de control*. Semnalele de control sunt acele semnale care influențează felul în care datele sunt transferate prin calea de date (ex. semnale de selecție la multiplexoare, semnale de activare a scrierii la elementele de memorare etc.). Unitatea de control poate lua decizii și pe baza *semnalelor de stare*, acestea fiind semnale din calea de date care au diferite semnificații (exemple în capitolele următoare).

Urmează două variante de rezolvare, în funcție de constrângerile de resurse disponibile (unități funcționale).

#### 8.1.1. Varianta 1: Fără constrângeri de resurse

În această variantă se presupune că cel care proiectează circuitul are la dispoziție un număr nelimitat de unități funcționale, iar circuitul obținut  $C$  este combinațional. Alternativ, se folosește denumirea de *ciclu unic* – dacă circuitul obținut este integrat într-un circuit părinte care funcționează sincron, tot calculul executat de  $C$  trebuie să se încadreze ca durată în perioada de ceas. În acest caz, nu este necesară o unitate de control pentru circuitul obținut.

Circuitul trebuie să calculeze expresia  $Y = a * X + b$ . În expresie se identifică două operații, pentru care trebuie alocate două unități funcționale: o sumă, respectiv un produs. Se vor folosi un sumator și un înmulțitor pentru efectuarea operațiilor. Realizarea căilor de date este directă, bazată pe fluxul de date din problemă. Fiecare operație este asociată cu o unitate funcțională, iar operanzii vor fi intrări în unitatea funcțională.

Se începe construirea căilor de date (Fig. 8.1) cu operația prioritară, înmulțitorul, care primește pe intrări  $a$  și  $X$ , producând  $(a * X)$  ca rezultat. Se adaugă următoarea unitate, sumatorul, care are ca intrări  $(a * X)$ , deci ieșirea din înmulțitor, și  $b$ . La ieșirea sumatorului

se va afla rezultatul  $Y$ . Se presupune că variabilele sunt reprezentate ca semnale pe  $n$  biți, furnizate din exterior (de către alte circuite mai complexe, în care circuitul proiectat ar trebui integrat). De asemenea, pentru a nu complica mult calea de date, se consideră că produsul  $a*X$  se poate reprezenta corect tot pe  $n$  biți. În general, produsul dintre două numere pe  $n$  biți se reprezintă pe  $2n$  biți, pentru a evita situațiile de depășire.

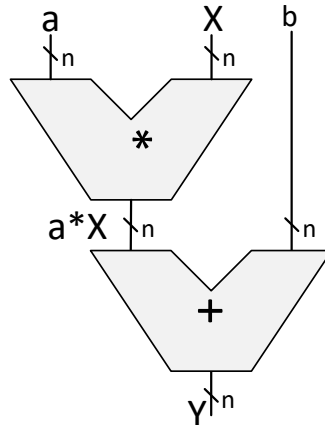


Fig. 8.1. Circuitul care calculează  $Y = a*X + b$ , obținut prin aplicarea metodei HLS

Pentru expresii complexe, cu un număr ridicat de operații, necesarul de resurse poate fi destul de ridicat. În funcție de obiectivele de proiectare, varianta combinațională poate fi prea costisitoare (multe resurse, cost ridicat și putere mare consumată de circuit).

### 8.1.2. Varianta 2: Cu constrângeri de resurse

O constrângere tipică de resurse este folosirea unui număr redus de unități funcționale, mai mic decât numărul de operații existente în expresia de calculat.

Pentru exemplul ales se presupune că este disponibilă o singură unitate funcțională de tip UAL, care poate executa la un moment dat una dintre cele două operații necesare: sumă sau produs.

Nu este posibilă obținerea unui circuit combinațional pentru calculul expresiei. Formula de calculat trebuie descompusă pe pași (cu rezultate intermediare). Se pornește de la fluxul de date și se ține cont de faptul că, pe fiecare pas, se poate executa o singură operație de către unitatea funcțională:

Pasul 1: *Rezultat intermediar*  $\leftarrow a*X$

Pasul 2: *Rezultat final*  $\leftarrow$  *Rezultat intermediar*  $+ b$

Ideea de pași discreți va fi dată de semnalul de ceas în implementarea circuitului, soluția fiind *multi ciclu*. Rezultatele (intermediare sau finale) vor fi stocate în elemente de memorare (registre sau memorii), iar operațiile vor fi executate de către unitățile funcționale.

Dacă s-ar memora separat rezultatul intermediar, atunci ar fi necesare 2 registre pentru memorarea rezultatelor. Deoarece rezultatul intermediar nu este necesar la final (după pasul 2), se poate optimiza circuitul prin folosirea unui singur registru *Reg*, care pe prima perioadă de ceas  $T1$  (pasul 1) va stoca  $a*X$ , iar pe a doua perioadă de ceas  $T2$  (pasul 2) rezultatul final. De subliniat că memorarea efectivă în registru are loc pe frontul crescător de ceas. Pe frontul crescător dintre  $T1$  și  $T2$  se va memora în *Reg* valoarea  $a*X$  care, apoi,



va fi disponibilă pe ieșirea *Reg* pe durata lui T2, iar la frontul dintre T2 și T3 (următoarea perioadă) se va scrie rezultatul final în *Reg* (valoarea fiind disponibilă la ieșirea lui *Reg* pe perioada T3).

Pentru a ușura proiectarea căii de date, se realizează un tabel (Tabel 8.1) cu intrările și ieșirile unităților funcționale și a registrelor folosite. În acest tabel se notează, pentru fiecare perioadă de ceas, ce valori (și sursele de unde se citesc aceste valori) trebuie să se găsească pe fiecare semnal, conform pașilor de calcul. Se presupune că la începutul lui T1 semnalele de intrare *a*, *b* și *X* sunt inițializate și sunt menținute stabile pe perioada calculului (până la finalul lui T2, această sarcină o are un circuit părinte, absent în descrierea curentă).

Tabel 8.1. Valorile semnalelor din circuit, pe fiecare perioadă de ceas

	T1	T2
UAL in 1	<i>a</i>	<i>a</i> * <i>X</i> (Reg out)
UAL in 2	<i>X</i>	<i>b</i>
UAL out	<i>a</i> * <i>X</i>	<i>a</i> * <i>X</i> + <i>b</i>
Reg in	<i>a</i> * <i>X</i> (UAL out)	<i>a</i> * <i>X</i> + <i>b</i> (UAL out)
Reg out	-	<i>a</i> * <i>X</i>

Din acest tabel se deduc mult mai ușor conexiunile necesare în calea de date. Pentru asamblarea căii de date se aplică următorii pași:

1. Se desenează elementele componente (de memorare și unitățile funcționale), cu o spațiere corespunzătoare
2. Se stabilesc conexiunile dintre elemente. Aici se folosește tabelul - pentru fiecare semnal de intrare se analizează linia corespunzătoare din tabel:
  - a. Dacă pe perioade diferite de ceas trebuie legate surse diferite la respectiva intrare, se va interpune un multiplexor (cu câte intrări sunt necesare)
  - b. Dacă pe toate perioadele de ceas aceeași sursă furnizează datele, se face o conexiune de la sursa respectivă la intrarea elementului
  - c. Exemplu: pentru prima intrare UAL (în 1), pe T1 ar trebui legat *a*, iar pe T2 ieșirea din registrul *Reg*. Se leagă cele două surse la un multiplexor, iar ieșirea multiplexorului se leagă la prima intrare a UAL. Pe intrarea lui *Reg*, ambele valori vin de la ieșirea UAL, nefiind necesar un multiplexor. Raționamentul este similar pentru restul conexiunilor.

Aplicând acești pași se obțin căile de date din Fig. 8.2. Din tabel rezultă că fiecare intrare din UAL va avea câte un multiplexor, care va permite selecția sursei necesare pe fiecare perioadă de ceas. Se denumesc semnalele de control apărute prin introducerea multiplexoarelor, și anume semnalele de selecție *S*<sub>1</sub>, respectiv *S*<sub>2</sub>. Semnalul care alege operația curentă, pe care o execută UAL, este ALUOp (0 pentru \*, 1 pentru +). Rezultă 3 semnale de control, pe câte un bit fiecare.

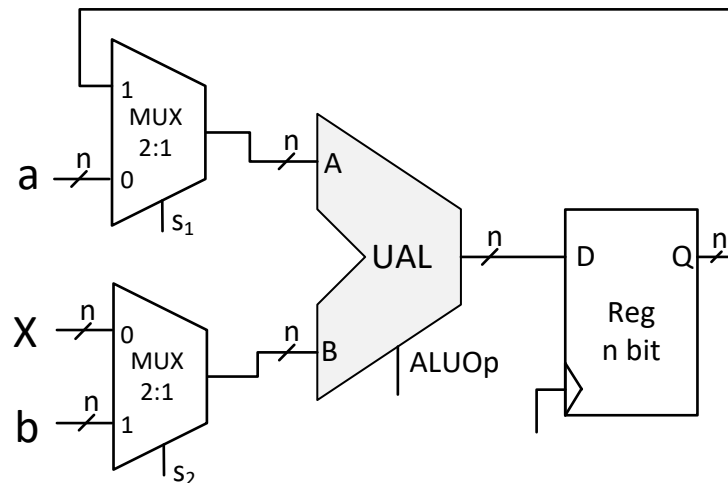


Fig. 8.2. Calea de date pentru varianta multi ciclu. Pentru a ușura desenarea conexiunilor, la multiplexoare s-a pus explicit pe schemă valoarea semnalului de selecție asociat fiecărei intrări (la multiplexorul de sus, de exemplu, sunt puse invers, 0 pentru intrarea de jos, 1 pentru sus, tocmai pentru a nu intersecta traseele)

Pentru a avea un circuit funcțional, calea de date trebuie comandată să execute cei doi pași necesari. Acesta este rolul unității de control. Pentru a proiecta unitatea de control se folosește conceptul de automat cu stări finite (Finite State Machine – FSM). Acesta este un subiect de bază în domeniul proiectării logice și se va prezenta sumar în acest subcapitol, deoarece este de interes doar pentru exemplul curent. Nu este util în capitolele următoare pentru proiectarea procesorului MIPS varianta *ciclu unic* (ar fi absolut necesar pentru a proiecta variante *multi ciclu* de procesor). Pentru cititorii interesați de acest subiect, se recomandă studierea capitolului 3 din (Harris & Harris, 2013).

Un automat cu stări finite este un model teoretic comportamental care este format dintr-un set finit de stări, tranziții între stări și acțiuni asociate stărilor. În orice moment, automatul se poate afla strict într-una dintre stările din setul finit.

Pentru a construi unitatea de control, se definește un automat finit cu 2 stări posibile, pe care le asociem cu cei doi pași: stările ST1 și ST2.

Tranzițiile posibile sunt simple: din ST1 automatul trece în starea ST2, iar din ST2 trece înapoi în ST1. Într-un regim de funcționare continuă se repetă calculul pentru următoarele date de intrare, dacă sunt furnizate pe intrările *a*, *b* și *X*.

Acțiunile sunt de fapt comenzile pe care unitatea de control le transmite căii de date, în fiecare stare (asociată cu perioada de ceas pentru problema discutată). Transmiterea comenzilor se realizează prin trimiterea valorilor adecvate pe semnalele de control:

ST1:  $S_1 \leq 0$  (trece *a*),  $S_2 \leq 0$  (trece *X*),  $ALUOp \leq 0$  (execută \*)

ST2:  $S_1 \leq 1$  (trece *Reg*),  $S_2 \leq 1$  (trece *b*),  $ALUOp \leq 1$  (execută +)

Proiectarea automatului se face folosind un registru special, de stare (*state register*), care codifică în binar starea curentă, și construind logica de calcul (combinațional) pentru starea următoare (*next state logic*):

1. Se codifică stările în binar. Sunt doar două stări, deci e suficient 1 bit: 0 pentru ST1 și 1 pentru ST2. Acest bit se va memora într-un registru de stare pe 1 bit (bistabil D sincron, cu semnal de reset pentru a inițializa automatul

la pornirea circuitului). Semnalul de ceas care controlează scrierea în registrul de stare va da cadența automatului

2. Se proiectează logica pentru calculul stării următoare. În exemplul dat, tranzițiile sunt foarte simple: în starea 0, starea următoare este 1, iar în starea 1 starea următoare este 0. Un simplu inversor rezolvă calculul stării următoare.

Schema unității de control este prezentată în continuare.

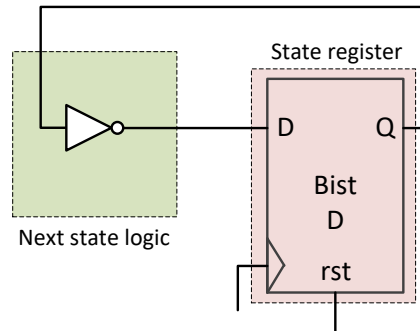


Fig. 8.3. Schema unității de control cu cele două componente evidențiate (logica de calcul pentru starea următoare, respectiv registrul de stare)

Ultimul pas este legarea unității de control la calea de date. Starea curentă (ieșirea Q a registrului de stare) trebuie decodificată în valorile necesare pentru semnalele de control. Fiind un exemplu foarte simplu (cu valori alese convenabil de la început pentru semnalele de control), valoarea stării în binar coincide cu valorile semnalelor de control pe fiecare stare. Pentru situații mai complexe se pot folosi inversoare unde e cazul sau, mai general, când starea e codificată pe mai mulți biți, circuite decodificatoare.

Schema completă a circuitului este prezentată în figura următoare.

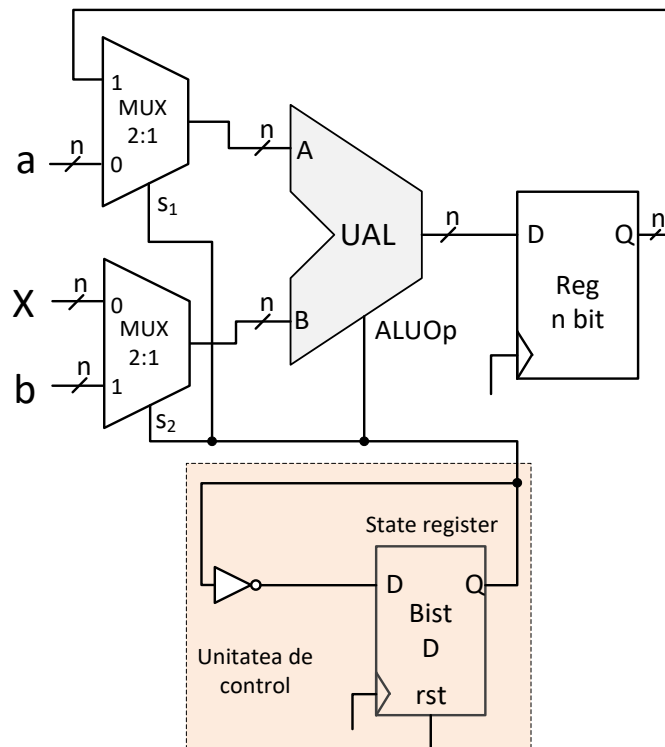


Fig. 8.4. Soluția completă cu calea de date și unitatea de control. Semnalul de ceas este același

Când este disponibil rezultatul final  $Y$ ? În cazul în care circuitul funcționează încontinuu, la finalul perioadei de ceas ( $T_2$ ), corespunzătoare stării  $ST_2$ , se scrie valoarea finală a lui  $Y$  în *Reg*. Această valoare persistă pentru perioada următoare de ceas ( $T_3$ ) în registru (de unde poate fi citită de circuitul părinte). Persistă doar o perioadă de ceas (pe  $T_3$ ), deoarece unitatea de comandă reintră în  $ST_1$  după  $ST_2$  (repetă calculul de la început), iar la finalul noii stări  $ST_1$  (perioada  $T_3$ ) se va suprascrie *Reg* cu rezultatul intermediar pentru noile valori de intrare ( $a, b, X$ ).

Aplicând sinteza de nivel înalt se obține o soluție particulară pentru expresia de calculat. Ce se întâmplă dacă se schimbă cerințele de proiectare, cu o expresie nouă (poate mai complexă) de calculat? În acest caz, este necesară reproiectarea de la început a circuitului, cu o nouă cale de date și unitate de control.

## 8.2. O abordare mai generală, spre conceptul de procesor

În acest subcapitol se va prezenta o metodă mai generală de proiectare, bazată pe transferul dintre registre. Circuitul obținut este mult mai ușor de modificat pentru expresii noi și ar trebui să poată calcula orice expresie care conține operații de tip sumă, diferență, înmulțire și/sau împărțire. Ca unitate funcțională se presupune existența unei UAL care poate executa aceste operații. Operanzii se vor citi dintr-un bloc de registre de 32x32 biți (la fel ca cel descris în subcapitolul 7.4), iar rezultatele se vor salva în același bloc.

Pornind de la aceste constrângeri, fiecare expresie de calculat se va descompune într-o succesiune de operații simple, în ordinea priorității, cu 2 operanzi de intrare și un rezultat. Operanzii de intrare vor fi, după caz, variabilele originale sau rezultate intermediare. Practic, se va introduce conceptul de instrucțiune, respectiv program, specific unui procesor. Instrucțiunile vor fi asociate cu operațiile de bază pe care le poate executa UAL.

Se va merge pe o arhitectură de tip *ciclu unic*, în care fiecare instrucțiune se execută pe o perioadă de ceas. Sunt necesari următorii pași de proiectare:

1. Definirea unui set de instrucțiuni, cu sintaxă și format binar de reprezentare
2. Analiza fiecărei instrucțiuni pentru a identifica componentele necesare în căile de date
3. Asamblarea căii de date
4. Construirea unității de control.

### 8.2.1. Setul de instrucțiuni

Pentru fiecare dintre cele 4 operații de bază se definește câte o instrucțiune. Fiecare instrucțiune este echivalentă cu executarea unei operații între cei doi operanzi sursă și salvarea rezultatului în operandul destinație. Sintaxa generică a acestor instrucțiuni este:

**Op** *rd, rs1, rs2*

Unde **Op** este operația de executat, cu etichetele particulare add, sub, mul, div, iar *rd* este registrul (operandul) destinație în care se pune rezultatul operației dintre cei doi operanzi (sursă) de tip registru *rs1* și *rs2*. Operanzii sursă ai instrucțiunii vor fi locațiile din blocul de registre:  $r_0, r_1, \dots, r_{31}$ .

Expresia simplă din subcapitolul anterior se poate calcula cu două astfel de instrucțiuni succesive. Se presupune că valorile  $a, b, X$  sunt stocate în registrele din bloc:

r1, r2, r3, în această ordine. Se alege r4 cu rol de destinație (intermediară, respectiv finală). Secvența de instrucțiuni este:

```
mul r4, r1, r3 -- r4 va memora a*X
add r4, r4, r2 -- r4 va memora rezultatul final Y
```

O expresie mai complexă se rezolvă cu mai multe instrucțiuni, în ordinea priorității operațiilor și cu păstrarea rezultatelor intermediare în registre libere. De exemplu:

$$Y = (a + b * X) * a * X$$

La fel ca în exemplul anterior, se presupune că, inițial, r1=a, r2=b și r3=X. Secvența de instrucțiuni nu este unică, depinzând de ordinea de calcul aleasă:

```
mul r4, r1, r3 -- r4 va memora a*X
mul r5, r2, r3 -- r5 va memora b*X
add r5, r5, r1 -- r5 va memora a + b*X
mul r4, r4, r5 -- r4 va memora rezultatul final Y
```

Ca o remarcă, folosirea aceluiași registru pe post de destinație și sursă în aceeași instrucțiune nu este o problemă (de exemplu, în a treia instrucțiune). Pe durata execuției instrucțiunii (perioada de ceas), ca operand sursă, registrul va furniza valoarea anterioară execuției instrucțiunii, deoarece doar la finalul execuției se scrie noua valoare (când se termină perioada curentă de ceas și are loc scrierea pe front crescător în blocul de registre).

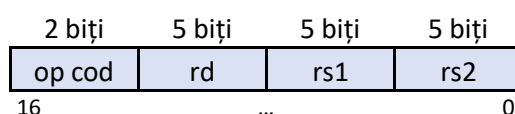
Secvența de instrucțiuni de mai sus, care se vor executa în ordinea în care sunt scrise, reprezintă un program (sau secțiune de program). Termenul folosit pentru acest tip de limbaj de programare este limbaj de asamblare. Acesta este cel mai jos nivel la care se pot scrie programe folosind o sintaxă predefinită, cu etichete simbolice de tip text pentru numele instrucțiunilor, numele operanzilor etc.

Pentru a executa o astfel de secvență de instrucțiuni, un circuit trebuie în primul rând să o înțeleagă. Așadar, aceste instrucțiuni trebuie traduse în limbajul circuitelor digitale: *cod binar*. Reprezentarea instrucțiunilor de asamblare în format binar mai poartă denumirea de *cod mașină* (eng. machine code).

Reprezentarea binară a instrucțiunilor se construiește pornind de la câmpurile componente din sintaxă. Toate instrucțiunile pentru exemplul ales au un format comun. Se identifică patru câmpuri în sintaxă/format: câmpul de operație și trei câmpuri de operanzi:

- pentru operație - fiind 4 valori posibile, se vor aloca 2 biți în formatul binar, cu o valoare unică pentru fiecare cod de operație/instrucțiune: add – 00<sub>2</sub>, sub 01<sub>2</sub>, mul 10<sub>2</sub>, div 11<sub>2</sub>
- pentru operanzi - valorile operanzilor se află în blocul de registre, deci în cod mașină se vor reprezenta adresele registrelor. Fiind un bloc de registre cu 32 de locații, sunt necesari câte 5 biți de adresă pentru fiecare din cei trei operanzi.

Rezultă formatul binar următor pentru instrucțiunile din set:



Fiecare instrucțiune necesită 17 biți pentru reprezentarea în format binar. Primii 2 biți, cei mai semnificativi (16..15), reprezintă codul de operație, iar celelalte grupuri de câte 5 biți reprezintă operandii: rd – 14..10, rs1 – 9..5, rs2 – 4..0.

Transpunerea unui program din asamblare în cod mașină se face respectând formatul de instrucțiune. Fiecare câmp este codificat binar: codul de operație conform convenției alese, iar adresele registrelor direct în binar pe 5 biți. Codul mașină pentru programul scris anterior este:

Asamblare	Cod mașină			
	op cod	rd	rs1	rs2
mul r4, r1, r3 --	10	00100	00001	00011
mul r5, r2, r3 --	10	00101	00010	00011
add r5, r5, r1 --	00	00101	00101	00001
mul r4, r4, r5 --	10	00100	00100	00101

### 8.2.2. Identificarea componentelor necesare + asamblarea parțială

În acest pas se analizează setul de instrucțiuni și transferurile ce au loc în calea de date și se identifică elementele necesare. Pentru a urmări mai ușor procesul de proiectare, se va prezenta în paralel și pasul de asamblare a părților relevante din calea de date.

Mai întâi se face o analiză a întregului proces de execuție pornind de la ideea de secvență de instrucțiuni în cod mașină. În primul rând, programul de executat va fi furnizat circuitului în formatul binar (cod mașină). De aici rezultă că este necesar un circuit care să păstreze codul mașină al programului. Se poate folosi un circuit de memorie RAM sau ROM. Pentru a urma tiparul care va fi folosit la proiectarea procesorului MIPS, se va alege o memorie ROM (model idealizat, citire asincronă). În această memorie ROM (Fig. 8.5) se va memora inițial programul de instrucțiuni, după care, în timpul execuției, din ea se vor face doar citiri. Dimensiunea unui element se va alege convenabil, de 17 biți. Dimensiunea memoriei se alege în funcție de dimensiunea maximă pe care o pot avea programele. De exemplu, presupunând programe de cel mult 256 de instrucțiuni, memoria va avea adresa pe 8 biți. Ieșirea de date va fi pe 17 biți.

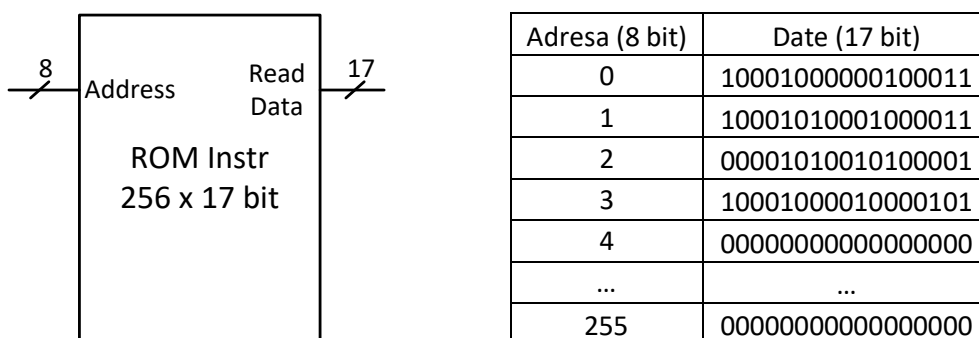


Fig. 8.5. Diagrama bloc a memoriei ROM de instrucțiuni și conținutul acesteia (pe primele 4 locații se află codul binar al programului de executat, în restul locațiilor 0)

Înainte de a discuta cum se face execuția efectivă (aceasta va rezulta din analiza instrucțiunilor), trebuie construit un mecanism pentru trecerea succesivă prin locațiile din memoria ROM de instrucțiuni, cu scopul de a citi pe rând fiecare instrucțiune. Practic, pe perioade succesive de ceas, la ieșirea memoriei ROM ar trebui să apară câte o instrucțiune,

în ordinea din program. Presupunând că instrucțiunile sunt puse la adrese consecutive în memoria ROM, începând cu adresa zero (Fig. 8.5), este necesar un circuit care să genereze adresele instrucțiunilor succesive. Un astfel de circuit se poate obține folosind un sumator și un registru (foarte asemănător cu numărătorul descris în subcapitolul 6.4). Registrul din acest circuit va reprezenta adresa instrucțiunii curente și poartă o denumire specifică în contextul procesoarelor: *contorul de program*, eng. Program Counter (PC, la MIPS) sau Instruction Pointer (IP, la Intel 8086).

Circuitul de generare a adreselor este prezentat în figura următoare.

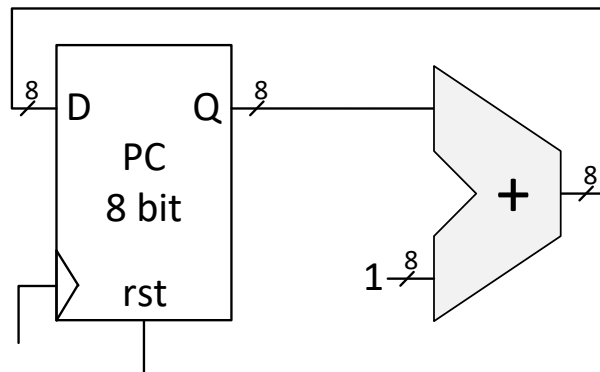


Fig. 8.6. Circuitul de generare a adreselor succesive pentru adrese pe 8 biți, format din registrul contor de program (PC) și un sumator

În acest moment se ajunge la un prim pas de asamblare a componentelor necesare, obținând partea din calea de date (Fig. 8.7) care extrage instrucțiunea din memorie și calculează următoarea adresă (eng. Instruction Fetch - IF). Se combină elementele enumerate până acum: ieșirea Q a PC, din circuitul de generare a adreselor, se leagă pe intrarea de adresă a memoriei ROM de instrucțiuni.

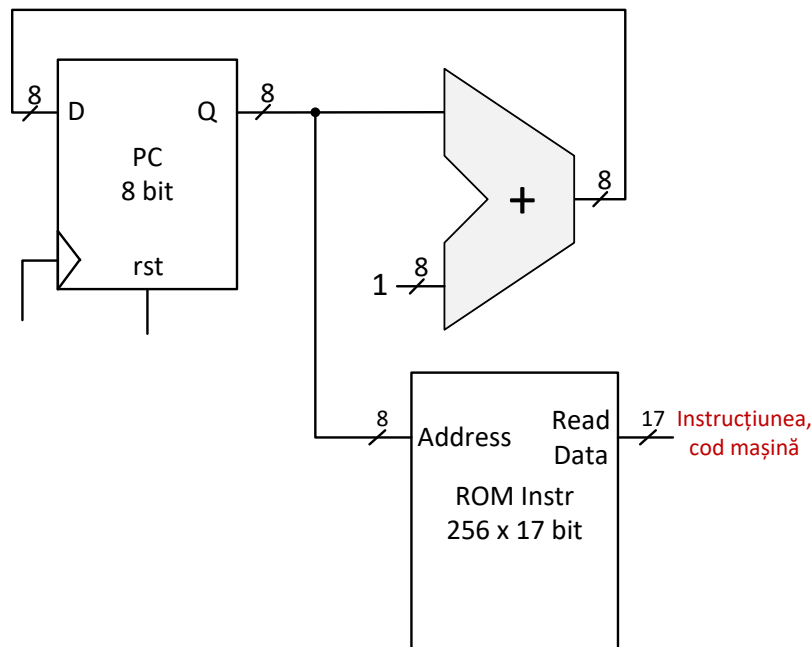


Fig. 8.7. Asamblarea căii de date: partea responsabilă de extragerea instrucțiunilor din memoria ROM. Ieșirea PC se leagă pe intrarea de adresă de la ROM

Procesul de analiză este dus mai departe, cu accent pe setul de instrucțiuni. Se va construi partea din calea de date cu rol de interpretare (decodificare, eng. Instruction Decode - ID), execuție (Instruction Execute – EX) și scriere a rezultatului (Write Back – WB). Aceste părți, cu rol particular, ale căii de date sunt specifice și pentru procesorul MIPS. Din acest motiv, s-a menționat și denumirea standard folosită în literatura de specialitate.

Pentru interpretarea/decodificarea biților din formatul de instrucțiune I(16..0), se analizează fiecare câmp individual:

- *op cod* – I(16..15) reprezintă codul de operație, acesta trebuie interpretat de către unitatea de control (se va reveni la acest subiect)
- *rd* – I(14..10) reprezintă registrul destinație (adresa lui), acesta este relevant pentru partea de scriere a rezultatului
- *rs1* – I(9..5) reprezintă registrul sursă 1 (adresa lui), acesta este relevant pentru partea de citire a operanzilor din blocul de registre
- *rs2* – I(4..0) reprezintă registrul sursă 2 (adresa lui), acesta este relevant pentru partea de citire a operanzilor din blocul de registre.

Considerând numărul și rolul operanzilor de tip registru din instrucțiuni (unul destinație – necesită scriere la finalul execuției instrucțiunii, doi sursă – necesită citire pe durata execuției) rezultă anumite proprietăți ale blocului de registre. Acesta trebuie să aibă două porturi de citire asincronă și un port de scriere sincronă. Se va folosi blocul de registre RF standard prezentat în subcapitolul 7.4.

Instrucțiunile au un format comun, execuția (pe o singură perioadă de ceas!) mergând după același tipar ca transfer între registre:

$$RF[ I(14..10) ] \leftarrow RF[ I(9..5) ] \text{ op } RF[ I(4..0) ] \quad (8.1)$$

Această descriere a transferului (cu operațiile aferente) dintre registre este foarte importantă în procesul de proiectare (Register Transfer Level – RTL). Prin analiza atentă a descrierii RTL pentru fiecare instrucțiune, se deduc atât elementele necesare, cât și conexiunile dintre ele.

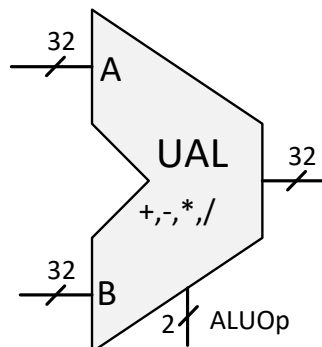


Fig. 8.8. UAL pe 32 de biți cu cele 4 operații necesare

Din descrierea RTL (8.1) de mai sus se identifică elementele necesare:

1. Un bloc de registre (deja discutat)
2. O unitate aritmetică-logică UAL: este necesară o unitate funcțională care să execute operația asociată instrucțiunii, unitate cu 2 operanzi de intrare și unul de ieșire. Operanzii vor fi pe 32 de biți, iar operațiile care pot fi executate de UAL sunt adunarea, scăderea, înmulțirea și împărțirea. Fiind 4 operații, rezultă



un semnal pe 2 biți pentru selecția operației. Observație: această UAL este diferită de cea construită în capitolul 5 și este descrisă doar la nivel de schemă bloc și funcționare, fără detalii interne.

Al doilea avantaj al descrierii RTL este modalitatea directă prin care se identifică legăturile dintre componente, prin analiza la diferite niveluri de detaliu (a se ține cont că I(16..0) este semnalul de ieșire din memoria ROM de instrucțiuni):

1.  $RF[I(14..10)]$  – scriere în RF, deci pozițiile 14..10 din I se vor lega la adresa de scriere a RF, *Write Address*
2.  $RF[I(9..5)]$ ,  $RF[I(4..0)]$  – citiri din RF, deci pozițiile 9..5 din I se vor lega la prima adresă de citire a RF, *Read Address 1*, iar pozițiile 4..0 la *Read Address 2*
3.  $RF[I(9..5)] \text{ op } RF[I(4..0)]$  – ieșirile *Read Data 1* și *Read Data 2* trebuie legate la cele două intrări UAL (*op* este executat efectiv de UAL)
4.  $RF[I(14..10)] \leftarrow RF[I(9..5)] \text{ op } RF[I(4..0)]$  – rezultatul produs de UAL trebuie scris în RF, deci ieșirea din UAL se va lega la intrarea de date a RF, *Write Data*.

Considerând elementele și conexiunile identificate, se assemblează partea din căile de date responsabilă pentru decodificarea și execuția instrucțiunii, respectiv scrierea rezultatului (Fig. 8.9).

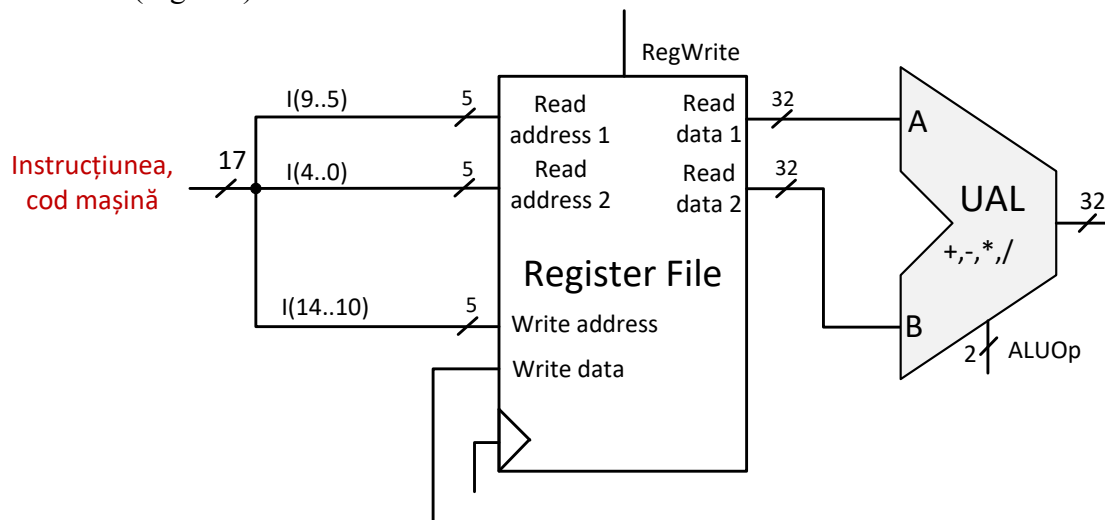


Fig. 8.9. Partea din calea de date responsabilă cu decodificarea, execuția și scrierea rezultatului pentru instrucțiunea curentă

### 8.2.3. Asamblarea completă a căii de date

Pentru a obține calea completă de date se conectează cele două părți principale obținute în pasul anterior (prezentate în Fig. 8.7 și Fig. 8.9). Unirea căilor parțiale se face în punctele (semnalele) comune pentru a asigura existența traseelor complete pentru date, necesare execuției instrucțiunilor.

Este necesară o singură conexiune între ieșirea din ROM și semnalul I(16..0) care este decodificat în partea de execuție.

Calea de date completă este prezentată în Fig. 8.10.

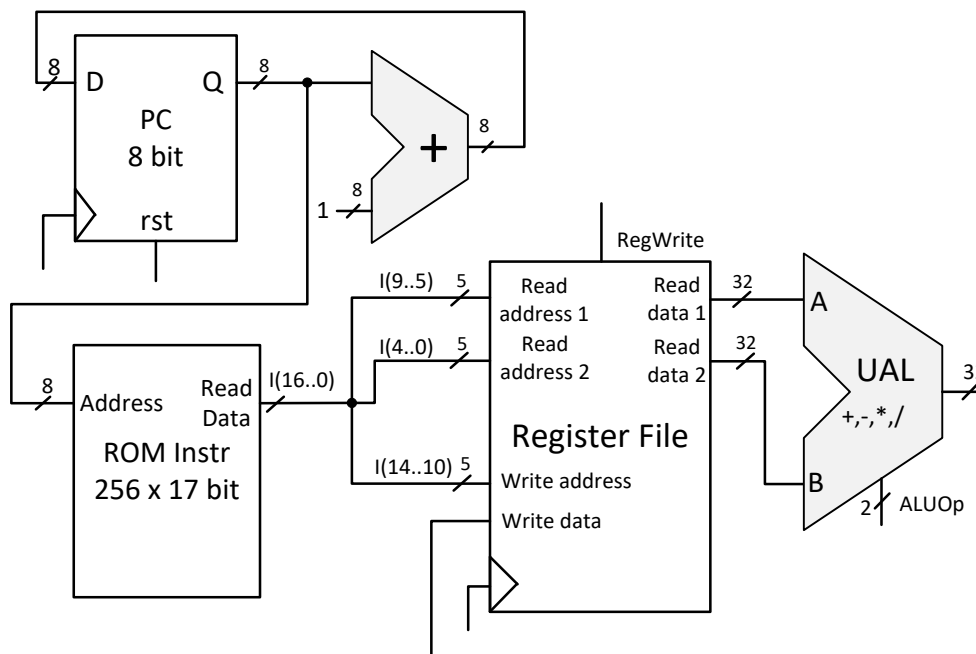


Fig. 8.10. Călea de date pentru circuitul proiectat. Pentru a nu aglomera schema cu legături triviale, nu a mai fost pus explicit semnalul de ceas pe schemă (se subînțelege că PC și RF sunt legate la același semnal de ceas pentru scrierea sincronă)

#### 8.2.4. Unitatea de control

Ultimul pas de proiectare are ca obiectiv obținerea unității de control. Rolul unității de control este de a identifica instrucțiunea, prin recunoașterea valorii codului de operație, și de a controla felul în care datele circulă prin călea de date pentru execuția corectă a instrucțiunii. Controlul căii de date se realizează prin intermediul semnalelor de control.

Fiind un circuit de tip *ciclu unic*, cu execuția instrucțiunii pe o singură perioadă de ceas, unitatea de control este combinațională. Semnalele de control depind doar de valoarea curentă a câmpului *op cod* din instrucțiune.

Mai întâi se identifică semnalele de control, cu valorile lor specifice, pentru fiecare instrucțiune. Cele 4 instrucțiuni distincte se execută în mod identic, singura diferență fiind operația executată de UAL.

Tabelul cu valorile semnalelor de control este prezentat în continuare (Tabel 8.2). Se alege o codificare convenabilă pentru *ALUOp*, pornind de la observația că atât *ALUOp*, cât și *op cod* sunt pe 2 biți, cu valori unice pentru fiecare instrucțiune: add – 00, sub 01, mul 10, div 11. Prin folosirea aceleiași codificări binare pentru *op cod* și *ALUOp*, se va simplifica mult unitatea de control. Deoarece fiecare instrucțiune execută o scriere în blocul de registre, se activează de fiecare dată scrierea prin intermediul semnalului *RegWrite*.

Tabel 8.2. Valorile semnalelor de control pentru fiecare instrucțiune

Instrucțiune	Semnale de control	
	ALUOp	RegWrite
add	00	1
sub	01	1
mul	10	1
div	11	1

La modul general, circuitul combinațional pentru unitatea de control se poate deduce folosind abordarea cu termeni canonici (subcapitolul 3.3 sau alte metode mai avansate) pornind de la tabela de adevăr, unde biții din *cod op* reprezintă variabilele de intrare, iar semnalele de control sunt variabile de ieșire.

În particular, pentru problema discutată, implementarea unității de control se poate face mai simplu. Pe *RegWrite* se pune implicit valoarea 1 (practic pentru acest circuit nu era necesar semnalul de validare a scrierii în blocul de registre), iar liniile corespunzătoare câmpului *cod op* din instrucțiune se leagă la *ALUOp* (din acest motiv s-a ales aceeași codificare binară pentru cele două câmpuri).

Schema completă a circuitului este prezentată în Fig. 8.11. Blocul Unității de Control (UC) a fost pus explicit pe schemă pentru a sublinia separarea unui circuit în calea de date și unitatea de control, chiar dacă, în acest caz, controlul se realizează într-un mod trivial: se forțează *RegWrite* pe 1 și se leagă direct *I(16..15)* la *ALUOp*.

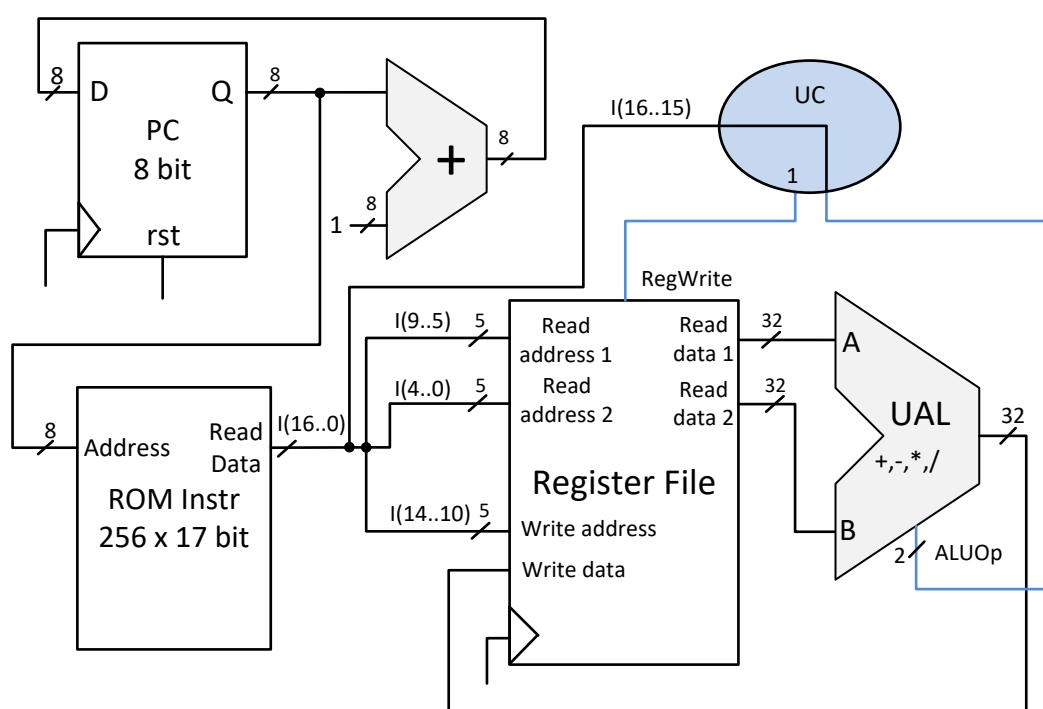


Fig. 8.11. Schema completă a circuitului, cu partea de unitate de control

Se pot discuta mai multe aspecte legate de funcționare. De exemplu, semnalul de control *rst* al PC nu este controlat de UC. Motivul este simplu, acest semnal nu este implicat în execuția efectivă a instrucțiunilor, fiind un semnal care se poate folosi pentru repornirea programului.

Un alt aspect este legat de execuția programului descris anterior, pentru calculul  $Y = (a + b * X) * a * X$ . Pornind de la prima perioadă de ceas,  $T_0$ , care ar urma după resetarea lui PC, se va executa prima instrucțiune cu scrierea rezultatului pe frontul crescător dintre  $T_0$  și  $T_1$ , apoi pe  $T_1$  a doua instrucțiune, pe  $T_2$  a treia și, în final, pe  $T_3$  a patra instrucțiune, care va scrie rezultatul  $Y$  în registrul  $r_4$  din RF. Conținutul lui  $r_4$  poate fi citit pe perioada  $T_4$  (de exemplu, de către alte circuite care ar folosi acest rezultat).

Ce se întâmplă după execuția celor patru instrucțiuni din program? Practic, pentru fiecare perioadă după  $T_4$ , inclusiv, din memoria ROM se va citi valoarea 0, pe 17 biți. Această valoare, dacă se analizează formatul de instrucțiune, reprezintă instrucțiunea *add*

$r0$ ,  $r0$ ,  $r0$ . Această instrucțiune, executată repetitiv, nu are nicio influență asupra registrelor folosite în program. Retrospectiv, acesta este motivul pentru care s-a evitat folosirea lui  $r0$  la scrierea programului. Această idee, de rezervare a registrului cu adresa 0 din RF, este de altfel o altă caracteristică a procesorului MIPS, care va fi proiectat în capitolele următoare (registrul 0 are valoarea 0 și nu poate fi scris cu alte valori). O astfel de instrucțiune, care nu are efecte în execuție, poartă numele de NoOp (eng. No Operation).

Ce se întâmplă când PC va ajunge la valoarea maximă 255 ( $11\dots11_2$ )? Răspunsul ține de aritmetica în baza 2, pe 8 biți. La următoarea perioadă de ceas se va aduna PC cu valoarea 1, toți biții din PC devenind 0 (apare un transport care se pierde, deoarece la ieșirea sumatorului pe 8 biți valoarea  $1111\ 1111_2 + 1_2 = 0000\ 0000_2$ ). Execuția programului va reîncepe de la prima instrucțiune.

În acest subcapitol soluția obținută a fost numită “circuit” chiar dacă are multe puncte comune cu un procesor. Circuitul obținut poate fi considerat un procesor cu un set foarte simplu de instrucțiuni (cu un format binar unic), care are doar un bloc de registre ca spațiu de stocare. Un pas în plus spre conceptul de procesor ar fi un set mai variat de instrucțiuni (care, de exemplu, să permită și instrucțiuni de salt pentru implementarea buclelor și a comparațiilor) și un spațiu mai mare de memorie (un bloc separat pentru memoria de date). Acesta este subiectul capitolelor următoare.

## 9. Proiectarea procesorului MIPS, varianta ciclu unic

Procesorul MIPS (Microprocessor without Interlocked Pipeline Stages) este frecvent studiat în mediul academic, fiind un procesor propus de John Hennessy și David Patterson și prezentat în seria de cărți *Computer organization and design: the hardware/software interface*. Un motiv pentru popularitatea acestei arhitecturi în mediul academic este că arhitectura sa (RISC, Reduced Instruction Set Computer) permite înțelegerea conceptelor fundamentale legate de proiectarea și funcționarea unui procesor, fără a necesita înțelegerea problemelor secundare rezultate din diverse compromisuri de proiectare. Arhitectura MIPS a fost permanent dezvoltată (MIPS Technologies, 2014), (MIPS Technologies, 2016), fiind prezentă în anumite procesoare comerciale.

În acest capitol se va prezenta și aplica procesul de proiectare pentru procesorul MIPS pe 32 de biți, versiunea cu *ciclu unic* (eng. *single cycle*), în care fiecare instrucțiune se execută într-o perioadă de ceas. Este important de menționat că se va prezenta doar proiectarea procesorului principal, fără partea de coprocesoare (pentru lucru cu numere reale sau suport al sistemului de operare). Descrierea care urmează în acest capitol este, în mare, bazată pe pașii standard și varianta clasică MIPS descrisă de Patterson și Hennessy, în care fiecare instrucțiune se execută pe o perioadă de ceas (variante de procesor cu micro arhitectura *ciclu unic*). Construirea procesorului MIPS, varianta *ciclu unic*, poate fi consultată în capitolele 2-4 din (Patterson & Hennessy, 2013) sau în capitolele 6-7, selectiv, din (Harris & Harris, 2013). Se poate consulta și varianta în limba română, dar dintr-o ediție mai veche (ediția a 2-a), în capitolele 3-5 din (Patterson & Hennessy, 2002).

Complementar abordării din literatură, pentru construirea procesorului MIPS se va pune accent pe metoda de proiectare bazată pe transferul dintre registre (RTL). Se explică în detaliu cum trebuie analizat transferul RTL pentru a identifica atât componentele din procesor, cât și conexiunile dintre ele. Această abordare, odată înțeleasă, este utilă și pentru cititorii cu mai puțină experiență în domeniu.

Câmpurile relevante din instrucțiuni și denumirile de pe diagrame vor fi în limba engleză, cu acronime similare cu cele din literatură. Se asigură astfel o uniformitate în notații și acronime cu literatura, tocmai pentru a ușura studiul literaturii de specialitate, pentru cititorii acestei cărți care doresc mai multe informații.

### 9.1. Arhitectura setului de instrucțiuni – aspecte generale

Proiectarea unui procesor începe de la setul de instrucțiuni sau, mai exact, de la Arhitectura Setului de Instrucțiuni (ASI, eng. Instruction Set Architecture – ISA). Arhitectura setului de instrucțiuni reprezintă o colecție completă de specificații care permite scrierea de programe în cod mașină pentru procesor. ISA definește, de obicei, setul de instrucțiuni, cu formatul binar asociat, tipul operanzilor permiși în instrucțiune, registre, spațiul de memorie care poate fi accesat/adresat, modurile de adresare etc. ISA reprezintă un mod abstract de a descrie procesorul (nu se referă la implementarea fizică), același set de instrucțiuni putând avea mai multe implementări fizice (procesoare produse de companii diferite, cu implementări particulare, dar același ISA).

Există mai multe criterii pentru clasificarea ISA-urilor, dar probabil cel mai relevant este cel legat de complexitatea arhitecturii. Potrivit acestui criteriu, se pot identifica două clase de ISA-uri: RISC (Reduced Instruction Set Computer) și CISC (Complex Instruction Set Computer).

Arhitecturile RISC (ex. MIPS, ARM) se bazează pe o complexitate de implementare (hardware) redusă, punând accent pe instrucțiunile de bază folosite frecvent. Astfel, se mută efortul pentru construirea de programe complexe spre compilator sau programator, dar simplitatea de implementare permite optimizări de tip pipeline (linie de asamblare). Acest tip de arhitectură are următoarele caracteristici:

- accesul la memorie se face prin instrucțiuni dedicate de tip load/store care citesc/salvează date din/în memorie
- procesarea datelor se face folosind doar registre, din acest motiv existând un număr mai mare de registre
- număr redus de moduri de adresare (ce și cum se poate adresa folosind câmpurile din formatul de instrucțiune)
- instrucțiunile au un număr redus de formate, cu dimensiune fixă (ex. la MIPS, toate instrucțiunile sunt pe 32 de biți – 4 octeți).

Arhitecturile CISC (ex. Intel x86) se bazează pe o complexitate crescută în hardware, rezultată din varietatea (cu diferite grade de complexitate) de instrucțiuni suportate. Se pot scrie programe mai scurte, dar complexitatea hardware crescută limitează posibilitățile de optimizare:

- accesul la memorie se face prin multe instrucțiuni, inclusiv cele de procesare, fiind permise operații directe de calcul între registre și locații din memorie
- număr crescut de moduri de adresare (ce și cum se poate adresa folosind câmpurile din formatul de instrucțiune)
- instrucțiunile au multe formate, de dimensiune variabilă (ex. la Intel x86 formatul de instrucțiune variază între 1 și 15 octeți).

Procesoarele moderne, chiar dacă mențin o arhitectură vizibilă a setului de instrucțiuni de tip CISC, intern au părți semnificative bazate pe principiile RISC.

## 9.2. Arhitectura setului de instrucțiuni – studiu de caz MIPS 32

Procesorul MIPS pe 32 de biți are o ISA cu 3 adrese (în formatul de instrucțiune pot fi reprezentate până la 3 adrese de operanzi), bazată pe registre de uz general (MIPS Technologies, MIPS® Architecture For Programmers, Volume I-A: Introduction to the MIPS32® Architecture, Revision 6.01, 2014). Este o arhitectură de tip Load/Store, cu instrucțiuni dedicate pentru accesarea memoriei. În continuare se prezintă principalele aspecte ale ISA MIPS relevante pentru procesul de proiectare care va urma (prezentate doar din perspectiva procesorului MIPS care urmează a fi construit).

### 9.2.1. Registre

Procesorul MIPS are 32 de registre de uz general grupate într-un bloc de registre, vizibile programatorului. Registrele sunt identificate în mod unic de indexul pe care îl au în bloc: 0, 1, ..., 31. În sintaxa instrucțiunilor, pentru a sublinia diferența dintre adresa și conținutul unui registru, se va folosi prefixul \$ pentru referirea conținutului. \$0, \$1, ..., \$31 se referă la datele pe 32 de biți memorate de fiecare registru.

Registrul de la adresa 0 este tot timpul 0 (valoarea \$0 este 0). Este permisă scrierea în registrul 0, dar valoarea acestuia nu se modifică. Acest registru se poate folosi ca valoare de referință pentru 0, atunci când se fac comparații sau se dorește inițializarea altor registre cu valori constante (immediate).

Un alt registru important este registrul contor de program, Program Counter – PC, pe 32 de biți, care indică adresa instrucțiunii curente.

### 9.2.2. Organizarea memoriei

Memoria este cu adresare pe octet,  $2^{32}$  octeți = 4 GB, organizată pe cuvânt de 32 de biți. Cuvintele din memorie sunt aliniate la adrese multiple de 4 octeți, la fel ca modelul prezentat în figura Fig. 7.2. Elementele procesate din memorie vor fi cuvinte de 32 de biți. În setul complet de instrucțiuni MIPS se permite accesarea individuală a octeților / semi-cuvintelor, dar acest aspect nu va fi dezvoltat în această carte.

### 9.2.3. Tipuri de date / operanzi

Se va folosi elementul standard de date, cuvântul pe 32 biți, care poate fi stocat în registre de uz general sau memorie. Acesta va avea rol de operand pentru diferite operații de procesare.

### 9.2.4. Tipuri de instrucțiuni

Sunt definite 3 tipuri de instrucțiuni: **tip R**, **tip I** și **tip J**. Instrucțiunile de același tip au aceleași câmpuri în sintaxă și în format.

Instrucțiunile de **tip R** sunt instrucțiuni cu operanzi de tip registru (de unde se iau datele și unde se salvează rezultatul), care au următoarea sintaxă generală:

**nume\_instr   \$rd, \$rs, \$rt**

Aceste instrucțiuni sunt folosite, în special, pentru operații aritmetice / logice, cu operanzi de tip registru. Se face o anumită operație între două registre sursă (cu indecșii rs și rt) din blocul de registre, și se salvează rezultatul în registrul destinație, cu indexul rd, din blocul de registre.

Instrucțiunile de **tip I** conțin o valoare imediată (semi-cuvânt pe 16 biți) în format, fiind de 3 subtipuri (toate au aceleași câmpuri în sintaxă). Prima variantă de instrucțiuni (subtip 1) de **tip I** este cea pentru operații aritmetice / logice cu valoare imediată (constantă numerică folosită în codul program):

subtip 1: **nume\_instr   \$rt, \$rs, imm**   sau   subtip 2: **nume\_instr   \$rs, \$rt, imm**

Se execută o operație între registrul sursă cu indexul rs și valoarea imediată imm, iar rezultatul este salvat în registrul reprezentat de rt. Al doilea subtip are sintaxă similară, dar sunt instrucțiuni de salt condiționat: se compară valorile din operanzii de tip registru, \$rs și \$rt, și, în funcție de rezultatul comparației, se execută un salt relativ în program. Destinația saltului se calculează în funcție de valoarea imediată imm.

Al treilea subtip din **tipul I** este pentru instrucțiunile de lucru cu memoria de date. Sintaxa este un pic diferită, dar sunt aceleași câmpuri (offset este echivalent cu imm):

subtip 3: **nume\_instr   \$rt, offset(\$rs)**

Pentru a accesa memoria, adresa se va forma din valoarea offset și \$rs, iar \$rt va reprezenta valoarea care trebuie salvată în memorie (la scriere), respectiv unde se va salva valoarea citită din memorie.

Instrucțiunile de **tip J** sunt folosite pentru salturi necondiționate la adresa indicată de câmpul `target_address`, având următoarea sintaxă generală:

**nume\_instr target\_address**

Toate instrucțiunile au un format binar pe 32 de biți, prezentat în Fig. 9.1 pentru fiecare tip.

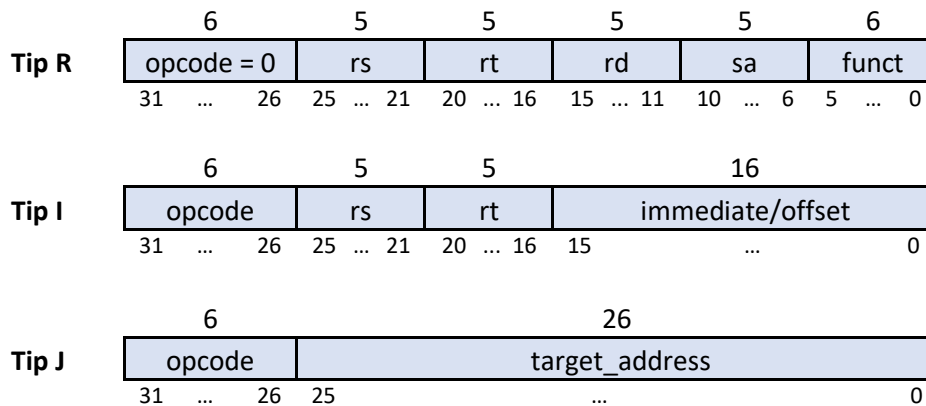


Fig. 9.1. Tipuri de instrucțiuni suportate de procesorul MIPS. Sunt prezentate câmpurile, cu denumirile folosite standard în literatură, și dimensiunile lor în biți (sus). Pozițiile din stânga sunt cele mai semnificative (începând cu bitul 31)

În continuare, se oferă o explicație logică pentru formatul binar al celor 3 tipuri de instrucțiuni: cum s-a ajuns la dimensiunea în biți a fiecărui câmp? Grupurile de biți din format trebuie să codifice, sub formă de valori binare, câmpurile prezente în sintaxa instrucțiunilor. Dimensiunile grupurilor de biți din format pot fi deduse pornind de la sintaxa generală pentru fiecare tip.

Pentru formatul de **tip I** se aplică următorul raționament, pornind de la cele 4 câmpuri care trebuie codificate în binar: codul instrucțiunii, 2 adrese pentru operanzii de tip registru și o valoare imediată (constantă care se include în codificarea binară a instrucțiunii). Cele două câmpuri `rs` și `rt`, reprezentând adrese din blocul de registre, necesită câte 5 biți fiecare. Rămân  $32 - 10 = 22$  de biți disponibili. Imediatul `imm`, conform specificațiilor, va fi codificat pe 16 biți. Rămân 6 biți disponibili pentru codificarea efectivă a instrucțiunii, câmp care va fi denumit în continuare cod de operație, prescurtat `opcode` (eng. operation code).

Mergând pe principiul de favorizare a uniformității, câmpurile care apar la mai multe tipuri se vor codifica la fel și pe aceleași poziții. Astfel, și la celelalte tipuri de instrucțiuni se va folosi același câmp pe 6 biți pentru codificarea instrucțiunii.

Formatul de **tip J** va conține câmpul pe 6 biți reprezentând `opcode`, iar restul de 26 de biți se vor aloca pentru a codifica adresa unde se dorește saltul necondiționat.

Formatul de **tip R** va conține următoarele câmpuri: `opcode` pe 6 biți, 3 câmpuri de câte 5 biți fiecare pentru a codifica adresa operanzilor de tip registru `rs`, `rt` și `rd`. Rămân 11 biți disponibili. O categorie aparte de instrucțiuni de tip R sunt cele de deplasare pe biți (nu se vor folosi în descrierea din această carte, dar sunt menționate pentru a justifica formatul binar). Datele care se deplasează sunt cele dintr-un registru. Fiind 32 de biți, deplasarea se poate face între 0 (rămâne neschimbat) și 31 de poziții. Deplasarea cu 32 de poziții nu are utilitate deoarece ar dispărea toți biții originali din registru. Cantitatea de deplasare sa (eng.



shift amount) necesită 5 biți pentru codificare, din cei 11 rămași. Rămân 6 biți liberi. Legat de acești biți nefolosiți încă, urmează o explicație care va lămuri singurul aspect mai puțin evident legat de instrucțiunile de tip R. Câmpul opcode, fiind pe 6 biți, ar permite definirea a  $2^6=64$  de instrucțiuni distincte. Deoarece în formatul de **tip R** există 6 biți liberi, se poate mări semnificativ numărul de instrucțiuni care se pot reprezenta, în felul următor: pentru instrucțiunile de tip R se folosește o valoare unică pentru opcode, și anume valoarea 0, iar operația efectivă este codificată prin cei 6 biți rămași, grupați în câmpul numit funct (din termenul function, în engleză). Astfel, se pot defini 63 de instrucțiuni distincte care nu sunt de **tip R**, și încă  $2^6=64$  de instrucțiuni de **tip R**.

În concluzie, semnificația câmpurilor din format este următoarea:

- opcode este codul de operație principal pe 6 biți, la tip R = 0
- rs este indexul primului registru sursă, 5 biți
- rt este indexul celui de al doilea registru sursă / indexul registrului destinație, 5 biți
- rd este indexul registrului destinație, 5 biți
- sa este cantitatea de deplasare
- funct este câmpul care indică operația efectivă pentru instrucțiuni de tip R, 6 biți
- immediate / offset:
  - imediat/constantă pe 16 biți pentru operații aritmetice-logice
  - deplasamentul cu semn, în octeți, pentru operații cu memoria
  - deplasamentul cu semn, relativ la PC, în instrucțiuni, pentru instrucțiuni de salt condiționat
- target address este un număr pe 26 biți din care se va calcula adresa de salt necondiționat.

### 9.2.5. Instrucțiuni reprezentative

În setul de instrucțiuni MIPS, conform formatului binar descris anterior, se pot defini 63 de instrucțiuni de tip I sau J (opcode permite codificarea a 64 de coduri individuale, iar codul 0 este rezervat pentru Tip R). În plus, pentru formatul Tip R, câmpul funct permite definirea a încă 64 de instrucțiuni. Proiectarea procesorului, de la început, pentru setul complet de instrucțiuni, este o sarcină foarte complexă. O astfel de abordare, prin volumul foarte mare de detalii, ar ascunde parțial principiile de proiectare. Setul complet de instrucțiuni pentru procesorul MIPS 32 poate fi consultat în (MIPS Technologies, MIPS® Architecture for Programmers, Volume II-A: The MIPS32® Instruction Set Manual, Revision 6.05, 2016).

O abordare realistă, care este mai ușor de aplicat, urmărit și înțeles, este proiectarea procesorului pornind de la un set restrâns de instrucțiuni, după care se poate extinde iterativ cu toate instrucțiunile.

În această secțiune se va prezenta subsetul de instrucțiuni MIPS pentru care se va proiecta procesorul. Acest subset este asemănător cu cel folosit în (Patterson & Hennessy, 2013). Se vor selecta instrucțiuni reprezentative din fiecare tip.

Pentru fiecare instrucțiune, pe lângă sintaxă și funcționare, se va prezenta și formatul binar. Pentru formatul binar se vor stabili valori unice pentru câmpurile opcode și funct. Stabilirea acestor valori ține de proiectant, important fiind să se respecte codificarea aleasă atunci când se va proiecta unitatea de control (la finalul capitolului). Pentru a fi în acord cu literatura de specialitate, în descrierea care urmează se vor folosi

valorile standard pentru codurile de operație, din setul de instrucțiuni (MIPS Technologies, 2016).

Pentru tipul R se vor folosi instrucțiunile *add* (adunare), *sub* (scădere), *or* (SAU logic) și *and* (ȘI logic). Sintaxa pentru aceste instrucțiuni este prezentată mai jos. Toate execută o operație între cele două registre sursă, *rs* și *rt*, și salvează rezultatul în registrul destinație *rd*:

```
add $rd, $rs, $rt
sub $rd, $rs, $rt
or  $rd, $rs, $rt
and $rd, $rs, $rt
```

Formatul binar (Tip R, cu codurile unice de operație în câmpul *funct*, și *sa*=0 nefiind instrucțiuni de deplasare) este:

	op code	rs	rt	rd	sa	funct
<b>add</b>	000000	rs	rt	rd	00000	100000 = add
<b>sub</b>	000000	rs	rt	rd	00000	100010 = sub
<b>or</b>	000000	rs	rt	rd	00000	100101 = or
<b>and</b>	000000	rs	rt	rd	00000	100100 = and

Atunci când se scrie un program, fiecare instrucțiune se convertește în cod mașină folosind formatul definit. Pe câmpurile *opcode* și *funct* se pun codurile instrucțiunii, iar operanzii se codifică pe câmpurile corespunzătoare. Pentru instrucțiunile de tip R, fiecare registru (indexul lui) se reprezintă pe 5 biți în câmpul asociat din formatul de instrucțiune.

Exemple de codificare:

```
add $1, $2, $3      cod mașină: 000000 00010 00011 00001 00000 100000
and $9, $2, $3      cod mașină: 000000 00010 00011 01001 00000 100100
```

Înainte de discutarea instrucțiunilor de tip I, trebuie explicată ideea de extindere a unui semnal pe mai mulți biți. În formatul de tip I apare câmpul *imm* pe 16 biți. Călea de date a procesorului va fi pe 32 de biți (dimensiunea operanzilor). *Imm* va trebui extins pe 32 de biți în calea de date pentru a putea fi folosit în operații cu alți operanzi. În funcție de instrucțiune (operație logică pe biți sau aritmetică), semnalul *imm* se va extinde (la stânga) cu zero (fără semn) sau cu bitul de semn.

Extinderea cu zero (fără semn) se face prin concatenarea de biți de 0 în partea stângă, pe pozițiile semnificative – astfel valoarea *imm* are aceeași semnificație ca magnitudine, dar reprezentată pe mai mulți biți. Extinderea cu semn se face prin duplicarea bitului de semn pe câte poziții este nevoie (reprezentarea în complement față de 2 asigură obținerea aceleiași valori, dar cu o reprezentare pe mai mulți biți). Exemple de extindere cu / fără semn:

imm <sub>10</sub>	imm <sub>2</sub> (16 biți)	imm <sub>2</sub> (32 biți)
<b>Extindere fără semn (cu 0)</b>		
65535	1111111111111111	00..00 1111111111111111
<b>Extindere cu semn</b>		
-1	1111111111111111	11..11 1111111111111111

24576	0110000000000000	00..00 0110000000000000
-------	------------------	-------------------------

Pentru tipul I se vor selecta mai multe subcategorii de instrucțiuni. Prima subcategorie este cea a instrucțiunilor aritmetice/logice (subtip 1). Se vor alege instrucțiunile *ori* (sau logic cu imediat/constantă) și *addi* (adunare cu imediat/constantă):

*ori* \$rt, \$rs, imm – sau logic între \$rs și valoarea imm (extinsă cu 0 pe 32 biți), salvează rezultatul în \$rt

Format binar *ori*:

	op code			
<b>ori</b>	001101	rs	rt	imm

*addi* \$rt, \$rs, imm – adunare între \$rs și valoarea imm (extinsă cu semn pe 32 biți), salvează rezultatul în \$rt

Format binar *addi*:

	op code			
<b>addi</b>	001000	rs	rt	imm

Exemple de codificare:

*ori* \$11, \$11, 2                      cod mașină: 001101 01011 01011 0000000000000010  
*addi* \$5, \$3, -2                      cod mașină: 001000 00011 00101 1111111111111110

A doua subcategorie este reprezentată de instrucțiunile de lucru cu memoria (subtip 3): *lw* (load word – încarcă cuvânt din memorie într-un registru) și *sw* (store word – stochează cuvântul dintr-un registru în memorie). Sintaxa este prezentată în continuare:

*lw* \$rt, offset(\$rs) – încarcă în \$rt cuvântul aflat în memorie la adresa efectivă (\$rs+offset)

*sw* \$rt, offset(\$rs) – stochează cuvântul din \$rt în memorie la adresa efectivă (\$rs+offset)

Format binar *lw/sw* (Tip I):

	op code			
<b>lw</b>	100011	rs	rt	offset
<b>sw</b>	101011	rs	rt	offset

Exemple de codificare:

*lw* \$3, -1(\$8)                      cod mașină: 100011 01000 00011 1111111111111111  
*lw* \$31, 254(\$8)                      cod mașină: 100011 01000 11111 0000000011111110  
*sw* \$7, 30(\$5)                      cod mașină: 101011 00101 00111 0000000000001110  
*sw* \$12, -11(\$2)                      cod mașină: 101011 00010 01100 11111111111110101

Ultima subcategorie din tip I este cea a instrucțiunilor de salt condiționat (subtip 2). Aceste instrucțiuni permit implementarea buclelor de program (for, do/while) sau comparațiilor (if-then-else). Se va alege cea mai reprezentativă instrucțiune, și anume *beq* (branch on equal, execută un salt relativ în program dacă două registre sunt egale):

*beq* \$rs, \$rt, imm – dacă \$rs== \$rt, atunci execută un salt peste imm instrucțiuni față de instrucțiunea imediat următoare după *beq* în program

	op code			
<b>beq</b>	000100	rs	rt	imm

Exemple de codificare:

*beq* \$2, \$14, 100      cod mașină: 000100 00010 01110 0000000001100100  
*beq* \$4, \$15, -11      cod mașină: 000100 00100 01111 111111111110101

Pentru tipul J se va selecta instrucțiunea *j* (jump, salt necondiționat):

*j* target\_addr – salt necondiționat la o adresă pseudo-absolută

<b>j</b>	000010	target_addr
----------	--------	-------------

Exemplu de codificare:

*j* 257      cod mașină: 000010 00000000000000000100000001

### 9.2.6. Etape de execuție

Pentru a ușura procesul de proiectare, se va ține cont de etapele tipice de execuție pentru o instrucțiune (în cazul procesorului MIPS cu ciclu unic aceste etape se desfășoară pe aceeași perioadă de ceas!). Aceste etape sunt construite din perspectiva fluxului de date, la execuția unei instrucțiuni, prin calea de date a procesorului. Se vor folosi acronimele specifice din engleză, întâlnite în literatura de specialitate:

1. Extragerea / aducerea instrucțiunii din memorie: Instruction Fetch – IF
2. Interpretarea/decodificarea instrucțiunii și citirea operanzilor: Instruction Decode – ID
3. Execuția efectivă a instrucțiunii: Execute – EX
4. Operații cu memoria: Memory – MEM
5. Scrierea rezultatului (înapoi în blocul de registre): Write Back – WB.

## 9.3. Metodologia de proiectare bazată pe RTL

Proiectarea procesorului se bazează pe descrierea transferului de date la nivel de registre (eng. Register Transfer Level – RTL, deja introdus în capitolul anterior, secțiunea 8.2.2).

RTL este o descriere a procesorului din perspectiva fluxului de date dintre elementele de stocare (registre, memorii). Prin RTL se descrie formal execuția fiecărei instrucțiuni (sau grup de instrucțiuni similare) ca un transfer, cu eventuale

transformări/calculare, a datelor din elementele de stocare sursă în elementul destinație. Se mai pot descrie condiții în cazul instrucțiunilor condiționate de anumite comparații. Descrierea RTL nu are neapărat o notație impusă, fiind similară cu un pseudocod simplu (se va prezenta sintaxa pe parcursul procesului de proiectare, în primul pas).

Pentru a proiecta procesorul, pornind de la setul de instrucțiuni, se parcurg următorii pași:

1. Se analizează setul de instrucțiuni la nivel RTL, rezultând cerințe pentru calea de date (componente principale, cu funcționalitate)
2. Se selectează setul de componente, ținând cont și de politica semnalului de ceas (o instrucțiune / perioadă de ceas, pentru cazul discutat)
3. Se assemblează calea de date conform cerințelor
4. Se identifică și se definesc semnalele de control pentru componentele din calea de date
5. Se construiește unitatea de control.

### 9.3.1. Analiza setului de instrucțiuni

Din acest pas de analiză trebuie să rezulte componentele principale necesare în calea de date, fără a ține neapărat cont de disciplina semnalului de ceas. Aceste transferuri sunt deduse din ceea ce trebuie să facă fiecare instrucțiune și arată ce se întâmplă în procesor la execuția unei instrucțiuni. Astfel, nu se vor specifica toate detaliile fiecărei componente (acesta va fi pasul următor).

Transferurile RTL de la fiecare instrucțiune sunt prezentate în Tabel 9.1 (va fi consultat permanent de-a lungul pașilor de proiectare). RF reprezintă blocul de registre, iar M blocul de memorie (unde sunt stocate datele).

Tabel 9.1. Transferurile RTL pentru instrucțiunile selectate

Instrucțiune	RTL	PC – modificare
op \$rd, \$rs, \$rt	$RF[rd] \leftarrow RF[rs] \text{ op } RF[rt]$ , op poate fi add, sub, or, and	$PC \leftarrow PC + 4$
ori \$rt, \$rs, imm	$RF[rt] \leftarrow RF[rs] \text{ or } Z\_Ext(imm)$	$PC \leftarrow PC + 4$
addi \$rt, \$rs, imm	$RF[rt] \leftarrow RF[rs] + S\_Ext(imm)$	$PC \leftarrow PC + 4$
lw \$rt, imm(\$rs)	$RF[rt] \leftarrow M[RF[rs] + S\_Ext(imm)]$	$PC \leftarrow PC + 4$
sw \$rt, imm(\$rs)	$M[RF[rs] + S\_Ext(imm)] \leftarrow RF[rt]$	$PC \leftarrow PC + 4$
beq \$rs, \$rt, imm	If( $RF[rs] == RF[rt]$ ) then	$PC \leftarrow PC + 4 + S\_Ext(imm) \ll 2$
	else	$PC \leftarrow PC + 4$
j target_addr	$PC \leftarrow (PC+4)[31:28] \& target\_addr \& 00$	

În tabelul anterior sunt prezentate transferuri RTL. Operatorii folosiți sunt:

- $\leftarrow$  este atribuire
- $\ll$  deplasare pe biți
- $==$  comparație de egalitate
- $S\_ext(x)$  extinde cu semn semnalul x de la 16 la 32 de biți, specific operațiilor aritmetice
- $Z\_ext(x)$  extinde cu biți de 0 semnalul x de la 16 la 32 de biți, specific operațiilor logice pe biți
- $\&$  realizează concatenarea dintre semnale.

Pentru fiecare instrucțiune este necesară calcularea noii valori a lui PC (registru contor de program), cu scopul de a continua execuția programului în ordinea dorită. În mod normal, se va trece la următoarea instrucțiune din program ( $PC + 4$ ), excepție făcând instrucțiunile de salt. Se adună 4 la PC deoarece o instrucțiune are 4 octeți, iar memoria unde sunt stocate instrucțiunile este adresabilă pe octet!

Procesul de proiectare, pentru acest pas, se va ghida după etapele de execuție a instrucțiunilor.

Pentru primele două etape, de extragere (IF) și interpretare (ID), procesorul execută sarcini aproape identice. Pentru restul, la fiecare etapă se va analiza RTL-ul fiecărei instrucțiuni pentru a identifica progresiv componentele necesare.

#### 9.3.1.1. Extragerea instrucțiunii - IF

Sunt necesare următoarele componente (nu se repetă în detaliu raționamentul, dar este aproape identic cu cel descris la începutul secțiunii 8.2.2 din capitolul 8):

1. Memoria (unde se memorează instrucțiuni), 32 de linii de adresă (adresabilă pe octet), organizată pe cuvânt de 4 octeți, fiecare instrucțiune se va afla la adrese multiple de 4 octeți
2. Registrul contor de program PC, pe 32 de biți
3. Sumator, pe 32 de biți, pentru calculul adresei următoarei instrucțiuni ( $PC + 4$ ), două intrări/o ieșire.

#### 9.3.1.2. Interpretarea instrucțiunii - ID

Fiind etapa de interpretare a codului binar al instrucțiunii, respectiv citirea operanzilor, în etapa de ID sunt necesare următoarele componente:

1. Blocul de registre RF
2. Unitate de extensie cu sau fără semn (pentru extinderea pe 32 biți a *imm*)
3. Unitatea de control (la acest subiect se va reveni spre finalul procesului de proiectare).

#### 9.3.1.3. Execuția instrucțiunii - EX

În această etapă se analizează instrucțiunile individual (sau pe grupuri de instrucțiuni cu execuție similară).

Pentru instrucțiunile de tip R este necesară o unitate aritmetică-logică ALU (se va folosi acronimul din engleză), pe 32 de biți, care să poată executa operații de adunare, scădere, SAU și ȘI logic pe biți. Se va folosi ALU construită (convenabil) în capitolul 5.

Pentru instrucțiunile *ori* și *addi* se va folosi aceeași ALU pentru a executa SAU logic, respectiv adunare, între registrul selectat și imediatul extins fără semn / cu semn.

În cazul instrucțiunilor de lucru cu memoria, este necesară calcularea adresei efective a locației de memorie. În RTL-ul instrucțiunilor *lw/sw*, acest calcul este reprezentat de operația de adunare care va fi executată tot de ALU.

Instrucțiunea de salt condiționat *beq* necesită o analiză în detaliu. În RTL-ul instrucțiunii există două părți care necesită calcule: evaluarea condiției și calcularea adresei de salt. Prima este condiția de evaluat  $RF[rs] == RF[rt]$ . În mod convențional, condițiile de salt în procesoare sunt evaluate de unitatea ALU. Unitatea ALU construită anterior are un semnal de stare denumit *Zero* prin care se semnalează prezența unui rezultat nul pe ieșirea ALU. Evaluarea condiției de egalitate se poate face printr-o operație de scădere între cele

două registre. Rezultatul este nul în caz de egalitate și, implicit, se va activa semnalul de stare *Zero*. Pentru calculul adresei de salt se va folosi sumatorul menționat deja pentru IF ( $PC + 4$ ), dar mai este necesară o adunare cu deplasamentul în octeți  $S\_Ext(imm) \ll 2$ . Se adaugă un al doilea sumator pe 32 de biți și un circuit de deplasare la stânga cu 2 poziții.

Ultima instrucțiune de analizat este cea de salt necondiționat  $j$ . În mod normal, sunt necesari 32 de biți pentru a reprezenta adresa instrucțiunii țintă, dar formatul de instrucțiune permite doar 26 de biți pentru adresă (denumită și pseudo-adresă). Este necesară o soluție de compromis. Pentru a obține adresa pe 32 de biți se concatenează cei mai semnificativi 4 biți din adresa următoarei instrucțiuni după  $j$ ,  $(PC+4)[31..28]$ , cu cei 26 de biți și cu 2 biți de 0.

Extinderea cu 2 biți de zero este ușor de justificat, fiind echivalentă cu înmulțirea cu 4 (instrucțiunile încep la adrese multiple de 4 octeți). Pentru a explica de ce se folosesc cei 4 biți din  $PC+4$  și care este efectul, trebuie rediscutat conceptul de adresare a memoriei. Dacă se folosește o adresă de 32 de biți, bitul cel mai semnificativ va împărți spațiul de adresare în două zone egale, următorul bit va împărți fiecare dintre cele 2 zone în alte două zone egale etc. Cei mai semnificativi 4 biți din PC împart spațiul de adresare în  $2^4 = 16$  zone egale (Fig. 9.2), fiecare zonă având  $4\text{ GB} / 16 = 256\text{ MB}$ . Concluzia este că instrucțiunea  $j$  permite, de fapt, un salt absolut în interiorul zonei curente de 256 MB unde se află secțiunea de program.

Din RTL-ul instrucțiunii  $j$ , operația  $target\_addr \& 00$  se va alocă pe o componentă de tip circuit de deplasare la stânga cu 2 poziții (care va și extinde semnalul pe 28 de biți), iar concatenarea cu  $(PC+4)[31..28]$  se va realiza prin alăturarea traseelor când se construiește calea de date (semnalele aferente pozițiilor).

PC / Adresa (32 biți)	ROM (instrucțiuni)
0000 0000...0000	256 MB
0000 0000...0001	
0000 0000...0010	
...	
0000 1111...1111	
0001 0000...0000	256 MB
0001 0000...0001	
0001 0000...0010	
...	
0001 1111...1111	
...	...
1111 0000...0000	256 MB
1111 0000...0001	
1111 0000...0010	
...	
1111 1111...1111	

Fig. 9.2. Împărțirea unui spațiu de memorie de 4 GB după primii 4 biți din adresă: rezultă 16 zone de 256 MB

În concluzie, au rezultat următoarele componente (în plus față de etapele anterioare):

1. O ALU pe 32 biți, două intrări/o ieșire, cu operațiile adunare, scădere, SAU/ȘI logic
2. Sumator, pe 32 de biți, pentru calculul adresei de salt pentru beq, două intrări/o ieșire
3. Două circuite de deplasare la stânga (pentru beq, respectiv j).

#### 9.3.1.4. Operații cu memoria - MEM

În cazul instrucțiunilor de lucru cu memoria, este necesar (evident) blocul de memorie. Este important de subliniat că, în această etapă de analiză, nu rezultă necesitatea folosirii a două blocuri distincte de memorie, pentru instrucțiuni, respectiv date. Memoria poate fi comună, reprezentată de un singur bloc. Această problemă se va lămurii la pasul următor, când se va ține cont de constrângerea de *ciclu unic*.

#### 9.3.1.5. Scrierea rezultatului – WB

Rezultatul se scrie înapoi în blocul de registre RF, prin portul de scriere.

### 9.3.2. Setul de componente

În pasul anterior nu a fost folosită constrângerea ca fiecare instrucțiune să se execute pe o perioadă de ceas. În acest pas, analiza este dusă mai departe incluzând această constrângere. Scrierile în elementele de memorare vor avea loc pe front crescător de ceas.

Pentru fiecare componentă dedusă anterior se vor specifica în detaliu caracteristicile. Blocurile de memorie ROM/RAM, respectiv blocul de registre RF au fost deja prezentate în capitolele anterioare, având caracteristicile necesare pentru a fi folosite în procesorul proiectat în acest capitol. Totuși, în această secțiune se va explica pe scurt, cum, din analiza RTL și constrângerile de ceas, se pot deduce proprietățile acestor blocuri.

Stocarea instrucțiunilor și a datelor se va face în memorie. În cazul instrucțiunilor care lucrează cu memoria (*lw/sw*), pe durata unui ciclu de ceas trebuie:

1. accesată memoria la adresa instrucțiunii curente pentru citirea instrucțiunii. Aceasta trebuie să rămână stabilă pe ieșirea memoriei, pe întreaga perioadă de ceas. Implicit, și valoarea adresei trebuie să fie constantă.
2. accesată memoria la adresa unde se va executa scrierea (*sw*) sau citirea (*lw*) și menținută stabilă valoarea adresei pe durata perioadei de ceas.

Pentru fiecare accesare, trebuie generată adresa dorită pe liniile de adresă ale circuitului de memorie. Dacă se folosește un singur circuit de memorie cu o singură intrare de adresă, nu se pot furniza două valori diferite pe intrarea de adresă, pe aceeași perioadă de ceas. Pentru a executa ambele accese, pe aceeași perioadă de ceas, vor fi necesare memorii separate pentru instrucțiuni și date.

Se va folosi o memorie ROM pentru a stoca programele, cu citire asincronă, și o memorie RAM pentru date, cu citire asincronă și scriere sincronă cu semnal de validare a scrierii (conform descrierii ROM / RAM din capitolul 7).

Din RTL-ul instrucțiunilor de tip R se observă că blocul de registre RF trebuie să permită o scriere și două citiri. Dacă aceste operații trebuie executate pe aceeași perioadă de ceas, rezultă că RF trebuie să permită simultan două citiri asincrone (pentru ca datele din registrele sursă să fie disponibile la ieșirea RF pe durata perioadei de ceas) și o scriere sincronă. Deoarece RF conține 32 de registre, liniile de adresă pentru RF trebuie să fie pe



5 biți. Nu toate instrucțiunile necesită scriere în RF, deci scrierea trebuie validată cu un semnal de control. Blocul RF prezentat în capitolul 7 îndeplinește aceste constrângeri.

Din analiza făcută până acum au fost identificate următoarele componente (scheme bloc în Fig. 9.3) cu proprietățile enumerate:

1. PC contorul de program:
  - registru 32 de biți, scriere pe front crescător
2. Memoria ROM de instrucțiuni:
  - adresă pe 32 de biți
  - ieșirea de date *instr* pe 32 de biți reprezintă instrucțiunea curentă
  - citirea combinațională
3. Memoria RAM de date:
  - adresă pe 32 de biți
  - ieșirea de date (*Read data*) pe 32 de biți, intrarea de date (*Write data*) pe 32 de biți
  - citirea combinațională
  - scrierea sincronă pe front crescător, cu validarea scrierii prin semnalul *MemWrite* (=1 pentru executarea scrierii, doar la sw)
4. Bloc de registre RF, 32 x 32 biți:
  - Se pot selecta la un moment dat (pe aceeași perioadă de ceas) 3 registre: 2 pentru citire asincronă, unul pentru scriere sincronă
  - *Read address 1, 2* sunt adresele celor 2 registre care se citesc, conținutul apare pe *Read data 1, 2*
  - *Write address* este adresa registrului în care se scrie valoarea de pe intrarea de date *Write data*, atunci când apare un front crescător de ceas și semnalul de validare *RegWrite* este 1 (pentru a efectua scrierea doar atunci când este necesar pentru instrucțiunea curentă)
5. Două sumatoare pe 32 de biți pentru formarea următoarei adrese de instrucțiune
6. Unitatea aritmetică-logică ALU:
  - proiectată anterior, pe 32 de biți
  - două intrări *A, B* și o ieșire cu rezultatul *Result*
  - operațiile de adunare, scădere, SAU logic, și ȘI logic sunt codificate prin semnalul de control *ALUCtrl* (pe 3 biți conform proiectării, permite adăugarea de noi operații)
  - detecția valorii de 0 pentru rezultatul curent, semnalată prin semnalul de stare *Zero* (*Zero*=1 dacă rezultatul ALU este 0, sau 0 în restul situațiilor)
7. Unitatea de extensie 16-32 cu sau fără semn *Ext*, se alege o codificare convenabilă pentru semnalul de control *ExtOp*:
  - *ExtOp* = 1, se face extensie cu semn
  - *ExtOp* = 0, se face extensie cu zero
8. Două unități de deplasare la stânga cu 2 poziții  $\ll 2$ . Una dintre unități efectuează și extinderea semnalului de intrare de la 26 la 28 de biți (pentru *j*).

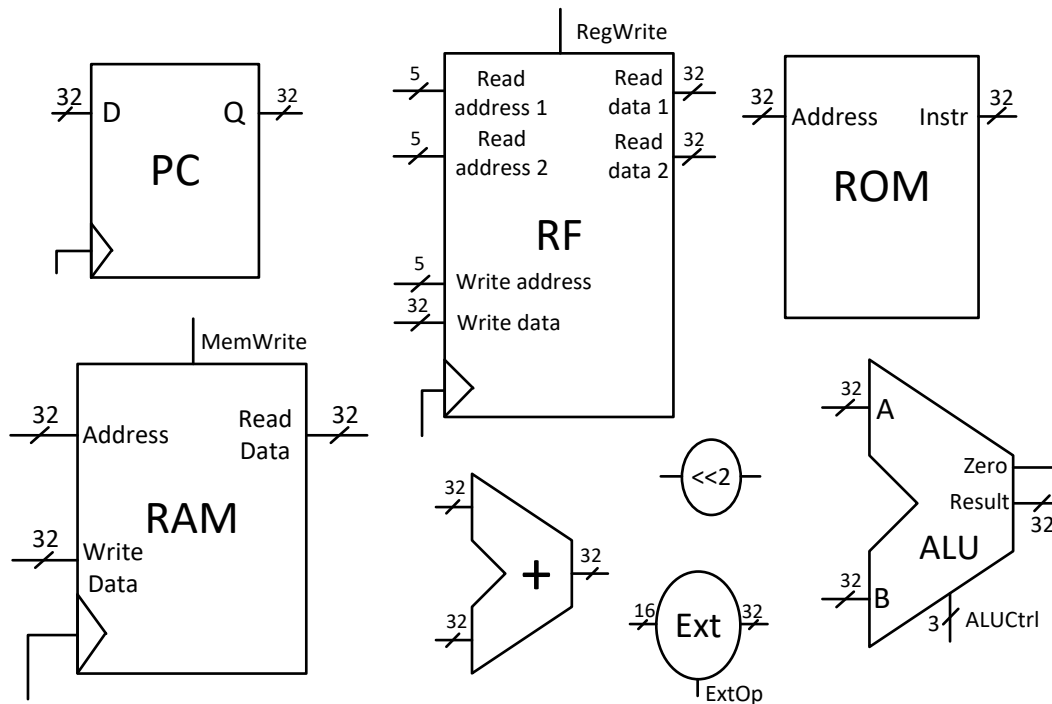


Fig. 9.3. Componentele rezultate din analiza RTL, descrierea în text

### 9.3.3. Asamblarea căii de date

La asamblarea căii de date e posibil să se identifice noi componente necesare. În mod particular, este vorba de multiplexoare, atunci când pe același semnal de intrare trebuie să ajungă valori din surse diferite în funcție de instrucțiune.

Asamblarea căii de date se poate face într-un singur pas, prin așezarea componentelor identificate într-o configurație convenabilă (în acord cu fluxul de date din RTL-ul instrucțiunilor, considerând registrul PC cel mai în stânga). Această abordare directă necesită experiență anterioară și este mai greu de înțeles la început, de aceea va fi discutată pe scurt la finalul acestei secțiuni.

Se va merge pe o abordare graduală, prin asamblarea mai multor căi parțiale de date pentru grupuri similare de instrucțiuni, urmând ca, la final, aceste căi de date să fie unite într-o cale de date comună. La unirea căilor parțiale de date se vor introduce multiplexoare acolo unde pe un semnal de intrare ajung valori din mai multe surse. Abordarea este similară cu cea din (Patterson & Hennessy, 2013), dar cu un nivel de detaliu mai mare și mai multe căi parțiale de date, insistând pe analiza atentă a transferurilor RTL și identificarea conexiunilor direct din aceste transferuri.

Pentru a urmări cât mai ușor fluxul de date, legăturile de pe scheme vor avea terminații de tip săgeată la destinație.

#### 9.3.3.1. Varianta cu căi parțiale de date

Asamblarea căilor parțiale de date se va ghida după etapele de execuție a instrucțiunilor.

Prima cale parțială de date este cea comună pentru faza IF de extragere a instrucțiunii. Componentele implicate sunt registrul PC, sumatorul pentru PC+4 și memoria ROM de instrucțiuni. Raționamentul este identic cu cel din capitolul 8, subsecțiunea 8.2.2.

Generarea adresei pentru următoarea instrucțiune este necesară pentru funcționarea secvențială a programului, fără salturi. Călea de date pentru IF este prezentată în Fig. 9.4. Semnalele de ieșire  $PC + 4$  și codul binar al instrucțiunii curente  $I(31..0)$  reprezintă legăturile cu următoarele căi parțiale de date.

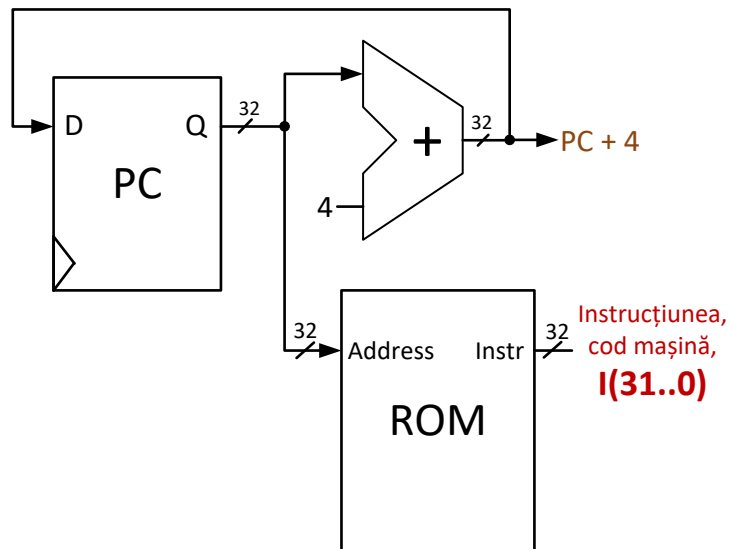


Fig. 9.4. Călea de date pentru IF

Următoarele căi parțiale de date vor fi construite din RTL pentru toate etapele rămase de execuție (ID, EX, MEM, WB). Pe măsură ce se vor obține căi de date cu elemente comune (rezultate din caracteristicile comune ale transferurilor RTL), acestea vor fi unificate.

Tiparul de construire a căii de date este următorul: (1) se aranjează convenabil componentele implicate în transferul RTL și (2) se completează legăturile/traseele pe schemă pe baza analizei transferului RTL. Criteriul „convenabil” înseamnă o aranjare care să permită un flux de date preponderent de la stânga spre dreapta.

Se va prezenta o analiză în detaliu a RTL doar pentru instrucțiunile de tip R, iar pentru restul instrucțiunilor se vor menționa direct legăturile deduse din RTL (pentru o mai bună aprofundare, cititorul este încurajat să facă analiza în detaliu la fiecare instrucțiune).

Pentru instrucțiuni de tip R, calea de date se assemblează (Fig. 9.5) din componentele identificate anterior, legăturile necesare deducându-se tot din transferul RTL. Analiza în detaliu pentru deducerea legăturilor din calea de date pentru tip R este prezentată în continuare:

1.  $RF[rd] \leftarrow RF[rs] \text{ op } RF[rt]$ : rd, rs și rt sunt câmpuri din codul mașină al instrucțiunii  $I(31..0)$ , conform formatului de tip R. Din semnalul  $I(31..0)$  se vor lega subgrupurile corespunzătoare adreselor de registre la intrările de adresă ale RF: **rs – Read address 1, rt – Read address 2, rd – Write address**
2.  $RF[rd] \leftarrow RF[rs] \text{ op } RF[rt]$ :  $RF[rs]$  și  $RF[rt]$  sunt ieșirile de date din RF, operația op fiind executată de ALU. Cele două ieșiri din RF se vor lega pe intrările ALU: **Read data 1 – intrare A din ALU, Read data 2 – intrare B din ALU**
3.  $RF[rd] \leftarrow RF[rs] \text{ op } RF[rt]$ : valoarea  $(RF[rs] \text{ op } RF[rt])$  este disponibilă pe ieșirea ALU și trebuie scrisă în RF. Ieșirea Result a ALU se va lega pe intrarea Write Data din RF: **ALU Result – Write Data**.

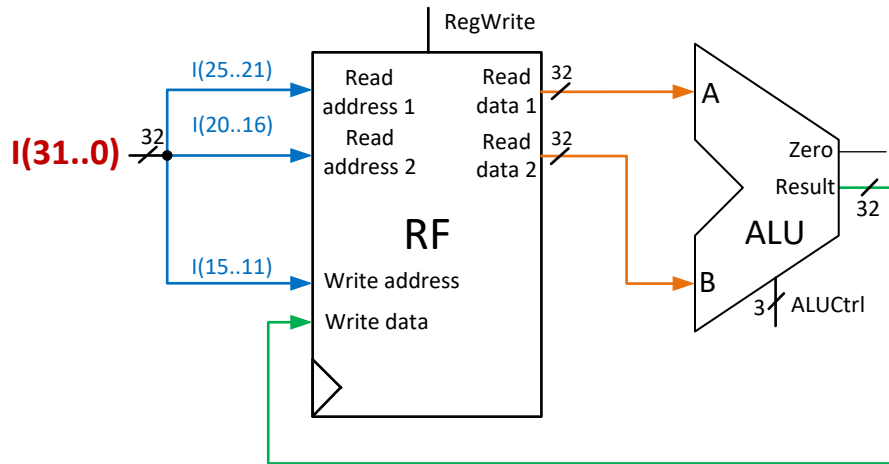


Fig. 9.5. Calea de date pentru tip R, legăturile dintre componente sunt prezentate cu culori diferite, conform analizei RTL (explicații în text, pe pagina anterioară)

Pentru instrucțiunile de tip I cu operații aritmetice-logice (*ori*, *addi*), analizând RTL-ul, rezultă următoarele conexiuni (Fig. 9.6):

*ori*:  $RF[rt] \leftarrow RF[rs] \text{ or } Z\_Ext(imm)$

*addi*:  $RF[rt] \leftarrow RF[rs] + S\_Ext(imm)$

1. Din  $I(31..0)$ : *rs* – Read address 1, *rt* – Write address, *imm* – intrare Ext
2. Operația executată de ALU are ca operanzi  $RF[rs]$  și *imm* (extins cu semn sau zero): Read data 1 – intrare A din ALU, ieșirea din Ext (circuitul de extensie) – intrare B din ALU
3. Rezultatul ALU trebuie să ajungă pe intrarea de date din RF: ALU Result – Write Data.

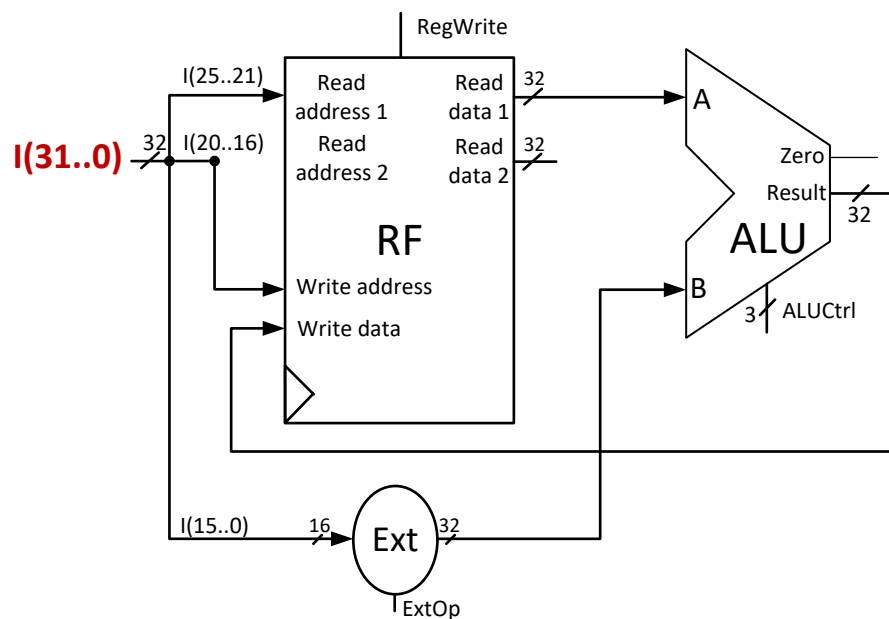


Fig. 9.6. Calea de date pentru tip I, instrucțiuni aritmetice-logice (*ori*, *addi*)

Pentru instrucțiunile de tip I de lucru cu memoria (*lw*, *sw*), din analiza RTL rezultă următoarele conexiuni (unele conexiuni din căile construite anterior se repetă, Fig. 9.7):

load word:  $RF[rt] \leftarrow M[RF[rs] + S\_Ext(imm)]$   
 store word:  $M[RF[rs] + S\_Ext(imm)] \leftarrow RF[rt]$

1. Din  $I(31..0)$ : *rs* – Read address 1, *rt* – Read address 2, *rt* – Write address, *imm* – intrare Ext
2. Calculul adresei efective de memorie se face de către ALU, care are ca operanzi  $RF[rs]$  și *imm* (extins cu semn): Read data 1 – intrare A din ALU, ieșirea din Ext (circuitul de extensie) – intrare B din ALU
3. Rezultatul ALU trebuie să ajungă pe intrarea de adresă a memoriei RAM de date: ALU Result – Address (RAM)
4. (pentru *lw*) Datele citite din memorie trebuie scrise în RF: Read Data (RAM) – Write data (RF)
5. (pentru *sw*) Valoarea  $RF[rt]$  trebuie scrisă în memoria RAM de date: Read data 2 (RF) – Write Data (RAM).

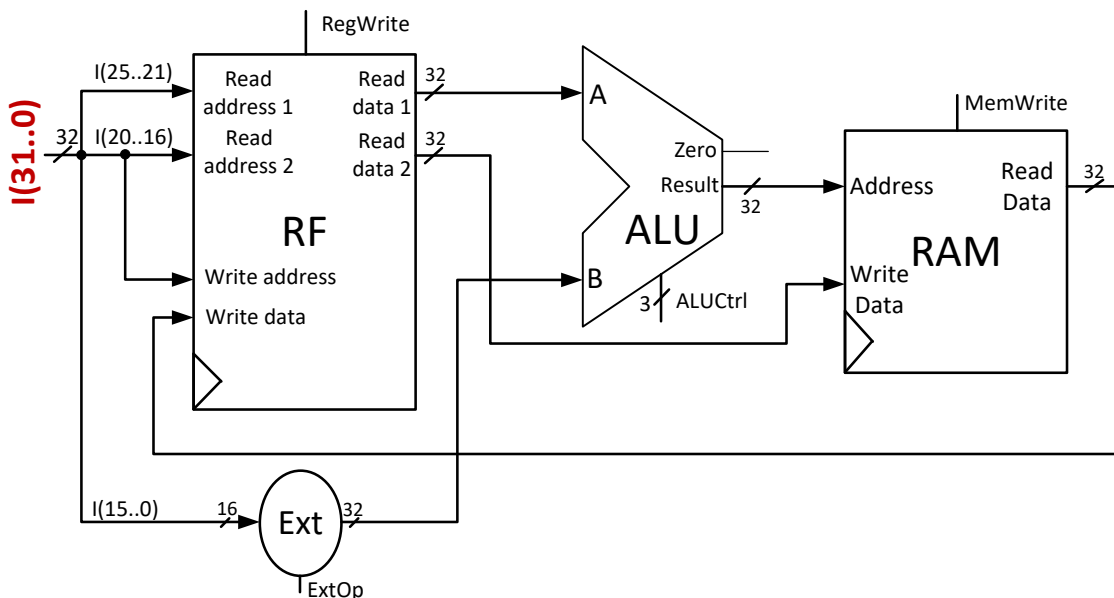


Fig. 9.7. Călea de date pentru tip I, instrucțiunile de lucru cu memoria (*lw*, *sw*)

Până acum au fost asamblate căile parțiale de date pentru instrucțiunile de tip R, tip I aritmetice-logice și tip I de lucru cu memoria. În continuare, se vor unifica aceste trei căi.

La unificarea a două căi parțiale de date se mențin componentele și conexiunile unice din fiecare schemă, se contopesc cele comune și se introduc multiplexoare acolo unde pe același semnal de intrare ajung rezultate diferite în funcție de instrucțiune.

Analizând cele două căi de date pentru tip I (din Fig. 9.6 și Fig. 9.7), se observă o singură situație unde este necesară introducerea unui multiplexor: pe intrarea Write data a RF trebuie legată atât ieșirea din ALU (Result), cât și ieșirea din memoria RAM (Read Data). Acest multiplexor 2:1 va avea semnalul de selecție MemtoReg: dacă semnalul este 0, calea de date obținută are comportamentul necesar pentru instrucțiunile *ori/addi*, iar dacă este 1 se va comporta conform cerințelor pentru lucrul cu memoria (*lw* mai exact). Călea

de date unificată pentru tip I (aritmetice-logice, lucru cu memoria) este prezentată în Fig. 9.8.

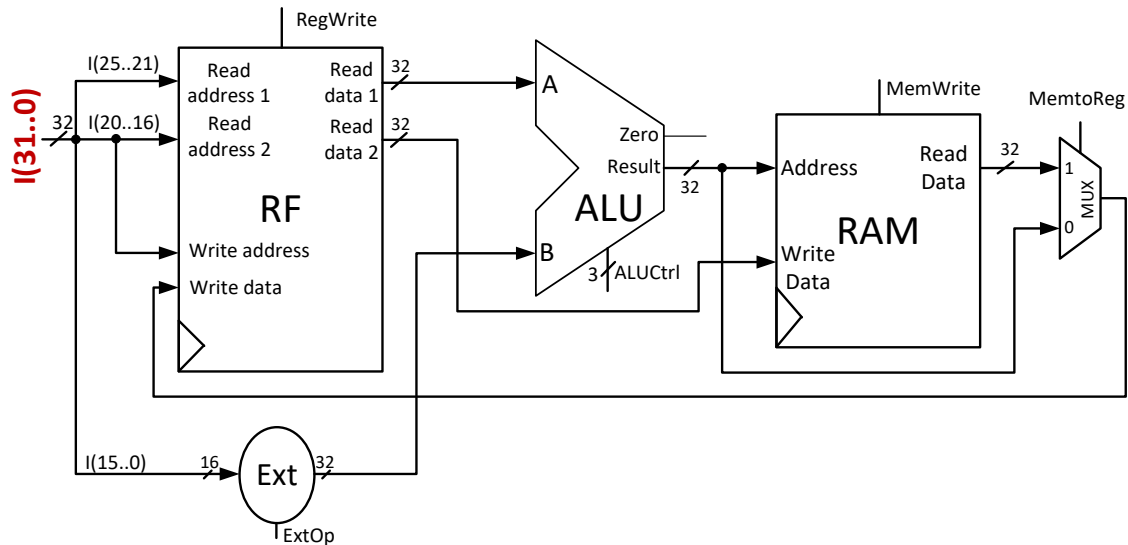


Fig. 9.8. Calea de date pentru tip I, instrucțiunile aritmetice-logice și de lucru cu memoria

Noua cale de date pentru tip I (Fig. 9.8) se unifică cu calea de date pentru tip R (Fig. 9.5), rezultatul fiind prezentat în Fig. 9.9. Apar două situații unde sunt necesare multiplexoare adiționale:

1. Prima este la intrarea Write address a RF pentru a selecta între câmpurile  $rt - I(20..16)$ , respectiv  $rd - I(15..11)$  din instrucțiune:
  - Se adaugă un mux 2:1 cu semnalul de selecție  $RegDst$ : dacă  $RegDst=0$ , atunci va trece valoarea  $rt$  (pentru tip I), altfel, dacă  $RegDst=1$ , va trece valoarea  $rd$  (pentru tip R)
2. A doua situație apare la intrarea B a ALU:
  - Se adaugă un nou mux 2:1 cu semnalul de selecție  $ALUSrc$  pentru a alege între valoarea furnizată de ieșirea Read data 2 a RF ( $ALUSrc=0$ ) sau ieșirea din unitatea Ext ( $ALUSrc=1$ ).

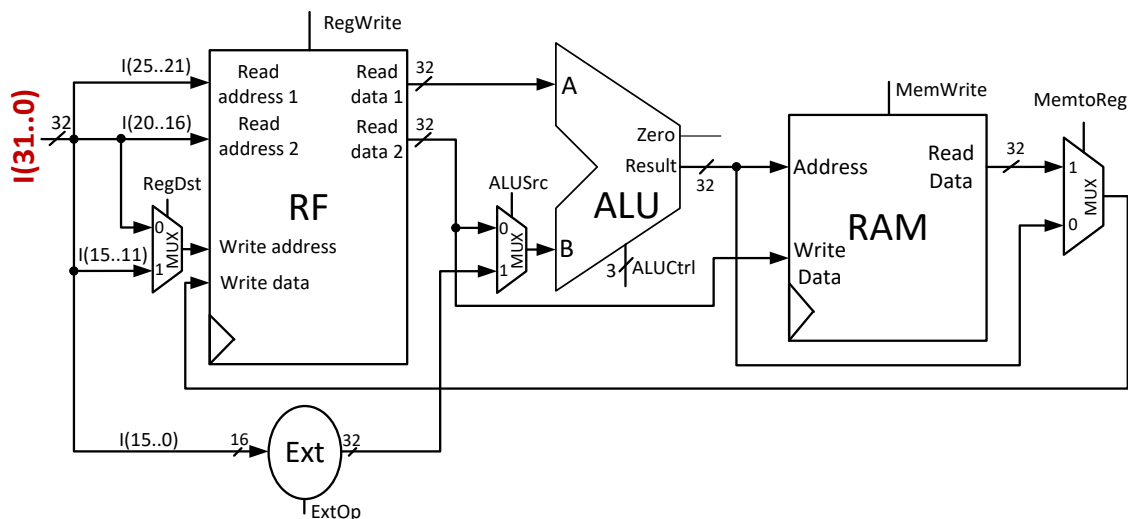


Fig. 9.9. Calea de date pentru tip R și tip I (instrucțiunile aritmetice-logice și lucru cu memoria)

Au mai rămas de asamblat căile de date pentru instrucțiunile de salt.

Pentru instrucțiunea de salt condiționat *beq* (tip I) se analizează RTL-ul și se ține cont de alocarea anterioară a operațiilor pe componente: ALU execută comparația dintre cele două registre, iar un sumator dedicat calculează adresa de salt.

*beq:*    If(RF[rs] == RF[rt]) then     $PC \leftarrow PC + 4 + S\_Ext(imm) \ll 2$   
    else     $PC \leftarrow PC + 4$

Rezultă următoarele conexiuni și calea de date din Fig. 9.10:

1. Conexiuni identice ca la tip R pentru legarea celor două registre (cu adresele rs și rt) la intrările ALU: rs – Read address 1, rt – Read address 2, Read data 1 – intrarea A din ALU, Read data 2 – intrarea B din ALU
2. Imediatul extins cu semn trebuie deplasat cu 2 poziții: legătură de la ieșirea unității Ext la intrarea circuitului de deplasare  $\ll 2$
3. Imediatul extins și deplasat trebuie adunat cu (PC + 4): ieșirea  $\ll 2$  și (PC + 4) se leagă la intrările sumatorului dedicat
4. În funcție de condiția evaluată, noua valoare care trebuie să ajungă pe intrarea PC este PC + 4 sau valoarea PC + 4 + S\_Ext(imm)  $\ll 2$ , disponibilă la ieșirea sumatorului. Se adaugă un nou mux 2:1 care va face selecția între cele două valori, în funcție de semnalul de selecție PCSrc (detalii în plus la proiectarea unității de control, pag. 91).

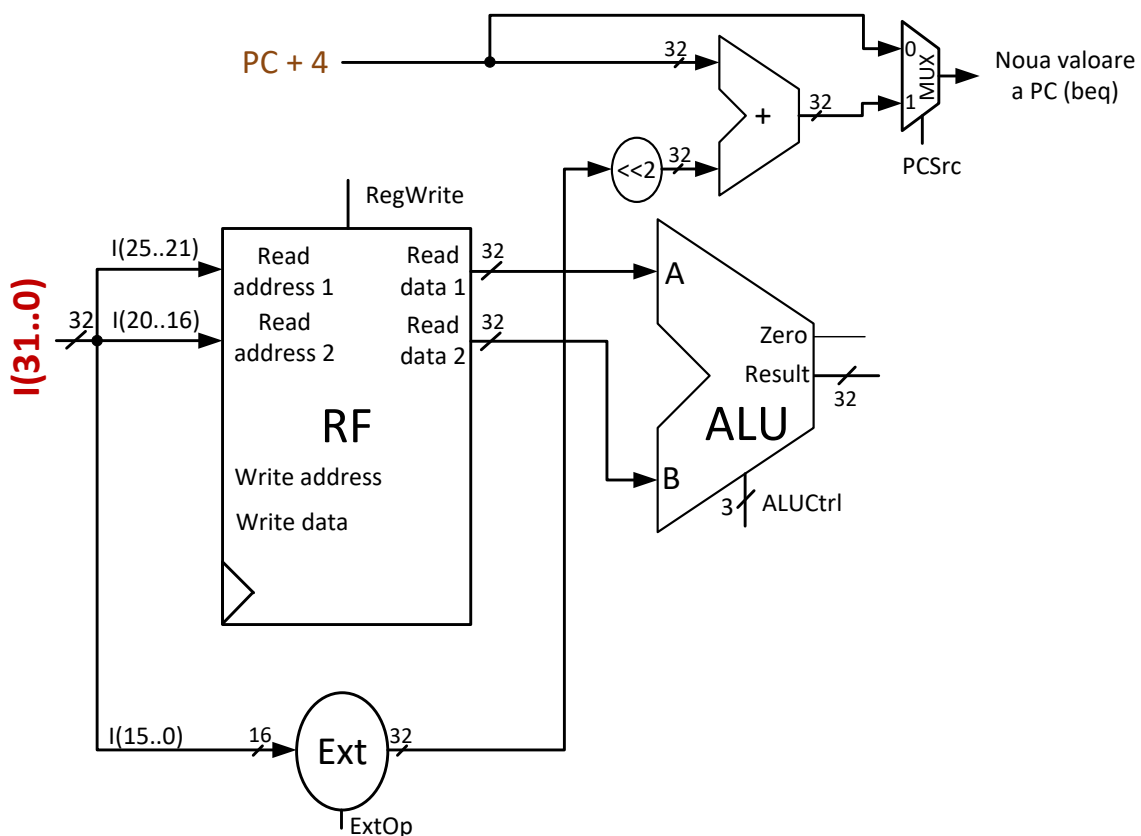


Fig. 9.10. Calea de date pentru *beq*, tip I

Ultima cale de date de asamblat este cea pentru instrucțiunea de salt necondiționat *j*, tip J.

Câmpul *target\_addr*, sau  $I(25..0)$ , din instrucțiune este legat la intrarea în circuitul de deplasare  $\ll 2$ , iar ieșirea din  $\ll 2$  este concatenată împreună cu primii 4 biți din  $PC+4$ . Rezultă calea de date din Fig. 9.11.

$$j: PC \leftarrow (PC+4)[31:28] \& \text{target\_addr} \& 00$$

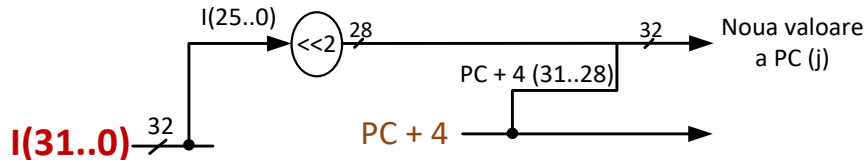


Fig. 9.11. Calea de date pentru *j*, tip J

Aceste ultime două căi parțiale de date se pot unifica ușor (Fig. 9.12). Fiecare cale parțială produce o nouă valoare pentru PC, valoare care trebuie să ajungă pe intrarea lui PC. Se adaugă un nou mux 2:1 care primește ca intrări cele două valori posibile pentru PC, cu semnalul de selecție *Jump*: dacă *Jump*=1, atunci va trece adresa absolută de salt (pentru *j*), altfel adresa de salt (*beq*) sau  $PC+4$  (depinde care a fost selectată de multiplexorul anterior).

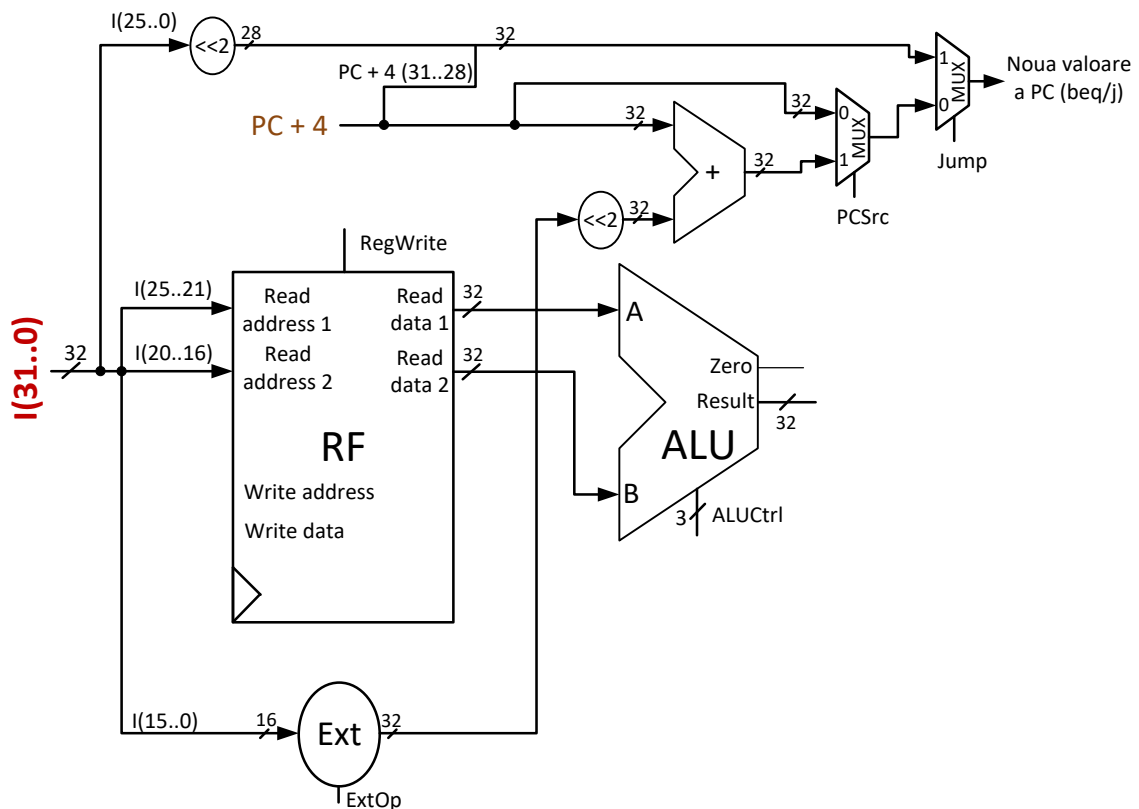


Fig. 9.12. Calea de date pentru instrucțiunile de salt *beq* (tip I) și *j* (tip J)



Până în acest moment, s-au obținut următoarele căi parțiale de date care trebuie unificate:

1. Calea de date pentru IF (toate instrucțiunile) cu funcționare secvențială pentru contorul de program (Fig. 9.4)
2. Calea de date (fără IF) pentru instrucțiunile de tip R și cele de tip I aritmetice-logice și lucrul cu memoria (Fig. 9.9)
3. Calea de date (fără IF) pentru instrucțiunile de salt *beq* - tip I, și *j* - tip J (Fig. 9.12).

Prima dată se vor unifica ultimele două căi enumerate, obținându-se calea parțială de date pentru toate instrucțiunile, cu etapele de execuție ID, EX, MEM și WB (Fig. 9.13). Nu sunt necesare multiplexoare suplimentare.

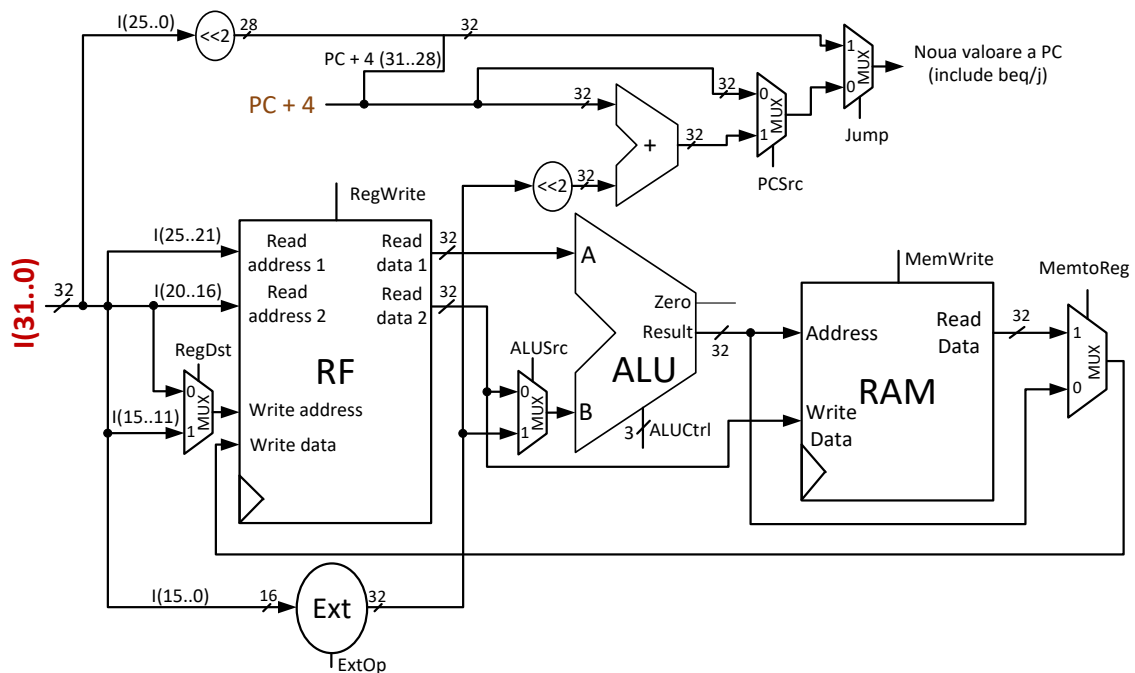


Fig. 9.13. Calea parțială de date pentru toate instrucțiunile, cu toate etapele de execuție exceptând IF

Se adaugă calea de date pentru IF și se obține calea de date a procesorului MIPS (Fig. 9.14, pagina 89). Unificarea se face prin legarea semnalelor comune PC + 4 și I(31..0), conexiunea (din IF) dintre sumator (PC + 4) și intrarea PC este eliminată, iar ieșirea de la multiplexorul cu semnalul de selecție Jump este legată la intrarea lui PC.



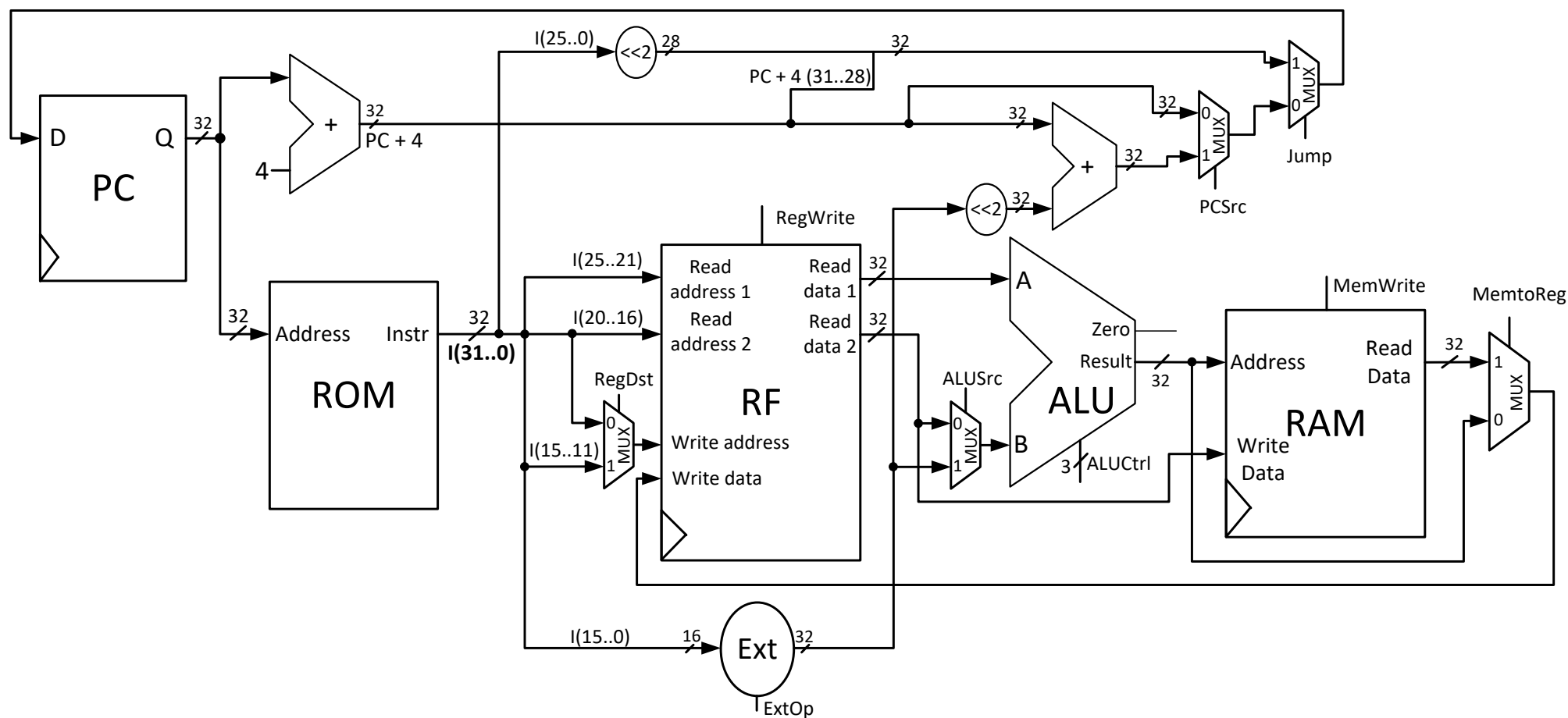


Fig. 9.14. Calea de date a procesorului MIPS 32 ciclu unic

### 9.3.3.2. Varianta de asamblare directă

O condiție necesară pentru asamblarea directă a căii de date este buna cunoaștere a setului de instrucțiuni, precum și înțelegerea modului în care fiecare instrucțiune se execută din punctul de vedere al transferului RTL.

Se începe cu o diagramă unde se poziționează de la stânga la dreapta, cu o spațiere corespunzătoare, cele 5 blocuri reprezentative: contorul de program PC, memoria ROM de instrucțiuni, blocul de registre RF, unitatea aritmetică-logică ALU și memoria de date RAM.

Se completează detaliile din schemă (conexiuni / alte componente) de la stânga la dreapta, conform fluxului de date care începe de la adresa instrucțiunii curente (ieșirea PC). Se consideră pentru fiecare componentă rolul în execuția tuturor instrucțiunilor suportate și se definesc conexiunile necesare (cu folosirea de multiplexoare unde este cazul).

### 9.3.4. Semnalele de control – definire completă

În acest pas de proiectare se vor enumera mai întâi semnalele de control din calea de date, cu semnificația lor, după care se vor stabili valorile semnalelor de control pentru execuția corectă a fiecărei instrucțiuni.

Semnalele de control pe un bit, menționate pe schemele anterioare, și efectul lor în calea de date sunt prezentate în continuare:

- **RegDst** – semnal de selecție pentru mux-ul de la adresa de scriere Write address a RF. Dacă are valoarea 0, atunci trece mai departe valoarea câmpului *rt*. Dacă are valoarea 1, atunci trece mai departe valoarea câmpului *rd*
- **RegWrite** – semnalul de validare a scrierii în RF. Dacă are valoarea 0, nu se execută scriere, altfel, dacă e 1, se scrie în blocul de registre valoarea de pe Write data la adresa de pe Write address
- **ExtOp** – controlează funcționarea unității Ext. Dacă are valoarea 0, imediatul se va extinde fără semn, cu 0. Dacă are valoarea 1, atunci se face extensie cu semn
- **ALUSrc** – semnal de selecție pentru mux-ul de la intrarea B a ALU. Dacă are valoarea 0, atunci pe intrarea B a ALU ajunge valoarea de la ieșirea Read data 2 a RF. Dacă are valoarea 1, atunci va ajunge valoarea de la ieșirea circuitului Ext
- **PCSrc** – semnal de selecție pentru mux-ul de salt condiționat. Dacă are valoarea 0, atunci trece mai departe valoarea PC + 4. Altfel, dacă are valoarea 1, va trece adresa de salt (pentru *beq*)
- **Jump** – semnal de selecție pentru mux-ul de salt necondiționat. Dacă are valoarea 0, atunci spre intrarea PC va trece valoarea prezentă pe ieșirea mux-ului precedent (**PCSrc**). Dacă are valoarea 1, atunci spre intrarea PC va trece valoarea adresei absolute de salt (pentru *j*)
- **MemWrite** – semnalul de validare a scrierii în memoria RAM de date. Dacă are valoarea 0 nu se execută scriere, altfel, dacă e 1, se scrie în memorie valoarea de pe Write Data la adresa de pe Address
- **MemtoReg** – semnal de selecție pentru mux-ul prin care se scrie înapoi rezultatul. Dacă are valoarea 0, atunci spre intrarea Write data a RF va trece rezultatul de la ieșirea ALU. Dacă are valoarea 1, atunci va trece valoarea citită din memoria de date.

Toate semnalele, mai puțin **PCSrc**, vor fi actualizate direct de către unitatea de control, în funcție de instrucțiunea curentă. Semnalul **PCSrc** depinde de semnalul de stare **Zero** generat de ALU. **PCSrc** ar trebui să aibă valoarea 1 într-un singur caz: instrucțiunea curentă aflată în execuție este *beq* și cele două registre citite din RF sunt egale. Egalitatea registrelor va fi semnalată prin semnalul de stare **Zero** care va avea valoarea 1 (se face o scădere în ALU și se activează **Zero** dacă rezultatul curent este nul). Se introduce un nou semnal de control, **Branch**, care va fi activat (=1) de către unitatea de control doar când instrucțiunea curentă este *beq*. Așadar, **PCSrc** trebuie să fie 1 doar dacă **Branch** și **Zero** sunt ambele 1, lucru care se testează cu o poartă ȘI (Fig. 9.15).



Fig. 9.15. Generarea semnalului **PCSrc** cu o poartă ȘI

Ultimul semnal de control, nediscutat până acum, este **ALUCtrl** (3 biți), care selectează operația curentă pe care o execută ALU. Codificarea binară a acestui semnal a fost prezentată deja în subcapitolul 5.5 când a fost construită ALU:

Tabel 9.2. Efectul **ALUCtrl** asupra ALU

<b>AluCtrl<sub>2...0</sub></b>	<b>Operație ALU</b>
0 0 0	Adunare
1 0 0	Scădere
0 0 1	ȘI
0 1 0	SAU

Prin unificarea căilor de date din pasul anterior s-a obținut o cale de date completă, care, în funcție de valorile semnalelor de control, se poate comporta în mod particular ca oricare dintre căile parțiale de la fiecare instrucțiune. Execuția corectă a unei instrucțiuni se face prin atribuirea valorilor necesare fiecărui semnal de control, astfel încât calea de date a procesorului să permită transferul RTL asociat cu instrucțiunea.

Pentru instrucțiunea *add*, tip R, transferul RTL este evidențiat (trasee colorate) în calea de date în Fig. 9.16. Atunci când nu contează valoarea unui semnal, acesta este marcat cu 'x' (=nu contează), dar, concret, el poate avea oricare dintre cele 2 valori (0 sau 1). Valorile semnalelor de control sunt stabilite astfel:

- **RegDst** – 1, pentru a trece adresa rd spre Write address a RF
- **RegWrite** – 1, pentru a valida scrierea în RF la finalul perioadei de ceas
- **ALUSrc** – 0, pentru a trece valoarea RF[rt] spre intrarea B a ALU
- **ExtOp** – x, deoarece imm nu este folosit în instrucțiunea *add*
- **ALUCtrl** – 000, operație de adunare
- **MemWrite** – 0, pentru a evita (!) o scriere eronată în memorie
- **MemtoReg** – 0, pentru a trece rezultatul ALU spre intrarea Write data a RF
- **Branch** – 0 (implicit **PCSrc** va fi 0), **Jump** – 0, pentru a permite trecerea valorii PC + 4 prin cele două multiplexoare.

Celelalte instrucțiuni de tip R au exact același tipar de execuție, singura diferență fiind la valoarea semnalului de control **ALUCtrl** (diferă doar operația din ALU).



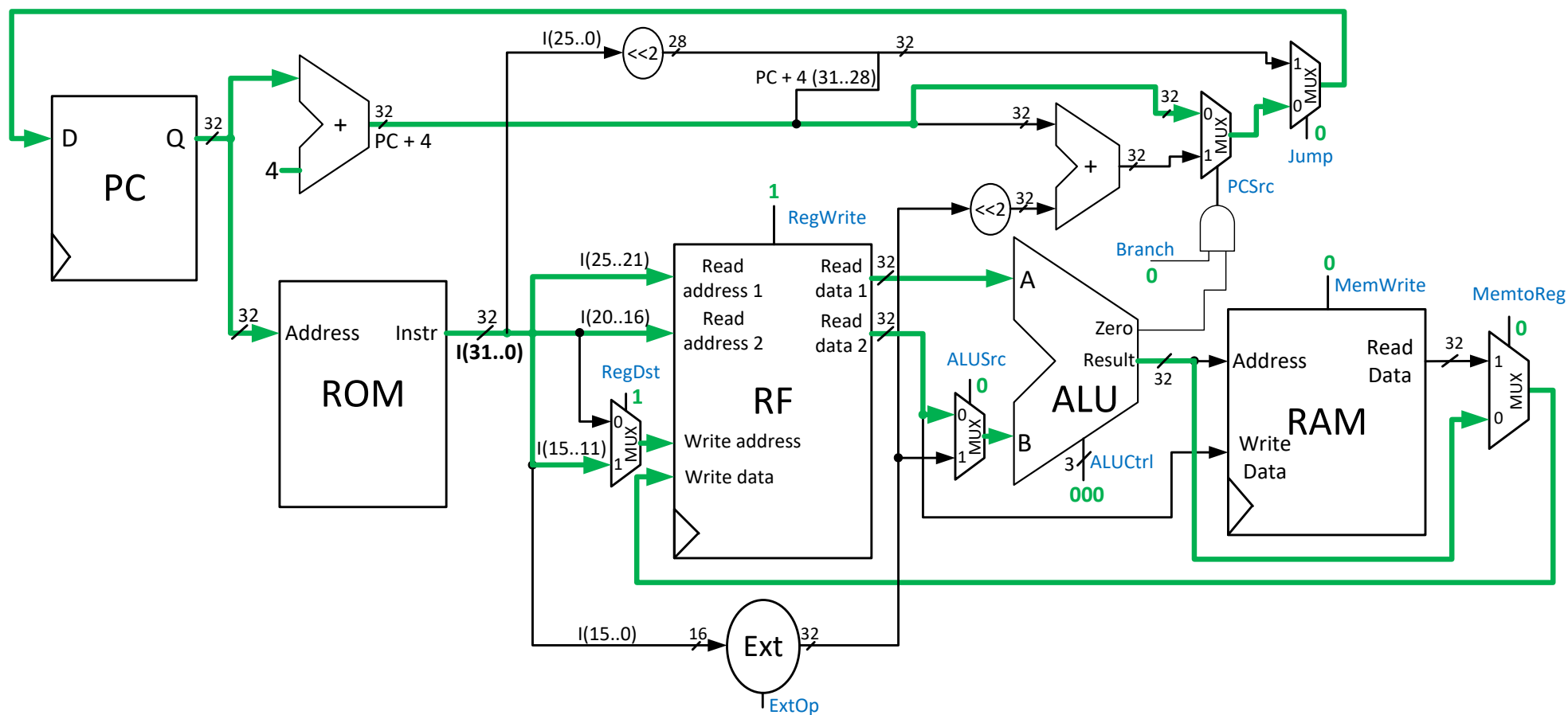


Fig. 9.16. Calea de date a procesorului MIPS 32 ciclu unic, cu transferul RTL pentru instrucțiunea *add*, tip R (cu verde). Sunt indicate valorile semnalelor de control pentru ca acest transfer să aibă loc.  $RF[rd] \leftarrow RF[rs] + RF[rt]$ ,  $PC \leftarrow PC + 4$

Pentru restul instrucțiunilor, cititorul este încurajat să traseze transferul RTL pe calea de date a procesorului, pentru a identifica mai ușor valorile corecte ale semnalelor de control. În figurile următoare, se prezintă aceste transferuri, având originea în ieșirea Q a contorului de program PC.

Instrucțiunile *ori* și *addi*, tip I, se execută în mod similar (aceleași transferuri RTL), singurele diferențe fiind la operația din ALU și la unitatea de extensie:

- **RegDst** – 0, pentru a trece adresa *rt* spre **Write address** a RF
- **RegWrite** – 1, pentru a valida scrierea în RF la finalul perioadei de ceas
- **ALUSrc** – 1, pentru a trece valoarea *imm* extinsă spre intrarea B a ALU
- **ExtOp**
  - *ori*: 0, pentru extinderea cu 0 a *imm*
  - *addi*: 1, pentru extinderea cu semn a *imm*
- **ALUCtrl**
  - *ori*: 010, pentru operația SAU logic în ALU
  - *addi*: 000, pentru operația de adunare în ALU
- **MemWrite** – 0, pentru a evita (!) o scriere eronată în memorie
- **MemtoReg** – 0, pentru a trece rezultatul ALU spre intrarea **Write data** a RF
- **Branch** – 0 (implicit **PCSrc** va fi 0), **Jump** – 0, pentru a permite trecerea valorii PC + 4 prin cele două multiplexoare.

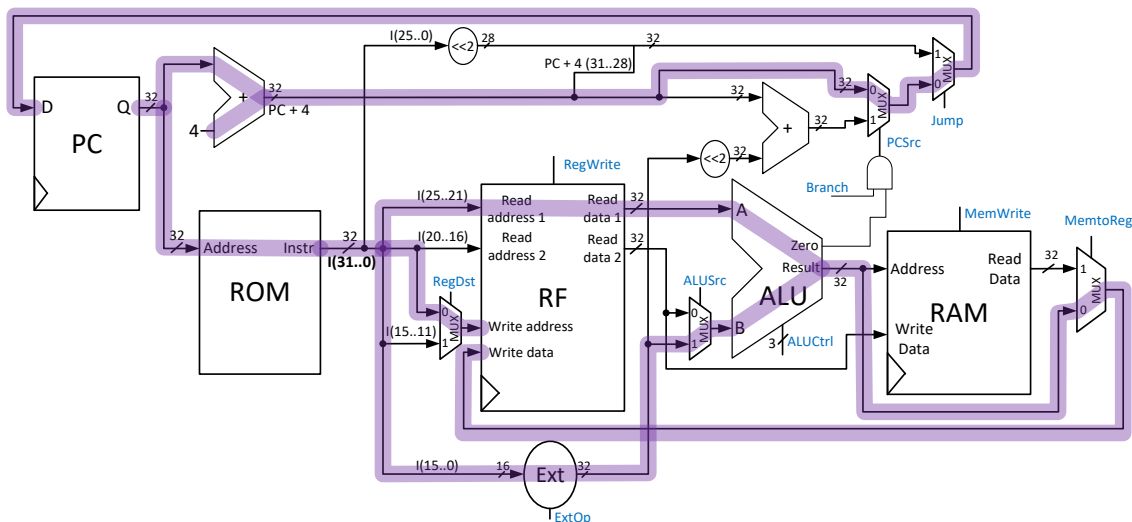


Fig. 9.17. Transferul RTL pentru instrucțiunile *addi* și *ori*.  $RF[rt] \leftarrow RF[rs] + S\_Ext(imm)$  sau  $RF[rt] \leftarrow RF[rs] \text{ or } Z\_Ext(imm)$ ,  $PC \leftarrow PC + 4$

Pentru instrucțiunea *lw* (load word), valorile semnalelor de control sunt:

- **RegDst** – 0, pentru a trece adresa rt spre Write address a RF
- **RegWrite** – 1, pentru a valida scrierea în RF la finalul perioadei de ceas
- **ALUSrc** – 1, pentru a trece valoarea imm extinsă spre intrarea B a ALU
- **ExtOp** – 1, pentru extinderea cu semn a imm
- **ALUCtrl** – 000, adunare în ALU – calcularea adresei efective de memorie
- **MemWrite** – 0, pentru a evita (!) o scriere eronată în memoria de date
- **MemtoReg** – 1, pentru a trece valoarea citită din memorie spre Write data din RF
- **Branch** – 0 (implicit **PCSrc** va fi 0), **Jump** – 0, pentru a permite trecerea valorii PC + 4 prin cele două multiplexoare.



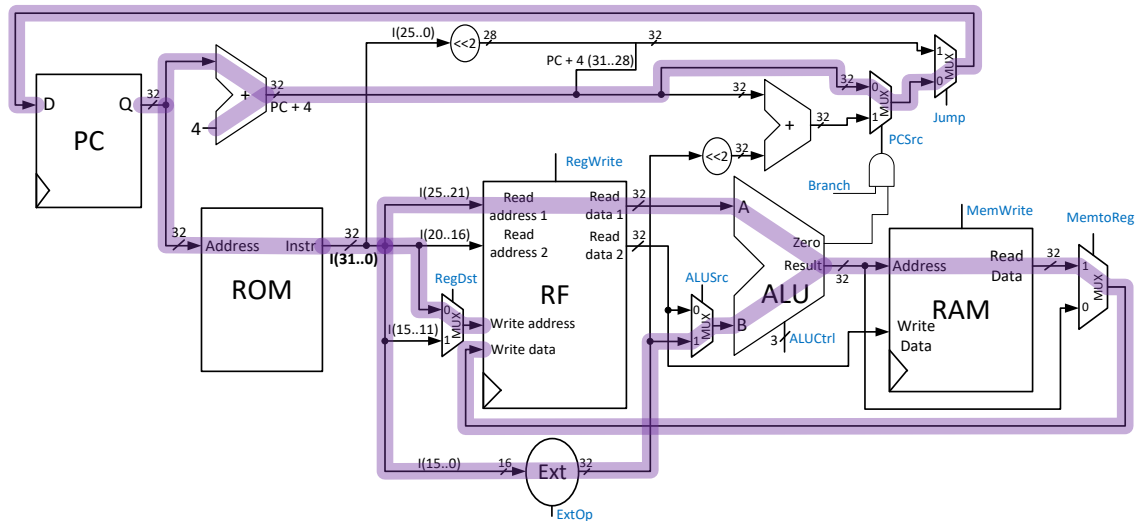


Fig. 9.18. Transferul RTL pentru instrucțiunea *lw*.  $RF[rt] \leftarrow M[RF[rs] + S\_Ext(imm)]$ ,  $PC \leftarrow PC + 4$

Pentru instrucțiunea *sw* (store word), valorile semnalelor de control sunt:

- **RegDst** – x, nu contează, pentru că nu se scrie în RF
- **RegWrite** – 0, pentru a evita (!) o scriere eronată în RF
- **ALUSrc** – 1, pentru a trece valoarea imm extinsă spre intrarea B a ALU
- **ExtOp** – 1, pentru extinderea cu semn a imm
- **ALUCtrl** – 000, adunare în ALU – calcularea adresei efective de memorie
- **MemWrite** – 1, pentru a valida scrierea în memoria de date la finalul perioadei de ceas
- **MemtoReg** – x, nu contează pentru că nu se scrie în RF
- **Branch** – 0 (implicit **PCSrc** va fi 0), **Jump** – 0, pentru a permite trecerea valorii PC + 4 prin cele două multiplexoare.

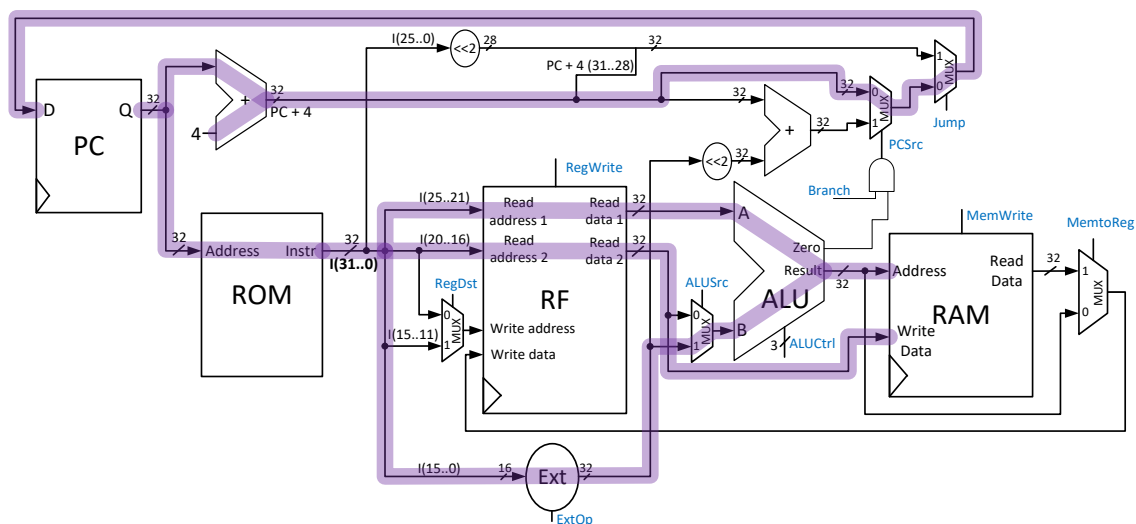


Fig. 9.19. Transferul RTL pentru instrucțiunea *sw*.  $M[RF[rs] + S\_Ext(imm)] \leftarrow RF[rt]$ ,  $PC \leftarrow PC + 4$

În cazul instrucțiunilor de salt trebuie protejate la scriere blocul de registre și memoria de date. Instrucțiunea de salt condiționat *beq* necesită următoarele valori pentru control:

- **RegDst** – x, nu contează pentru că nu se scrie în RF
- **RegWrite** – 0, pentru a evita (!) o scriere eronată în RF
- **ALUSrc** – 0, pentru a trece valoarea RF[rt] spre intrarea B a ALU (se face comparația)
- **ExtOp** – 1, pentru extinderea cu semn a imm
- **ALUCtrl** – 100, scădere în ALU – în caz de rezultat 0 se va activa semnalul **Zero**
- **MemWrite** – 0, pentru a evita (!) o scriere eronată în memoria de date
- **MemtoReg** – x, nu contează pentru că nu se scrie în RF
- **Branch** – 1, implicit **PCSrc** va depinde de semnalul **Zero** (egalitate sau nu)
- **Jump** – 0, pentru a permite trecerea valorii din mux-ul anterior (**PCSrc**).

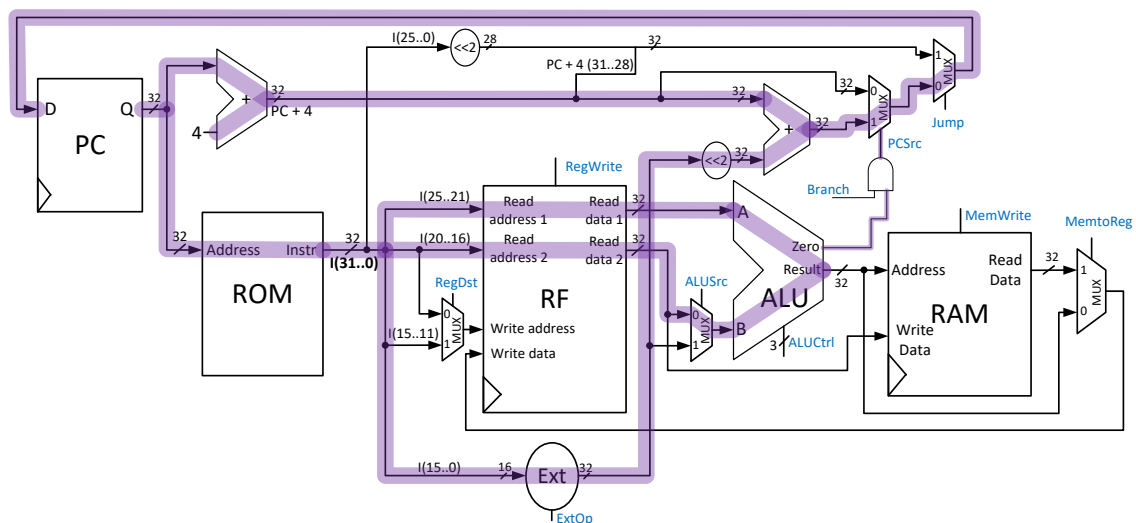


Fig. 9.20. Transferul RTL pentru instrucțiunea *beq*, în ipoteza că se execută saltul: cele două registre sunt egale, deci semnalul Zero este 1.  $\text{If}(\text{RF}[\text{rs}] == \text{RF}[\text{rt}]) \text{ then } \text{PC} \leftarrow \text{PC} + 4 + \text{S\_Ext}(\text{imm}) \ll 2, \text{ else } \text{PC} \leftarrow \text{PC} + 4$

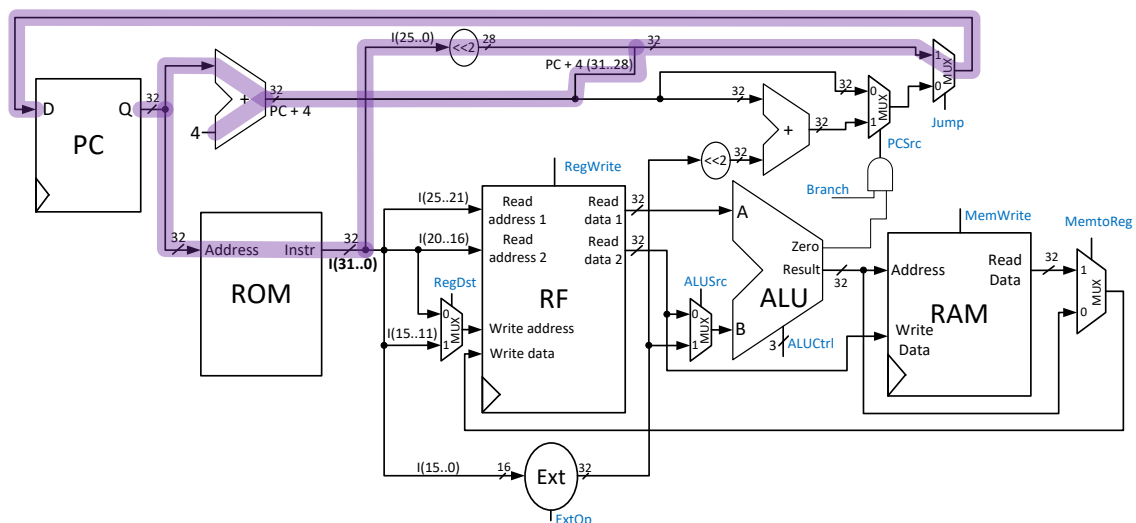


Fig. 9.21. Transferul RTL pentru instrucțiunea *j*.  $\text{PC} \leftarrow (\text{PC} + 4)[31:28] \& \text{target\_addr} \& 00$

Instrucțiunea de salt necondiționat *j* nu necesită calcule, exceptând pe ramura de calcul a noii valori pentru PC:

- **RegWrite** – 0, pentru a evita (!) o scriere eronată în RF
- **MemWrite** – 0, pentru a evita (!) o scriere eronată în memoria de date
- **Jump** – 1, pentru a permite trecerea adresei absolute de salt
- **Restul semnalelor** – x, nu contează.

În concluzie, valorile stabilite pentru semnalele de control sunt centralizate în tabelul următor.

Tabel 9.3. Specificarea completă a semnalelor de control

Instrucțiune	Reg Dst	Reg Write	ALU Src	Ext Op	ALU Ctrl	Mem Write	Memto Reg	Branch	Jump
<b>add</b> (tip R)	1	1	0	x	000 (+)	0	0	0	0
<b>sub</b> (tip R)	1	1	0	x	100 (-)	0	0	0	0
<b>or</b> (tip R)	1	1	0	x	010 (sau)	0	0	0	0
<b>and</b> (tip R)	1	1	0	x	001 (și)	0	0	0	0
<b>ori</b>	0	1	1	0	010 (sau)	0	0	0	0
<b>addi</b>	0	1	1	1	000 (+)	0	0	0	0
<b>lw</b>	0	1	1	1	000 (+)	0	1	0	0
<b>sw</b>	x	0	1	1	000 (+)	1	x	0	0
<b>beq</b>	x	0	0	1	100 (-)	0	x	1	0
<b>j</b>	x	0	x	x	xxx	0	x	x	1

### 9.3.5. Unitatea de control

În acest ultim pas se va implementa unitatea de control. Valorile pentru semnalele de control sunt necesare în calea de date imediat ce începe ciclul de ceas aferent instrucțiunii curente. Unitatea de control trebuie să fie combinațională, valorile semnalelor depinzând doar de câmpurile care codifică operația din formatul de instrucțiune (opcode și funct).

În general, implementarea unei unități de control combinaționale se face cu un circuit de decodificare (subcapitolul 5.3), care primește pe intrare codul de operație din instrucțiune și generează pe ieșire valorile semnalelor de control.

Pentru procesorul MIPS (Patterson & Hennessy, 2013), versiunea cu ciclu unic, se folosește o unitate de control ierarhică formată din unitatea principală de control (*Main Control*) și din unitatea secundară de control (*ALU control*). Există două motive pentru această separare:

1. Comportamentul identic pentru instrucțiunile de tip R, care au aceleași valori pentru toate semnalele de control, exceptând **ALU Ctrl**
2. Codificarea operației în formatul de tip R: codul principal opcode=0, iar operația efectivă este codificată în câmpul funct.

Rolul unității principale *Main Control* este de a interpreta câmpul opcode din formatul de instrucțiune și de a trimite direct valorile necesare pe semnalele de control, mai puțin pentru **ALU Ctrl**, care va fi generat de unitatea secundară *ALU Control*. Pentru a genera

valoarea lui **ALUCtrl** va interpreta comanda primită de la *Main Control* (pentru tip I) sau câmpul funct (pentru tip R). Pentru transmiterea comenzii de la *Main Control* la *ALU Control* se folosește un nou semnal intermediar **ALUOp** (=ALU operation). Diagrama bloc pentru unitatea ierarhică de control este prezentată în Fig. 9.22.

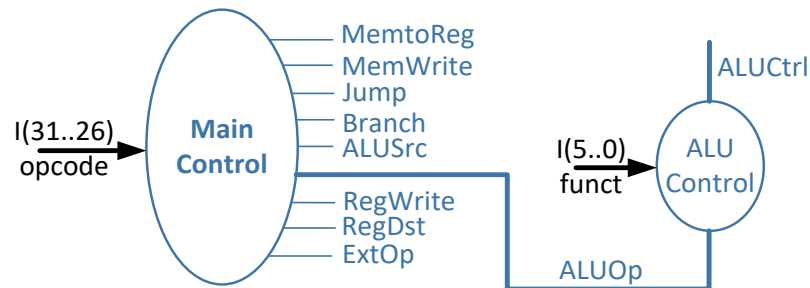


Fig. 9.22. Unitatea ierarhică de control a procesorului MIPS, schema bloc

Pentru setul ales de instrucțiuni, *Main Control* poate comanda unității *ALU control* următoarele (simbolic, deduse din cerințele pentru ALU la fiecare instrucțiune): „instrucțiune tip R, folosește funct”, „instrucțiunea *ori*, comandă SAU pentru ALU”, „instrucțiunea *addi / lw / sw*, comandă + pentru ALU” sau „instrucțiunea *beq*, comandă – pentru ALU”. Sunt 4 comenzi care, prin intermediul **ALUOp**, se vor codifica pe 2 biți (codificarea este aleasă conform descrierii din literatură, chiar dacă acest aspect nu este important în procesul de proiectare):

Tabel 9.4. Codificarea pentru **ALUOp**

Instrucțiune	<b>ALUOp</b>	Semnificația comenzii pentru ALU Control
tip R	10	folosește funct
ori	11	comandă SAU
addi / lw / sw	00	comandă +
beq	01	comandă –

Unitatea *Main control* va genera direct semnalele de control pornind de la codul de operație al instrucțiunii curente (a se vedea formatul instrucțiunilor):

Tabel 9.5. Valorile semnalelor de control pentru unitatea principală de control

opcode (instr)	Reg Dst	Reg Write	ALU Src	Ext Op	<b>ALUOp<sub>1..0</sub></b>	Mem Write	Memto Reg	Branch	Jump
<b>000000</b> (tip R)	1	1	0	x	10 (funct)	0	0	0	0
<b>001101</b> (ori)	0	1	1	0	11 (sau)	0	0	0	0
<b>001000</b> (addi)	0	1	1	1	00 (+)	0	0	0	0
<b>100011</b> (lw)	0	1	1	1	00 (+)	0	1	0	0
<b>101011</b> (sw)	x	0	1	1	00 (+)	1	x	0	0
<b>000100</b> (beq)	x	0	0	1	01 (–)	0	x	1	0
<b>000010</b> (j)	x	0	x	x	xx	0	x	x	1

Unitatea secundară va genera valoarea semnalului **ALUCtrl** pe baza valorilor **ALUOp** și **funct** (pentru tip R, la restul nu contează) după cum urmează (în concordanță cu codificarea din Tabel 9.3):

Tabel 9.6. Valoarea **ALUCtrl** în funcție de **ALUOp** și **funct**

Instrucțiune	ALUOp	funct	ALUCtrl	operația ALU
add (tip R)	10	100000	000	adunare
sub (tip R)	10	100010	100	scădere
or (tip R)	10	100101	010	SAU
and (tip R)	10	100100	001	ȘI
ori	11	xxxxxx	010	SAU
addi / lw / sw	00	xxxxxx	000	adunare
beq	01	xxxxxx	100	scădere

În continuare, se va construi unitatea principală de control până la nivel de porți logice, folosind rețele logice programabile (eng. Programmable Logic Arrays). Din perspectiva celor discutate deja în subcapitolul 3.3, abordarea pleacă de la reprezentarea unei ecuații booleene ca sumă de produse, însă e aplicată intuitiv: se vor folosi porți ȘI, pentru a „detecta” apariția fiecărei instrucțiuni, și porți SAU, pentru a genera efectiv valorile semnalelor. Fiecare semnal de control va fi generat ca o sumă (SAU) de produse (ȘI).

Cum se poate detecta apariția unei anumite instrucțiuni? Practic, trebuie recunoscut codul de operație al instrucțiunii. Se folosește câte o poartă ȘI pentru fiecare opcode distinct, cu un număr de intrări egal cu dimensiunea opcode (6 biți). Pentru o anumită instrucțiune, fiecare poziție de bit din opcode se va lega pe intrarea corespunzătoare a porții ȘI, iar pozițiile care corespund unor valori de 0 se vor nega. Exemplu: opcode(5..0) pentru ori – 001101, pentru a obține 1 pe ieșirea porții ȘI cu 6 intrări trebuie negate pozițiile 5, 4 și 1. Porțile ȘI reprezintă termenii canonici din suma de produse. Aceste porți, care detectează apariția instrucțiunilor, sunt prezentate în Fig. 9.23. Intrările negate sunt evidențiate prin cercuri.

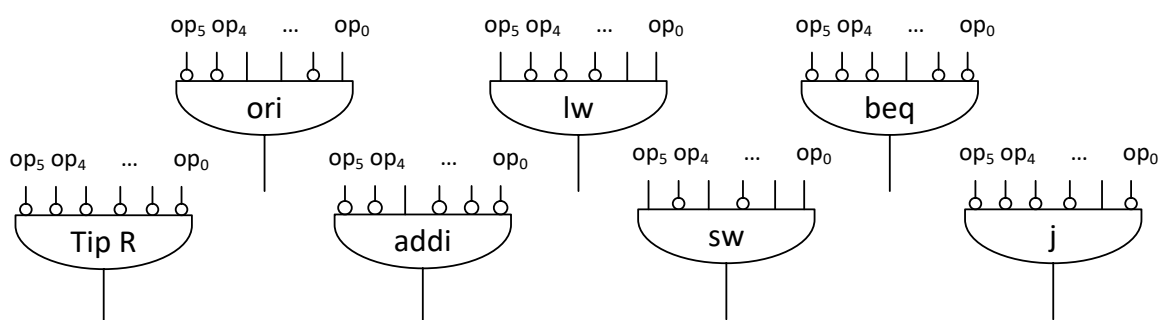


Fig. 9.23. Porțile ȘI care detectează apariția fiecărui cod de operație. Intrările negate corespund valorilor de 0 din codurile de operație op (opcode)

Pentru generarea fiecărui semnal de control, se identifică instrucțiunile pentru care semnalul trebuie să aibă valoarea 1 (Tabel 9.5). De exemplu, semnalul **RegWrite** trebuie să fie activ dacă instrucțiunea este de tip R sau **ori** sau **addi** sau **lw**, iar pentru restul

instrucțiunilor trebuie să fie inactiv. Valoarea semnalului de control se obține printr-o operație SAU logic între ieșirile din porțile ȘI asociate cu instrucțiunile pentru care semnalul trebuie activat. Pentru semnalele pe mai mulți biți se va considera fiecare poziție separat (de exemplu **ALUOp**). Nu sunt necesare porți SAU pentru acele semnale care trebuie să fie active doar pentru o singură instrucțiune. Implementarea completă a unității principale de control este prezentată în Fig. 9.24.

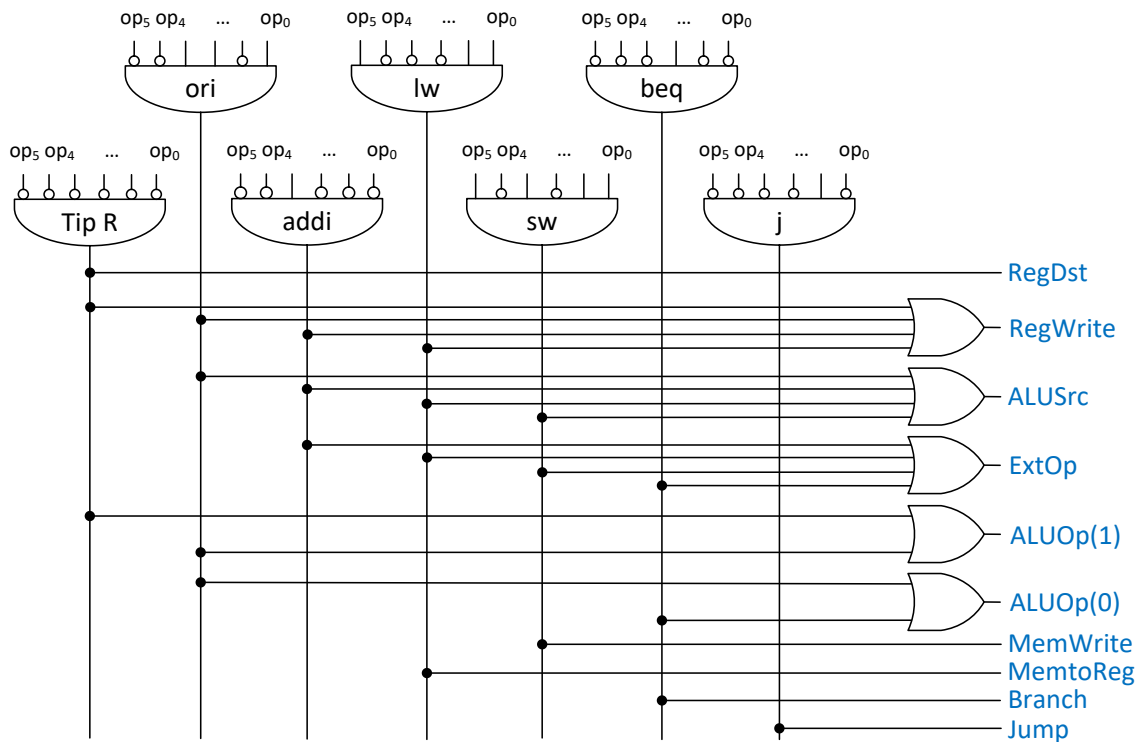


Fig. 9.24. Implementarea unității principale de control (*Main Control*) la nivel de porți logice

Implementarea unității secundare *ALU Control* se face printr-o abordare similară, porțile ȘI detectând combinațiile de biți care apar pentru semnalele de intrare **ALUOp** și funct.

La finalul acestui pas, unitatea de control se leagă la calea de date, obținându-se procesorul din Fig. 9.25 (pagina 103).

#### 9.4. Funcționarea în implementarea fizică. Frecvența de ceas

Transferurile RTL specifice fiecărei instrucțiuni au fost trasate pe calea de date a procesorului în secțiunea 9.3.4 (se recomandă consultarea figurilor pentru a înțelege mai bine paragrafele următoare).

Toate transferurile încep de la ieșirea Q a registrului PC, imediat după începerea perioadei curente de ceas (după frontul crescător), se propagă combinațional prin calea de date (prin elementele implicate în transfer) și se termină în potențialele destinații unde scrierea se va executa pe frontul crescător de la finalul perioadei de ceas: intrarea D a PC, intrarea Write data a blocului de registre RF, respectiv intrarea Write Data a memoriei de date RAM.

În implementarea fizică, fiecare element are anumite caracteristici de timp pentru propagarea semnalelor (a se revedea subcapitolul 6.3). Inclusiv la elementele combinaționale, atunci când se schimbă valorile de pe semnalele de intrare, va exista o anumită întârziere specifică până la apariția noii valori pe semnalul de ieșire. Fiecare transfer RTL se va executa pe implementarea fizică prin propagarea valorilor (niveluri de tensiune) de la ieșirea Q a PC spre diferitele destinații, prin elementele întâlnite pe traseu – cu întârzierile asociate. Valorile pentru destinație se vor stabili abia după propagarea completă (se însumează întârzierile de pe traseu) prin calea de date.

Dacă se analizează diferitele transferuri RTL care apar, se constată ca acestea sunt de diferite lungimi, respectiv întârzieri cumulate. Pentru fiecare instrucțiune se poate calcula cel mai lung traseu, considerând ca referință frontul crescător de la începutul perioadei de ceas.

La instrucțiunile de tip R, Fig. 9.16, cel mai lung traseu este dat de următoarele întârzieri: timp de propagare  $t_{prop}$  pentru PC + acces ROM instrucțiuni + acces RF + Mux (**ALUSrc**) + operație ALU + Mux (**MemtoReg**) + timp  $t_{setup}$  la scriere în RF.

Pentru instrucțiunile de tip I *ori/addi* traseul cel mai lung este similar cu tip R.

În cazul instrucțiunilor de lucru cu memoria *lw/sw*, cel mai lung traseu apare pentru *lw* (față de tip R se adaugă accesarea memoriei de date): timp de propagare  $t_{prop}$  pentru PC + acces ROM instrucțiuni + acces RF + Mux (**ALUSrc**) + operație ALU + **acces RAM** + Mux (**MemtoReg**) + timp  $t_{setup}$  la scriere în RF.

Instrucțiunea *beq* are un traseu mai scurt: timp de propagare  $t_{prop}$  pentru PC + acces ROM instrucțiuni + acces RF + Mux (**ALUSrc**) + operație ALU (**Zero**) + Mux (**PCSrc**) + Mux (**Jump**) + timp  $t_{setup}$  la scriere în PC.

Cel mai scurt traseu este cel pentru instrucțiunea *j*: timp de propagare  $t_{prop}$  pentru PC + acces ROM instrucțiuni +  $<<2$  + Mux (**Jump**) + timp  $t_{setup}$  la scriere în PC.

Analizând traseele discutate, cel mai lung traseu apare pentru instrucțiunea *lw*. Acest traseu poartă denumirea de cale critică, fiind cea mai lungă cale combinațională din procesor. Pentru execuția corectă a instrucțiunilor, perioada de ceas trebuie să acopere cel puțin durata căii critice. Altfel spus, procesorul poate funcționa doar la o frecvență pentru care perioada de ceas este mai mare sau egală cu calea critică. În funcție de performanța implementării fizice a elementelor din procesor, se poate obține o cale critică de durată mai scurtă și, implicit, un procesor care poate funcționa la o frecvență mai mare. O analiză extinsă, inclusiv cu exemple numerice pentru întârzieri, poate fi consultată în (Harris & Harris, 2013), capitolul 7, secțiunea 7.3.4, sau în (Patterson & Hennessy, 2002), capitolul 5, finalul subcapitolului 5.3.

Prin folosirea unei perioade de ceas care acoperă cazul cel mai defavorabil (cea mai costisitoare instrucțiune) apare un compromis: restul instrucțiunilor, cu trasee mai scurte, se vor executa în același interval de timp - mai mare decât ar fi necesar. Pentru o secvență de program timpul total de execuție va fi mai mare față de timpul teoretic care s-ar obține dacă fiecare instrucțiune s-ar executa pe durata strict necesară (individual). Acest dezavantaj se elimină prin execuția instrucțiunilor pe mai multe perioade de ceas. În funcție de abordare, procesorul se reproiectează (prin transformarea variantei *ciclu unic*) și se obține un procesor *multi-ciclu* sau *pipeline*. Proiectarea procesorului MIPS *multi-ciclu*, unde fiecare instrucțiune se execută pe un anumit număr de cicluri de ceas, este prezentată în subcapitolul 7.4 din (Harris & Harris, 2013) sau în subcapitolul 5.4 din (Patterson & Hennessy, 2002). Proiectarea variantei *pipeline* a procesorului MIPS este prezentată în subcapitolul 7.5 din (Harris & Harris, 2013) sau în capitolul 4 din (Patterson & Hennessy, 2013).





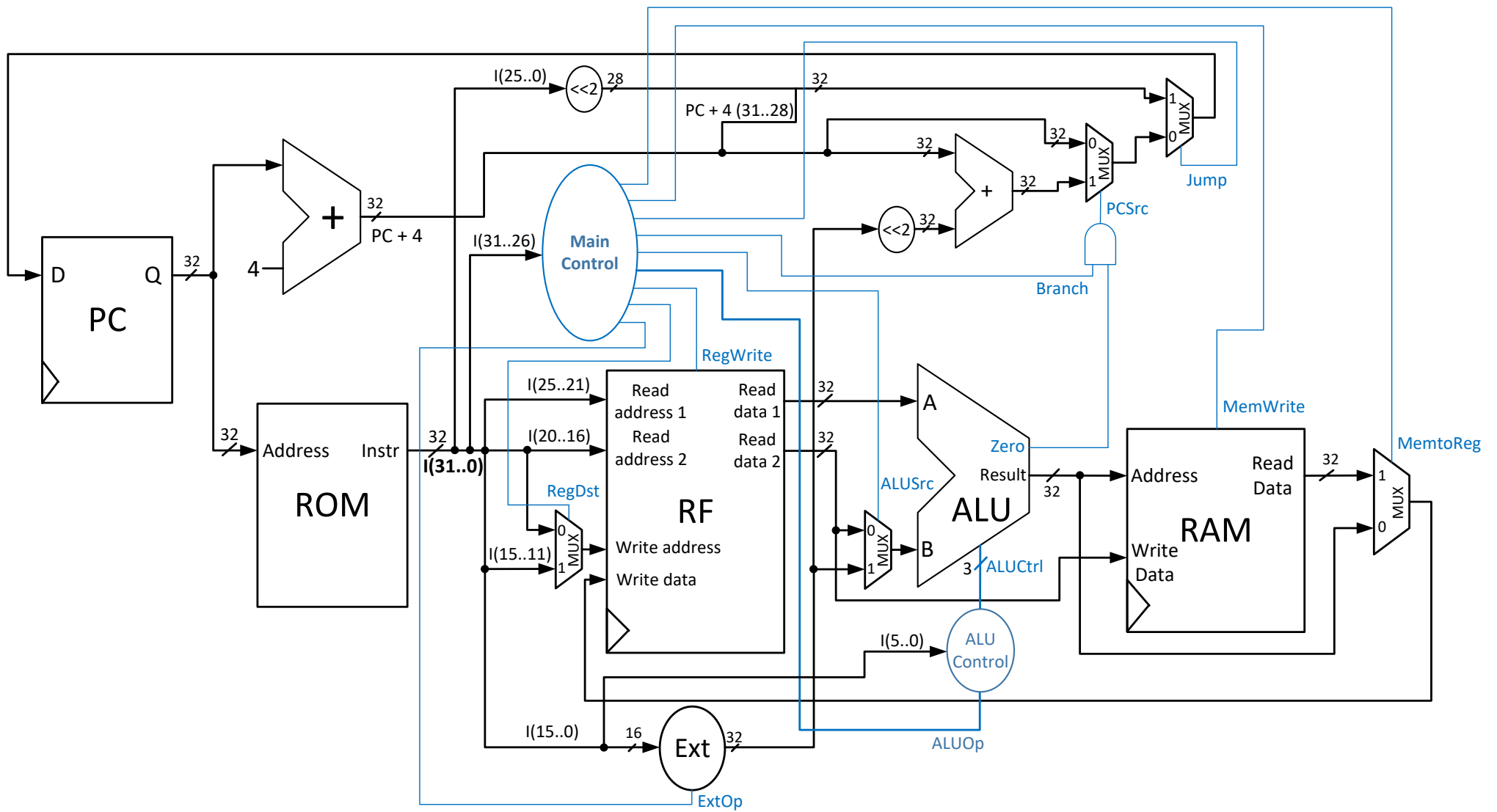


Fig. 9.25. Schema completă a procesorului MIPS 32 ciclu unic, calea de date și unitatea de control



## 10. Programe, compilare, execuție

În acest capitol se va discuta (sumar și simplificat) procesul de transformare a unei secvențe de program în cod mașină, care este ulterior executat de către procesor. În contextul unui calculator real, la pornire se încarcă automat în memorie și se lansează în execuție o parte importantă din sistemul de operare. Sistemul de operare este un ansamblu de programe, scrise în același cod mașină interpretabil de către procesor, care gestionează resursele fizice și programele într-un calculator. Pentru cititorii interesați de acest subiect, se recomandă seria de cărți cu titlul *Sisteme de Operare Moderne*, de Andrew S. Tanenbaum, în limba română (Tanenbaum, 2004), sau, în limba engleză (ediție mai nouă), (Tanenbaum & Bos, 2014).

Primul pas în ciclul de viață al unui program este descrierea acestuia într-un anumit limbaj de programare, într-un mediu de programare. Apoi, programul este tradus în cod mașină care poate fi interpretat și executat de către procesor. Traducerea este (în principal) sarcina compilatorului. Codul mașină și datele programului (structuri declarate și inițializate în cod – ex. șiruri etc.) sunt salvate într-un fișier executabil pe discul fix. Structura fișierului executabil este specifică sistemului de operare.

Când se dorește execuția programului, sistemul de operare încarcă programul din fișierul executabil în memoria principală a calculatorului. Pe scurt, se încarcă datele și codul binar în zonele disponibile din memoria principală, se actualizează în codul binar încărcat anumite deplasamente care depind de poziția absolută în memoria principală (ex. adresele absolute de salt), după care se execută un salt la prima instrucțiune a programului pentru a începe efectiv execuția. Programul încărcat în memorie și lansat în execuție poartă denumirea de proces.

În continuare se va prezenta un exemplu de program care accesează un șir în memorie și face diferite operații cu elementele șirului. Pentru a face legătura cu limbajele de nivel înalt, se va porni de la o secțiune de cod C și se va traduce acest cod în limbaj de asamblare MIPS, respectiv cod mașină. La finalul capitolului se va discuta execuția secvenței de program de către procesorul proiectat.

### 10.1. Un program simplu. Din C în asamblare

În programul (secțiunea de program) ales ca exemplu se declară un șir A cu 10 elemente, se parcurge șirul pentru a scădea din fiecare element valoarea 2 și se calculează suma elementelor din șir (după scădere). Secțiunea de program este descrisă în continuare (termenul de „secțiune” este mai adecvat, deoarece impune cititorului o vedere de ansamblu – înainte sau după această secțiune se presupune că există alte secțiuni de cod, într-un program părinte).

```
int A[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
int sum = 0;

for( int i = 0; i < 10; i++ )
{
    A[i] = A[i] - 2;
    sum = sum + A[i];
}
```

Pentru a traduce acest cod în limbajul de asamblare MIPS (conform ISA restrâns descris anterior), prima dată trebuie alocate variabilele fie unor locații de memorie, fie unor registre. În limbajele de nivel înalt, aceste sarcini sunt invizibile programatorului, ele fiind rezolvate de către compilator. În cazul codului scris în asamblare (respectiv cod mașină), programatorul trebuie să stabilească unde va fi păstrată fiecare variabilă. Șirurile vor fi alocate în zone de memorie (ale căror adrese se vor reprezenta inițial cu etichete simbolice), iar variabilele în registre libere, dacă sunt folosite frecvent și local, sau în locații de memorie pentru folosirea ulterioară în alte secțiuni de cod:

1. Se presupune că șirul A se află în memoria de date, având primul element la adresa A\_addr, iar următoarele la adrese consecutive, din 4 în 4 octeți (fiecare element este întreg pe 32 de biți)
2. Variabila sum se află în memorie la adresa sum\_addr, dar fiind utilizată de multe ori în buclă, ea va fi stocată temporar în registrul \$5 (se aleg registre libere),
3. Contorul buclei, i, va fi reprezentat de registrul \$1
4. Pentru a accesa fiecare element din șir, este necesar un index care să crească din 4 în 4 (fiecare element este întreg, nu se poate folosi direct i). Se va folosi registrul \$2 ca index în șir
5. Contorul i trebuie comparat cu valoarea 10 pentru a termina bucla. Din setul de instrucțiuni disponibile, doar *beq* permite un salt condiționat, dacă două registre sunt egale: unul va fi \$1, iar celălalt trebuie să reprezinte numărul de iterații. Se alege \$4 pentru a păstra numărul de iterații necesare.

Înainte de a începe scrierea codului se trece în revistă setul de instrucțiuni disponibile, pentru a alege ce instrucțiuni pot fi folosite pentru diferitele operații din cod.

Pentru a inițializa un registru \$x cu 0, există mai multe variante, bazate pe proprietatea registrului 0 din RF, care are permanent valoarea 0. Astfel, registrul \$x va fi operandul destinație pentru o instrucțiune de tip R (*add*, *sub*, *and*, *or*) cu operandii sursă \$0 și \$0.

Inițializarea unui registru \$x cu o altă valoare se poate face cu instrucțiunile *addi* sau *ori* cu destinația \$x, executând operația dintre \$0 și valoarea imediată cu care se dorește inițializarea.

Salturile condiționate se vor implementa cu *beq*, iar cele necondiționate cu *j*. Implementarea mecanismului de buclă se va face cu o pereche de instrucțiuni *beq* / *j*. Instrucțiunea *beq* va testa condiția de terminare la începutul buclei și va executa saltul în afara buclei dacă s-a terminat, iar *j* va fi ultima instrucțiune din buclă și va executa saltul la începutul buclei (repetare). În descrierea în limbaj de asamblare, adresele de salt vor fi reprezentate simbolic cu etichete (ex. *begin\_loop*, *end\_loop*).

Accesarea memoriei se va face cu *lw* / *sw*. Adresa elementului de accesat va fi formată din deplasamentul absolut (adresa de start a șirului) și deplasamentul relativ (registrul folosit pentru indexul elementului curent).

Ținând cont de aspectele subliniate anterior, secvența de instrucțiuni în asamblare, cu comentarii pentru fiecare linie de cod (#), este prezentată pe pagina următoare.

```

0      add $1, $0, $0      # i = 0, contorul buclei
1      addi $4, $0, 10     # se salvează numărul maxim de iterații (10)
2      add $2, $0, $0      # inițializarea indexului locației de memorie
3      add $5, $0, $0      # sum = 0
4  begin_loop: beq $1, $4, end_loop # s-au făcut 10 iterații? dacă da, salt în afara buclei
5          lw $3, A_addr($2) # în $3 se aduce elementul curent din șir
6          addi $3, $3, -2    # $3 = $3 - 2
7          sw $3, A_addr($2) # salvarea noii valori $3 în elementul curent din șir
8          add $5, $5, $3    # se adună la suma parțială din $5 elementul curent
9          addi $2, $2, 4    # indexul următorului element din șir
10         addi $1, $1, 1    # i = i + 1, actualizarea contorului buclei
11         j begin_loop     # salt începutul buclei
12  end_loop: sw $5, sum_addr($0) # salvarea sumei în memorie la adresa sum_addr

```

## 10.2. Un program simplu. Din asamblare în cod mașină

Limbajul de asamblare este ultimul nivel unde se folosește o descriere simbolică pentru diferitele câmpuri din instrucțiune. Aceste simboluri au o anumită semnificație pentru programator (numele instrucțiunii, numele operanzilor, etichete etc). Pentru a fi înțeles de către procesor, programul trebuie reprezentat în cod mașină, pornind de la formatul binar al instrucțiunilor.

Înainte de a începe conversia efectivă, trebuie stabilite valori pentru simbolurile de tip etichetă folosite în descrierea programului în asamblare. Aceste etichete sunt de două tipuri: deplasamente absolute în memoria de date, respectiv deplasamente relative sau pseudo-absolute la instrucțiunile de salt.

Deplasamentele absolute pentru variabilele păstrate în memorie trebuie să reprezinte adresele din memoria de date la momentul rulării programului. Pentru exemplul discutat, se va presupune că șirul A are adresa de început `A_addr = 40`, iar variabila sum se află în memorie imediat după șir, `sum_addr=80` (șirul începe la adresa 40 și ocupă 10 locații a câte 4 octeți fiecare).

Eticheta folosită pentru câmpul imm din instrucțiunea de salt condiționat `beq` arată peste câte instrucțiuni se va sări (dacă este îndeplinită condiția), față de instrucțiunea imediat următoare după `beq`. Pentru a ieși din buclă, trebuie executat un salt la ultima instrucțiune din program (index 12), deci imediatul `end_loop` din `beq` trebuie să aibă valoarea  $7 = 12 - 5$ , unde 5 este indexul instrucțiunii `lw` care urmează după `beq`.

Pentru a stabili valoarea adresei de salt `begin_loop`, trebuie rediscutată împărțirea memoriei de instrucțiuni în zone de 256 MB, pe baza valorii primilor 4 biți din PC (a se revedea paragraful corespunzător din subsecțiunea 9.3.1.3, execuția la *j*). Adresa de salt trebuie să indice la a câta instrucțiune din zona curentă (de 256 MB) se face saltul absolut. Așadar, valoarea acestei adrese se poate stabili numai când se știe unde se va amplasa codul binar al programului în memoria de instrucțiuni (sistemul de operare va actualiza această adresă când încarcă programul în memorie). Pentru exemplul discutat, se merge pe presupunerea (simplificatoare) că programul este pus la începutul memoriei ROM de instrucțiuni, începând cu adresa 0. În acest caz, saltul absolut se face la pseudo-adresa `begin_loop = 4`.

Codul în asamblare este rescris, pe pagina următoare, folosind valoarea numerică a etichetelor.

0	add \$1, \$0, \$0	# i = 0, contorul buclei
1	addi \$4, \$0, 10	# se salvează numărul maxim de iterații (10)
2	add \$2, \$0, \$0	# inițializarea indexului locației de memorie
3	add \$5, \$0, \$0	# sum = 0
4	beq \$1, \$4, 7	# s-au făcut 10 iterații? dacă da, salt în afara buclei
5	lw \$3, 40(\$2)	# în \$3 se aduce elementul curent din șir
6	addi \$3, \$3, -2	# \$3 = \$3 - 2
7	sw \$3, 40(\$2)	# salvarea noii valori \$3 în elementul curent din șir
8	add \$5, \$5, \$3	# se adună la suma parțială din \$5 elementul curent
9	addi \$2, \$2, 4	# indexul următorului element din șir
10	addi \$1, \$1, 1	# i = i + 1, actualizarea contorului buclei
11	j 4	# salt începutul buclei
12	sw \$5, 80(\$0)	# salvarea sumei în memorie la adresa 80

Sub această formă, unde localizarea datelor și a codului în memorie este stabilită, programul se transformă în cod mașină. Fiecare instrucțiune este codificată conform tipului de care aparține (opcode și funct se codifică conform descrierii din ISA, discutată în secțiunile 9.2.4 și 9.2.5), operandii de tip registru sunt codificați în binar pe câmpurile de 5 biți din format, iar valorile imediate sunt codificate în binar (complement față de 2) pe câmpul de 16 biți din format.

În continuare, se explică în detaliu conversia la cod mașină pentru câteva instrucțiuni, din fiecare tip.

Instrucțiunea add \$1, \$0, \$0 este de tip R. Codul de operație este 000000, iar câmpul funct are valoarea 100000. Adresele celor trei registre sunt reprezentate pe câte 5 biți în binar.

add \$1, \$0, \$0      cod mașină: 000000 00000 00000 00001 00000 100000

Instrucțiunea addi \$3, \$3, -2 este de tip I. Codul de operație este 001000, cele două registre se vor codifica pe câmpurile corespunzătoare, iar imediatul pe cei 16 biți alocați în format. Fiind o valoare negativă, reprezentarea binară a imm se va face folosind complementul față de 2.

addi \$3, \$3, -2      cod mașină: 001000 00011 00011 1111111111111110

Instrucțiunea beq \$1, \$4, 7 este de tip I. Codul de operație este 000100, cele două registre, la fel ca la *addi*, se vor codifica pe câmpurile corespunzătoare, iar imediatul 7 pe cei 16 biți alocați în format (număr pozitiv, direct în binar).

beq \$1, \$4, 7      cod mașină: 000100 00001 00100 000000000000111

Instrucțiunea j 4 este de tip J. Codul de operație este 000010, iar adresa pseudo-absolută 4 se va codifica pe cele 26 de poziții de bit din formatul de instrucțiune.

j 4      cod mașină: 000010 000000000000000000000000100

Cu aceeași abordare se completează codul mașină în format binar pentru toate instrucțiunile și, pentru o reprezentare mai compactă, se poate specifica și reprezentarea codului mașină în hexazecimal (8 cifre în hexazecimal, câte una pentru fiecare 4 biți).

		Cod binar	Cod hexa
0	add \$1, \$0, \$0	000000 000000 000000 00001 00000 100000	0000 0820
1	addi \$4, \$0, 10	001000 000000 00100 0000000000001010	2004 000A
2	add \$2, \$0, \$0	000000 000000 000000 00010 00000 100000	0000 1020
3	add \$5, \$0, \$0	000000 000000 000000 00101 00000 100000	0000 2820
4	beq \$1, \$4, 7	000100 00001 00100 0000000000000111	1024 0007
5	lw \$3, 40(\$2)	100011 00010 00011 0000000000101000	8C43 0028
6	addi \$3, \$3, -2	001000 00011 00011 1111111111111110	2063 FFFE
7	sw \$3, 40(\$2)	101011 00010 00011 0000000000101000	AC43 0028
8	add \$5, \$5, \$3	000000 00101 00011 00101 00000 100000	00A3 2820
9	addi \$2, \$2, 4	001000 00010 00010 0000000000000100	2042 0004
10	addi \$1, \$1, 1	001000 00001 00001 0000000000000001	2021 0001
11	j 4	000010 00000000000000000000000100	0800 0004
12	sw \$5, 80(\$0)	101011 00000 00101 0000000001010000	AC05 0050
13	...	...	...

### 10.3. Execuția programului

Se presupune că programul reprezentat în cod mașină este încărcat în memoria ROM de instrucțiuni, începând cu adresa 0, iar datele programului (șirul A) sunt încărcate la adresele prestabilite în memoria de date (conform deplasamentelor prezente în codul mașină). După cum s-a menționat în introducerea acestui capitol, încărcarea programului în memoria principală este sarcina sistemului de operare sau a altor programe specializate.

Pentru a începe execuția, se încarcă contorul de program cu adresa primei instrucțiuni din program, 0 în acest caz. Pe fiecare perioadă de ceas, se va executa câte o instrucțiune din program, după tiparul discutat în capitolul anterior. Rezultatul (pentru instrucțiunile care actualizează un registru sau o locație de memorie) se scrie în elementul de memorare la finalul perioadei de ceas, pe frontul crescător, devenind disponibil pentru instrucțiunile care urmează.

După parcurgerea de 10 ori a buclei formate din secvența de instrucțiuni 4–11, instrucțiunea 4 va executa saltul la instrucțiunea 12 (se iese din buclă), care memorează valoarea sumei. După execuția instrucțiunii 12, se va trece la următoarele instrucțiuni (adresele 13, 14 etc.) – fie există alte secvențe de instrucțiuni din același program, fie se va trece, prin salt, la secvențe de instrucțiuni ale sistemului de operare etc. (aici trebuie privită funcționarea procesorului într-un context mai general).





## 11. Extinderea procesorului

Procesorul MIPS proiectat în capitolul 9 implementează un număr redus de instrucțiuni. Ca abordare generală, pornind de la procesorul proiectat pentru un set restrâns, se adaugă restul instrucțiunilor dorite. Alte posibilități de extindere, pentru a avea capabilitățile unui procesor real, pot fi: mecanisme pentru tratarea excepțiilor (evenimente externe, de la dispozitivele conectate, sau interne – de exemplu: depășiri aritmetice), interfațare cu sisteme de intrare-ieșire etc. Aceste îmbunătățiri nu se vor prezenta în detaliu, fiind tratate pe larg în literatură (Harris & Harris, 2013), (Patterson & Hennessy, 2013).

În acest capitol se va prezenta un exemplu de extindere a procesorului cu o nouă instrucțiune, prin aplicarea pașilor de proiectare folosiți deja. De-a lungul reproiectării este foarte important ca instrucțiunile suportate anterior de către procesor să rămână valide (procesorul să le poată executa în continuare, corect). În funcție de complexitatea instrucțiunii, sunt necesare modificări semnificative ale căii de date sau doar la nivelul unității de control.

Se va extinde procesorul pentru a suporta instrucțiunea *jalr* (jump and link register). Sintaxa instrucțiunii este:

*jalr* \$rs

Această instrucțiune execută un salt necondiționat la adresa conținută în \$rs și (“and link”) salvează adresa instrucțiunii care urmează după *jalr* în \$31 (registrul cu indexul 31 din RF). Salvând adresa următoarei instrucțiuni se creează posibilitatea revenirii din subrutine (apelate cu *jalr*).

Transferul RTL este:

$RF[31] \leftarrow PC+4, PC \leftarrow RF[rs]$

Procesul de extindere (reproiectare) începe de la stabilirea unui format de instrucțiune (în cazul în care nu este impus). Pentru instrucțiunea *jalr*, trebuie să existe în format câmpul rs, deci formatul de instrucțiune poate fi de tip R sau de tip I. Dar, deoarece formatul de tip R a fost folosit strict pentru instrucțiuni care execută operații între două registre și salvează rezultatul în al treilea, se va alege formatul de tip I. Câmpurile nefolosite se vor codifica cu 0 (chiar dacă valoarea acestor câmpuri nu influențează execuția). Sunt relevante câmpurile opcode, unde se alege o valoare unică, nefolosită la instrucțiunile existente, și câmpul rs:

<i>jalr</i>	101111	rs	00000	0000000000000000
-------------	--------	----	-------	------------------

*Remarcă:* Dacă se consultă manualul oficial pentru arhitectura setului de instrucțiuni (MIPS Technologies, MIPS® Architecture for Programmers, Volume II-A: The MIPS32® Instruction Set Manual, Revision 6.05, 2016) se va constata că instrucțiunea *jalr* are format de tip R, operația propriu-zisă fiind codificată în câmpul funct. Dacă s-ar alege formatul de tip R, ar fi necesară modificarea unității de control prin legarea câmpului funct din instrucțiune la unitatea *Main Control*. În urma acestei modificări, semnalele generate de *Main Control* vor depinde atât de opcode cât și de funct.

În continuare, se trece succint prin cei cinci pași de proiectare.

În primul pas se analizează RTL-ul pentru a identifica cerințe pentru calea de date. Spre deosebire de proiectarea inițială a procesorului, aici se ține cont de calea de date existentă și se identifică doar eventuale cerințe suplimentare. În acest caz, elementele prezente în RTL-ul instrucțiunii există deja în procesor. Trebuie să existe posibilitatea de a scrie în RF la adresa cu valoarea 31.

La pasul doi de proiectare, pentru *jalr*, nu este necesară adăugarea de noi componente pentru căile de date.

La pasul trei, de asamblare a căii de date, trebuie extinsă calea de date existentă pentru a suporta transferurile RTL ale instrucțiunii *jalr*. În cazul în care există deja anumite conexiuni utile în procesor, se vor folosi cele existente. Pentru instrucțiunea curentă sunt necesare noi conexiuni.

Pentru transferul  $RF[31] \leftarrow PC+4$  se va extinde calea de date astfel:

- conexiune de la ieșirea sumatorului  $PC + 4$  la intrarea Write data a RF. Implicit, deoarece pe intrarea respectivă exista deja o legătură, se va introduce un nou multiplexor cu semnalul de selecție *Jalrsel* (=1 pentru a trece  $PC + 4$ )
- valoarea 31 trebuie să ajungă pe Write address a RF. Este necesar un multiplexor suplimentar care va selecta între ce era deja legat și valoarea 31. Prin codificarea convenabilă a selecției, se va folosi același semnal de selecție *Jalrsel* (=1 pentru a trece valoarea 31).

Pentru transferul  $PC \leftarrow RF[rs]$  este necesară o legătură de la ieșirea Read data 1 a RF la intrarea D a PC. Se va introduce încă un multiplexor pentru a realiza această legătură, folosind același semnal de selecție *Jalrsel* (=1 pentru a trece valoarea  $RF[rs]$ ).

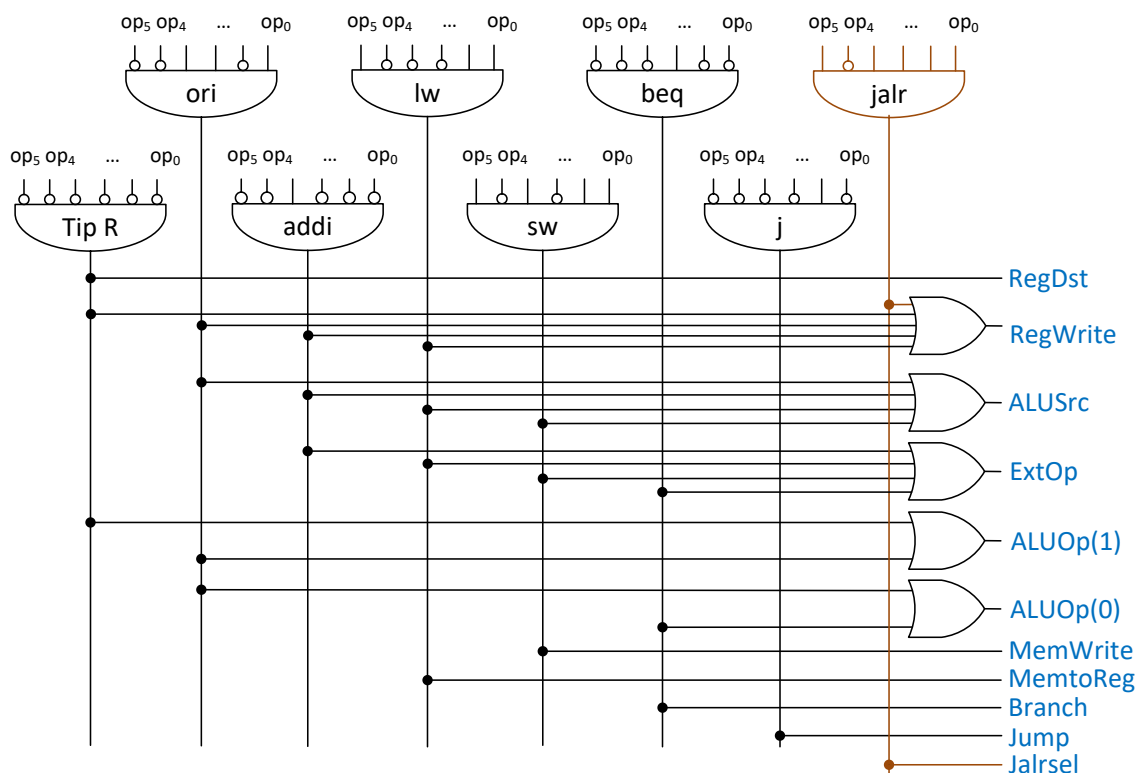
Calea de date extinsă este prezentată în Fig. 11.2 (pagina 115).

Al patrulea pas constă în definirea completă a valorilor pentru semnalele de control. Instrucțiunea trebuie să scrie în blocul de registre, deci *RegWrite* trebuie activat. Instrucțiunea nu modifică memoria de date, deci semnalul de validare a scrierii în acest bloc trebuie să fie inactiv (=0). Exceptând *Jalrsel*, restul semnalelor nu contează:

Instr.	Reg Dst	Reg Write	ALU Src	Ext Op	ALUOp <sub>1..0</sub>	Mem Write	Memto Reg	Branch	Jump	Jalrsel
<b>jalr</b>	x	1	x	x	xx	0	x	x	x	1

Pentru celelalte instrucțiuni suportate de procesor, semnalul *Jalrsel* va avea valoarea 0.

Ultimul pas de proiectare constă în extinderea unității de control pentru a suporta noua instrucțiune. Se adaugă o poartă ȘI cu 6 intrări pentru a detecta codul de operație al instrucțiunii *jalr*, iar semnalul *Jalrsel* se va genera direct din ieșirea acestei porți ȘI (doar pentru această instrucțiune el trebuie să fie 1, deci nu este necesară o poartă SAU pentru a-l genera). Deoarece *RegWrite* trebuie să fie 1, se va lega ieșirea din poarta ȘI la poarta SAU care generează acest semnal. Unitatea de control extinsă este prezentată în Fig. 11.1.

Fig. 11.1. Unitatea de control extinsă pentru a suporta instrucțiunea *jalr*Soluție alternativă (pe scurt)

Procesul de reproiectare, pentru a implementa o nouă instrucțiune, permite de obicei obținerea mai multor soluții. De exemplu, pentru instrucțiunea *jalr*, la alegerea formatului de tip I pentru instrucțiune, este importantă observația că instrucțiunile de tip I pot face scrieri în blocul de registre la adresa indicată de câmpul *rt* din instrucțiune. Astfel, se poate codifica la valoarea 31 ( $=11111_2$ ) câmpul *rt* în formatul binar al instrucțiunii:

<i>jalr</i>	101111	rs	11111	00000000000000000000
-------------	--------	----	-------	----------------------

Cu această codificare, din prima soluție nu mai este necesar multiplexorul care permite valorii 31 să ajungă pe câmpul Write address al RF. Pentru a scrie la adresa 31, este suficient ca semnalul **RegDst** să aibă valoarea 0: prin multiplexorul controlat de **RegDst** va trece valoarea 31 prezentă pe *rt* în codul mașină al instrucțiunii.



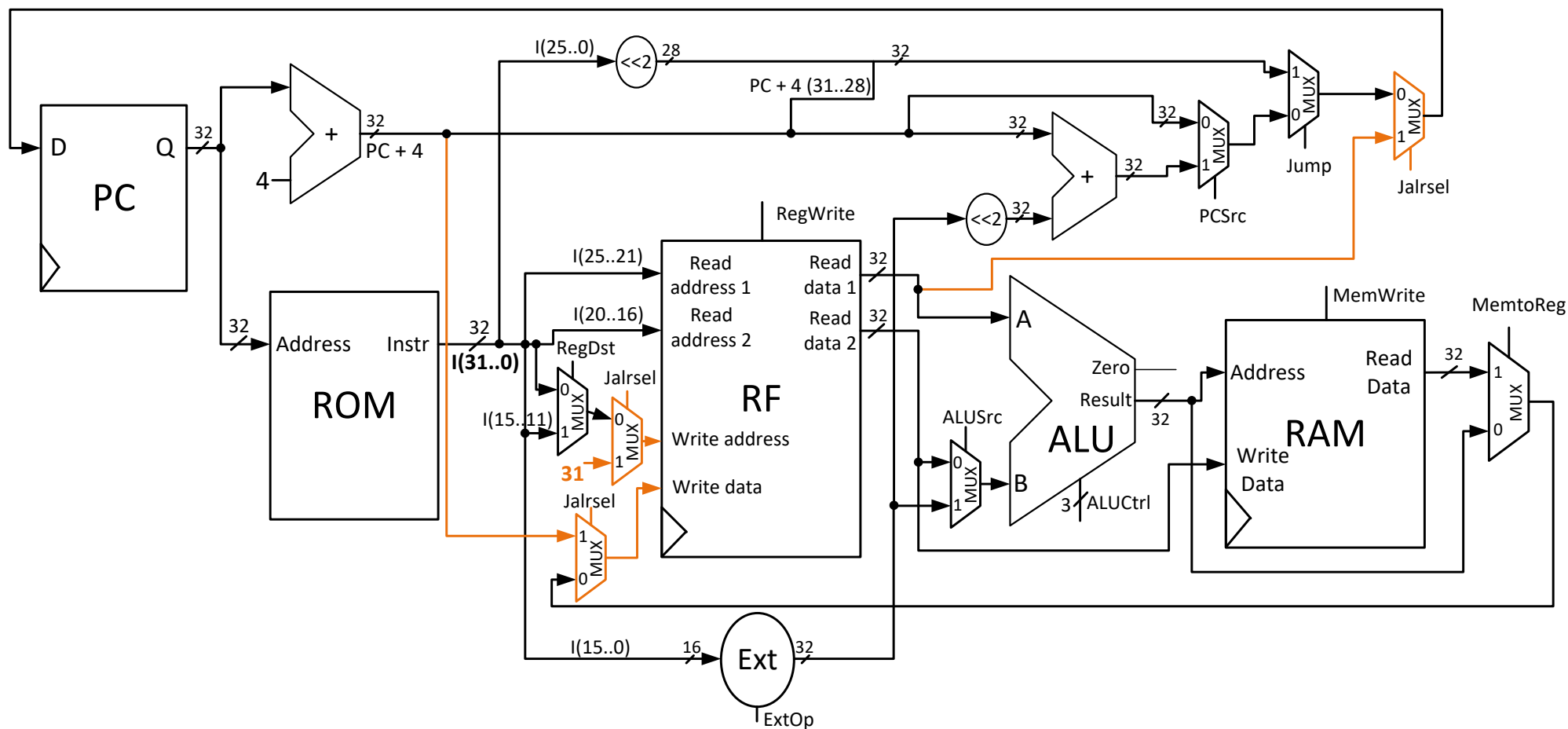


Fig. 11.2. Calea de date a procesorului MIPS extinsă pentru a suporta instrucțiunea `jalr`. Modificările sunt evidențiate cu culoare



## Bibliografie

- Coussy, P., Gajski, D. D., Meredith, M., & Takach, A. (2009). An Introduction to High-Level Synthesis. *IEEE Design & Test of Computers*, 26(4), 8-17.
- Harris, D., & Harris, S. (2013). *Digital Design and Computer Architecture, 2nd Edition*. Waltham, SUA: Morgan Kaufmann.
- Hennessy, J. L., & Patterson, D. A. (2011). *Computer Architecture: A Quantitative Approach, 5th Edition*. USA: Morgan Kaufmann.
- Hennessy, J. L., & Patterson, D. A. (2017). *Computer Architecture: A Quantitative Approach, 6th Edition*. USA: Morgan Kaufmann.
- MIPS Technologies, I. (2014, August 20). MIPS® Architecture For Programmers, Volume I-A: Introduction to the MIPS32® Architecture, Revision 6.01. Preluat de pe <https://www.mips.com/products/architectures/mips32-2/>
- MIPS Technologies, I. (2016, Mai 20). MIPS® Architecture for Programmers, Volume II-A: The MIPS32® Instruction Set Manual, Revision 6.05. Preluat de pe <https://www.mips.com/products/architectures/mips32-2/>
- Patterson, D. A., & Hennessy, J. L. (2002). *Organizarea și Proiectarea Calculatoarelor. Interfața Hardware/Software, traducere după ediția a 2-a*. București: ALL EDUCATIONAL.
- Patterson, D. A., & Hennessy, J. L. (2013). *Computer organization and design: the hardware/software interface, 5th Edition*. USA: Morgan Kaufmann.
- Tanenbaum, A. S. (2004). *Sisteme de Operare Moderne, Ediția a 2-a*. București: Byblos.
- Tanenbaum, A. S., & Bos, H. (2014). *Modern Operating Systems, 4th Edition*. USA: Pearson Prentice-Hall.
- Văcariu, L., & Creț, O. A. (2012). *Probleme de proiectare logică a sistemelor numerice = Logic design problems for digital systems, Ediția a 2-a*. Cluj-Napoca, România: U. T. Press.